Module 1: Introduction

Information about the course is on the website:
https://www.student.cs.uwaterloo.ca/~cs115/

Contact info for all course staff is there, but if in doubt, here are the key email addresses:

Instructor	Cameron Morland – cjmorland@uwaterloo.ca
Coordinator	Barbara Daly – bmzister@uwaterloo.ca
ISAs	cs115@uwaterloo.ca

In this course we will be using DrRacket.

Exercise

Install DrRacket www.racket-lang.org/download/
Choose Distribution: **Racket** and Variant: **Regular**.
If you have any trouble, ask in the discussion forum.

Course components

Class slides available on the course web page. These contain exercises for you to practice and discuss in the discussion boards.
Complete a reflection on your work, due each week on Sunday.

Labs available on the course web page. The ISAs will be running sessions where you can discuss these with other students. These are due most weeks on Monday.

Assignments available on the course web page. These are due most weeks on Wednesdays.

Midterm and Final assessments will similar to assignments. More details as we get closer.

Marking Scheme

Details are on the course website.

70%	Assignments
8%	Midterm Assessment
14%	Final Assessment
4%	Weekly reflections
4%	Participation in discussion boards
3%	Lab Bonus

To pass the course you must pass both:

- the assignments component and
- the weighted average of the midterm/final components.

Review the Policies on course web page. In particular:

> *"In order to maintain a culture of academic integrity, members of the University of Waterloo community are expected to promote honesty, trust, fairness, respect and responsibility. [Check www.uwaterloo.ca/academicintegrity/ for more information.]"*

Participation

Computer programming is *not* a solitary task.
Programmers interact routinely with others:

- Many workplaces hold a Daily Scrum, a brief meeting where each member gives an update on what they're doing.
- Programmers interact frequently with their peers in small conversations in person or through discussion tools (Mattermost, IRC, Slack, etc).

Participation

There are two ways that *participation* will directly affect your grade in this course:

1. Weekly *reflections*.
   In Learn, you will write a short statement on what you have accomplished this week, and what you have not yet accomplished.
   You can think of this as being like a scrum report.
   For full marks, complete a reflection every week, any time Friday – Sunday.
2. Use of discussion boards.
   Ask a question, answer a question, or otherwise participate meaningfully in discussions.
   For full marks, make at least one meaningful post every week, any time Monday – Sunday.

First Week Tasks

Be sure you do all the following:

1. Install DrRacket on your computer: www.racket-lang.org/download/
2. Find out about your labs, and participate in the first one.
3. From the website www.student.cs.uwaterloo.ca/~cs115/ download the course notes and review the course details, including survival guide, marking scheme, and grade appeals policy.
4. Find the style guide on the course website. Look it over, and get in the habit of referencing it for every assignment.
5. Bookmark the course textbook, www.htdp.org , and read the appropriate sections.
6. Complete Assignment 00.

Assignments

Most of your learning comes from struggling with material. You learn little from merely copying work, or even ideas, from another.

! All assignments are to be done individually.

!

- Don't look at someone else's programs written for an assignment.
- Don't show your programs to other students.
- Don't search on the web or in books other than the textbook for answers to assignment questions, or even for hints. Ask in discussion instead!

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

Assignments

A few pieces of advice:

- Start your assignments early.
- Make sure you have time to test thoroughly, and fix your code when needed! Bring questions to the discussion board as soon as possible.
- Go over your graded assignments: learn from your mistakes.
- Do your labs early; what you learn from them will help on the assignments.

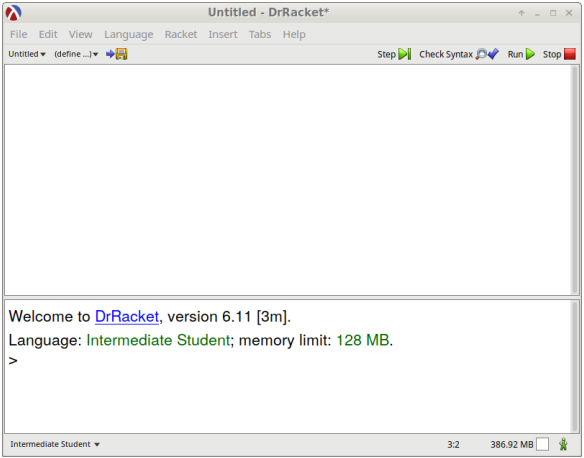
What is the difference between Assignments and Labs?

	Labs	Assignments
Questions:	Short, relatively easy questions for practice.	Longer, more involved questions for consolidation.
Time Needed:	$\sim$ 1 hour	3–8 hours, or more. Start early!
Marking:	Correctness only. When you submit each lab (2-12) to MarkUs, the system will test your functions. If you pass at least half the tests, you get the mark for that lab.	Correctness contributes half the marks. But you need to pass all or nearly all of the tests to get these marks. Test your own code! The other marks come from the design recipe, including testing, and style. Once we discuss testing, you need to write good tests for your code.
Grades:	3% bonus mark.	70%; lowest assignment is dropped.

