# 2: The Design Recipe, And More Racket

## Programs as communication

Every program is an act of communication:

## Programs as communication

Every program is an act of communication:

- Between you and the computer

## Programs as communication

Every program is an act of communication:

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

## 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

## 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

Only-for-humans **comments** in Racket programs:
from a semicolon (**;**) to the end of the line.

The design recipe is a development process that leaves behind written explanation of the development.

The design recipe is a development process that leaves behind written explanation of the development.

It results in a trusted (tested) function which future readers (you or others) can understand.

The design recipe is a development process that leaves behind written explanation of the development.

It results in a trusted (tested) function which future readers (you or others) can understand.

Please use the design recipe for every function you write in this course.

## The five design recipe components

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

1. **Contract:** Describes what type of arguments the function consumes and what type of value it produces.
2. **Purpose:** Describes what the function is to compute.

1. **Contract:** Describes what type of arguments the function consumes and what type of value it produces.
2. **Purpose:** Describes what the function is to compute.
3. **Examples:** Illustrating the use of the function.

## The five design recipe components

1. **Contract:** Describes what type of arguments the function consumes and what type of value it produces.
2. **Purpose:** Describes what the function is to compute.
3. **Examples:** Illustrating the use of the function.
4. **Definition:** The Racket definition (header and body) of the function.

1. **Contract:** Describes what type of arguments the function consumes and what type of value it produces.
2. **Purpose:** Describes what the function is to compute.
3. **Examples:** Illustrating the use of the function.
4. **Definition:** The Racket definition (header and body) of the function.
5. **Tests:** A representative set of function applications and expected values.

# Using the design recipe

We'll write a function which squares two numbers and sums the results.

We'll write a function which squares two numbers and sums the results.

**Mathematically:**

sum-of-squares: $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$

We'll write a function which squares two numbers and sums the results.

**Mathematically:**

sum-of-squares: $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$

**In Racket:**

```
; sum-of-squares: Number Number -> Number
; Purpose: produces sum of squares of x and y
; Examples:
(check-expect (sum-of-squares 3 4) 25)
(check-expect (sum-of-squares 0 2.5) 6.25)

(define (sum-of-squares x y)
   (+ (* x x) (* y y)))

; Tests:
(check-expect (sum-of-squares 0 0) 0)
(check-expect (sum-of-squares -2 7) 53)
```

## Types in contracts

`Number`: any Racket numeric value

`Int`: restriction to integers

`Nat`: restriction to natural numbers (including 0)

`Any`: any Racket value

We will see others soon.

1. **Contract**
   - Write the contract first.

1. **Contract**
   - Write the contract first.
2. **Purpose**
   - The header of the function should be written earlier than the purpose.
   - The purpose can then use parameter names in a way that makes their meaning clear.

## Details of the design recipe

1. **Contract**
   - Write the contract first.
2. **Purpose**
   - The header of the function should be written earlier than the purpose.
   - The purpose can then use parameter names in a way that makes their meaning clear.
3. **Examples**
   - Examples should be written earlier than the body.
   - The examples can then serve as a guide to writing the body.

1. **Contract**
   - Write the contract first.
2. **Purpose**
   - The header of the function should be written earlier than the purpose.
   - The purpose can then use parameter names in a way that makes their meaning clear.
3. **Examples**
   - Examples should be written earlier than the body.
   - The examples can then serve as a guide to writing the body.
4. **Definition**
   - Header (name of function and parameters)
   - Body (expression using parameters)

1. **Contract**
   - Write the contract first.
2. **Purpose**
   - The header of the function should be written earlier than the purpose.
   - The purpose can then use parameter names in a way that makes their meaning clear.
3. **Examples**
   - Examples should be written earlier than the body.
   - The examples can then serve as a guide to writing the body.
4. **Definition**
   - Header (name of function and parameters)
   - Body (expression using parameters)
5. **Tests**
   - Tests should be written later than the code body.
   - They 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.