Exercise

Install DrRacket www.racket-lang.org/download/
If you have any trouble, ask in the discussion forum.

When you run it, you should a window that looks something like this:



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.

By hand, the answer is six: $\{1, 2, 3, 4, 6, 12\}$ are the only natural numbers that divide 12.

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

> **!** How many natural numbers is 5 218 303 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.)

The Design Process

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.

The Design Process

We will cover the whole process of designing programs.

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

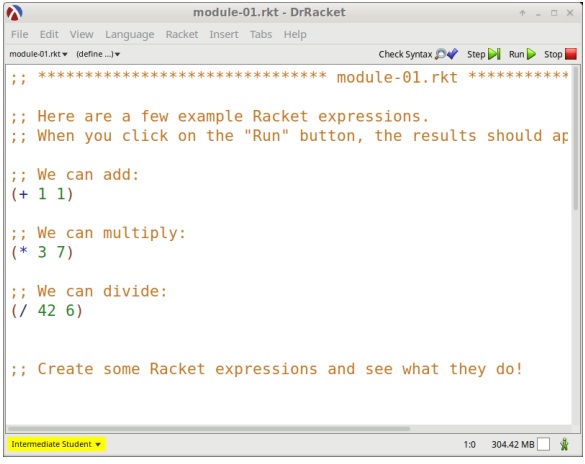
Careful use of design processes can save time and reduce frustration, even with the fairly small programs written in this course.

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 other's programs, and making your programs understandable

Simple calculations in Racket

Ex. Make sure you have DrRacket installed, and run it.
Download the Module Interface File, save it, and open it in DrRacket.



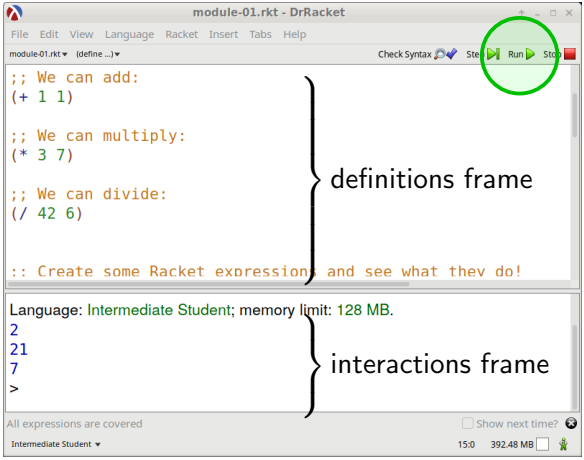
The interface is further described in the DrRacket Documentation. Read it and use the discussion forum if you have questions.

Ex. Click "Run".

We see two "frames":

1. *definitions* and
2. *interactions*.

We will write programs in the definitions frame. Then when we click "Run", the results will appear in the interactions frame.



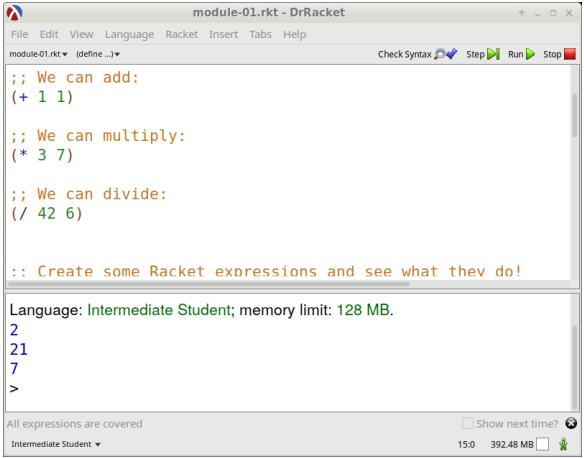
Note the output in the interactions frame:

- `(+ 1 1)` calculates $1 + 1$, which is 2
- `(* 3 7)` calculates 3×7, which is 21
- `(/ 42 6)` calculates $42/6$, which is 7.

Exercise

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

$2 + 3$ 2×3 $44 - 2$



If I want to indicate that I am evaluating an expression, I will use a little arrow `=>` between the expression and the value that it evaluates to.
For example, because `(+ 1 1)` evaluates to `2`, I could write `(+ 1 1) => 2`.
Similarly, `(* 3 7) => 21` and `(/ 42 6) => 7`.

! `=>` is not code. It is a symbol we use to indicate what the code does.

Vocabulary of mathematics

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 **returns** 2.

Order of Operations

Consider: if we want to calculate $3 - 2 + 10/2$ in Mathematics, we need to know *order of operations*. For example, that we do division before addition.
If we didn't want to remember order of operations, we could always add brackets to clarify what we mean: that is, write $((3 - 2) + (10/2))$ instead.

For simplicity, Racket does not consider order of operations. Instead, it requires us to write all the brackets. And the function always comes at the beginning, before the arguments.
So $(3 - 2)$ in Racket we write `(- 3 2)`; $(10/2)$ we write `(/ 10 2)`. and $((3 - 2) + (10/2))$ we write `(+ (- 3 2) (/ 10 2))`.
Note that `(+ (- 3 2) (/ 10 2)) => 6`, the same value we get from the mathematical expression.

Exercise

Rewrite each mathematical expression in Racket.
Click "Run", and make sure you understand what happens.

$3 \times 4 + 2$ $\qquad$ $\dfrac{2+4}{5-1}$ $\qquad$ $3(1 + (6/2 + 5))$

Racket Expressions

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.

Exercise

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) 3)`
2. `(+ (* 2 4)`
3. `(5 * 14)`
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.
- `(+ (* 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.

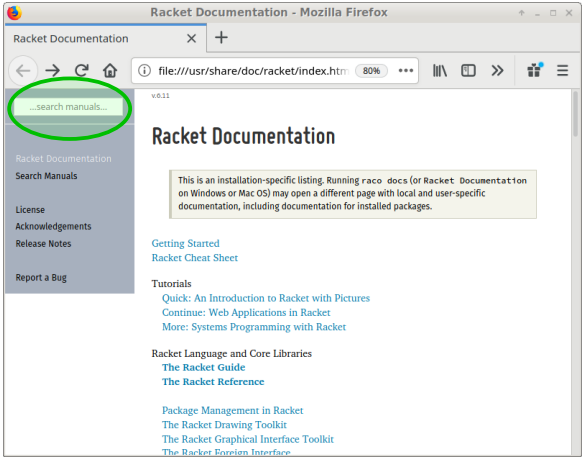
- `(* + 3 5 2)` contains a semantic error since we cannot multiply "plus" with 3, 5, and 2.
- `(/ 25 0)` contains a semantic error since we cannot divide by zero.

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 the menus select:
Help→*Racket Documentation*.

This brings up the web browser, like this →



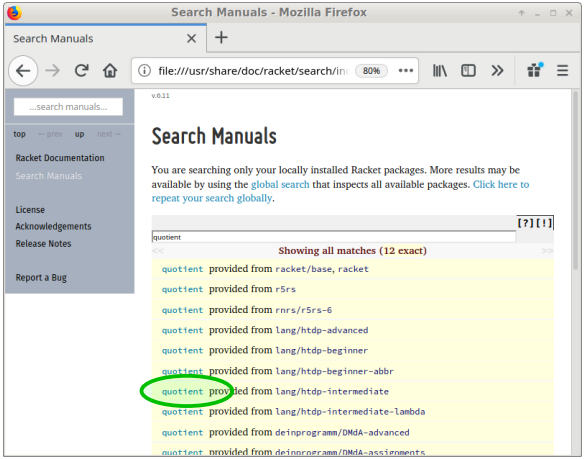
Exercise

In the *...search manuals...* field type the name of a function, and hit *Enter*.
For example, `quotient`.

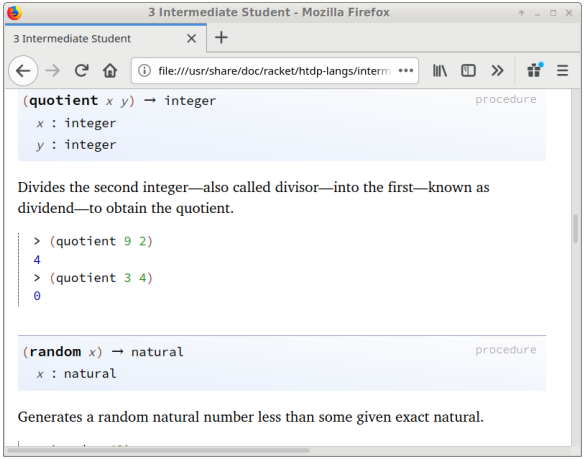
There is separate documentation for different language variants.
We are using the *How To Design Programs-Intermediate Student* variant, so

Exercise

click on `quotient` beside where it says *lang/htdp-intermediate*.



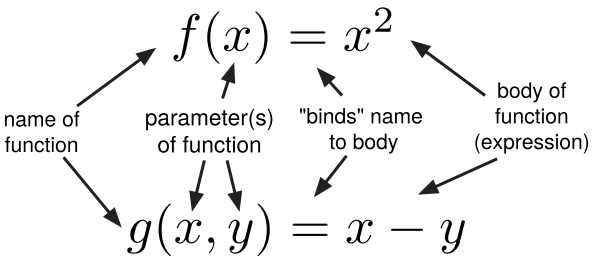
Finally we see information about the function we are interested in: $\longrightarrow$



Exercise

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

`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.

Defining functions in Racket

To translate $f(x) = x^2$ and $g(x, y) = x - y$ into Racket, in the definitions frame, type:

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

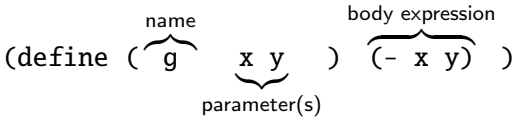
Now we can write expressions that use these new functions.

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`.

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.



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`.

Identifiers

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 `` `'" ``

A few examples:

```
(define (g x y) (- x y))
(define (square-it! it!) (* it! it!))
(define (2remainder? n) (remainder n 2))
```

Exercise

Type these into the definitions frame. Also write expressions that use these functions, and make sure you understand what happens.

Reviewing our observations

As with Mathematical functions:

- 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.
- Different functions may use the same parameter name; there is no problem with
  ```
  (define (f x) (* x x))
  (define (g x y) (- x y))
  ```
- Parameter order matters. The following two functions are **not** the same:
  ```
  (define (g x y) (- x y))
  (define (g y x) (- x y))
  ```

Constants in Racket

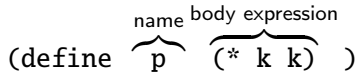
Racket also allows a special kind of value called a **constant**:

```
(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 the 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 has at most one semantic interpretation.

[1] from *Syntactic Structures* by Noam Chomsky, 1957.

Exercise

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.

A model of Racket semantics

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:

```
=> (* 3 (+ 1 (+ 3 5)))
```

Ex. Complete the interpretation of `(* 3 (+ 1 (+ (/ 6 2) 5)))`

Now consider the following program:

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

The function `(f 2)` is bound to `(* 2 2)`, so simplify:

```
=> (g (* 2 2) (g 3 1))
```

Ex. Complete the interpretation of `(g (f 2) (g 3 1))`

Rules for evaluating a Racket expression

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) => 5` before the substitution `(+ 5 7) => 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.)

substitution rules

Built-in function application use mathematical rules.
E.g. `(+ 3 5)` => `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

```
(define x 3)
```

then to evaluate

```
(* x (+ x 5))
```

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

```
=> (* 3 (+ x 5))
```

Now evaluate the second argument, which is the expression `(+ x 5)`. And its first argument is another copy of the constant `x`. So substitute this copy:

```
=> (* 3 (+ 3 5))
=> (* 3 8)
=> 24
```

Note we did not replace both copies of `x` in one step! We replace only the single copy.

User-defined function application a function is 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`. For example,

```
(define (foo a b) (+ a (- a b)))
(foo 4 3)
=> (+ 4 (- 4 3))
```

Note: each `vi` must be a value. To evaluate `(foo (+ 2 2) 3)`, *do not* substitute `(+ 2 2)` for `a`, to give `(+ (+ 2 2) (- (+ 2 2) 3))`.

Always evaluate the arguments first.

Here is an example of tracing a function containing a user-defined function:

```
(foo (+ 2 2) 3)
=> (foo 4 3)
=> (+ 4 (- 4 3))
=> (+ 4 1)
=> 5
```

Beware of tricksters...

Type in the code and see what each evaluates to. Make sure you understand why.

Exercise

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

Exercise

```
(define (huh? huh?) (+ huh? 2))
(huh? 1/2)
```

Exercise

```
(define y 3)
(define (g x) (+ x y))
(g 5)
```

Exercise

```
(define z 3)
(define (h z) (+ z z))
(h 6.4)
```

Tracing a program

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 `(sqr 2 3)` cannot be simplified since `sqr` has only one parameter.

Racket has a feature call the *Stepper* that traces code automatically. We're not going to discuss it, but you can experiment with it if you like.

Tracing practice

Exercise

Trace the program to determine what the result should be.
Then remove the `;` and run it in Racket to check your work.

```
; (+ (remainder (- 10 2) (quotient 10 3)) (* 2 3))
```

Exercise

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.)

Tracing practice

Exercise

Trace the program to determine what the result should be.
Then remove the `;` and run it in Racket to check your work:

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

```
; (define (disc a b c) (sqrt (- (sqr b) (* 4 (* a c)))))
; (define (proot a b c) (/ (+ (- 0 b) (disc a b c)) (* 2 a)))
; (proot 1 3 2)
```

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.

Further Reading: *How to Design Programs*, Prologue.

Before we begin the next module, please

- Read the Survival Guide on assignment style and submission.
- Read the Style Guide, Sections 1-4 and 6.