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 **by hand**.

The teaching languages offer a convenient testing method that can also be used for examples.

The teaching languages offer a convenient testing method that can also be used for examples.

```
(check-expect (sum-of-squares 3 4) 25)
(check-within (sqrt 2) 1.414 0.001)
(check-error (/ 1 0) "/: division by zero")
```

The teaching languages offer a convenient testing method that can also be used for examples.

```
(check-expect (sum-of-squares 3 4) 25)
(check-within (sqrt 2) 1.414 0.001)
(check-error (/ 1 0) "/: division by zero")
```

Tests written using these forms are saved and evaluated at the very end of the computation.

The teaching languages offer a convenient testing method that can also be used for examples.

```
(check-expect (sum-of-squares 3 4) 25)
(check-within (sqrt 2) 1.414 0.001)
(check-error (/ 1 0) "/: division by zero")
```

Tests written using these forms are saved and evaluated at the very end of the computation.

This means that examples can be written as code.

DrRacket provides a nice test report.

- Comments rarely needed inside body of functions

# Some notes on style

- Comments rarely needed inside body of functions
- Indent to indicate level of nesting and align subexpressions

# Some notes on style

- Comments rarely needed inside body of functions
- Indent to indicate level of nesting and align subexpressions
- Let DrRacket help you with indentation

- Comments rarely needed inside body of functions
- Indent to indicate level of nesting and align subexpressions
- Let DrRacket help you with indentation
- Avoid overly horizontal or vertical code

# Some notes on style

- Comments rarely needed inside body of functions
- Indent to indicate level of nesting and align subexpressions
- Let DrRacket help you with indentation
- Avoid overly horizontal or vertical code
- Put multiple close parentheses on the same line

# Some notes on style

- Comments rarely needed inside body of functions
- Indent to indicate level of nesting and align subexpressions
- Let DrRacket help you with indentation
- Avoid overly horizontal or vertical code
- Put multiple close parentheses on the same line
- Use reasonable line lengths

## Some notes on style

- Comments rarely needed inside body of functions
- Indent to indicate level of nesting and align subexpressions
- Let DrRacket help you with indentation
- Avoid overly horizontal or vertical code
- Put multiple close parentheses on the same line
- Use reasonable line lengths
- Choose meaningful identifier names

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

## Boolean-valued functions

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

Racket provides many built-in Boolean functions (for example, to do comparisons).

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

Racket provides many built-in Boolean functions (for example, to do comparisons).

`(= x y)` will evaluate to `#true` if numbers `x` and `y` are equal, otherwise it will evaluate to `#false`.

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

Racket provides many built-in Boolean functions (for example, to do comparisons).

`(= x y)` will evaluate to `#true` if numbers `x` and `y` are equal, otherwise it will evaluate to `#false`.

The teaching languages provide the predefined constants `true` and `false`. These are names bound to the appropriate values.

## Other types of comparisons

In order to determine whether the proposition "$x < y$" is true or false, we can evaluate `(< x y)`.

## 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 $>$, $\leqslant$ (written `<=`) and $\geqslant$ (written `>=`).

In order to determine whether the proposition "$x < y$" is true or false, we can evaluate `(< x y)`.

There are also functions for $>$, $\leqslant$ (written `<=`) and $\geqslant$ (written `>=`).

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

```
; = : Number Number -> Boolean
```

Racket provides the function `not` and the forms `and`, `or` to combine Boolean values. These can be used to test complex relationships.

Racket provides the function `not` and the forms `and`, `or` to combine Boolean values. These can be used to test complex relationships.

**Example**

The proposition "$3 \leqslant x < 7$", which is the same as "$x \in [3, 7)$", can be computationally tested by evaluating `(and (<= 3 x) (< x 7))`.

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

## Some computational subtleties

- The Racket forms `and`, `or` may have more than two arguments.
- The form `and` has value `#true` exactly when all of its arguments have value `#true`.

# Some computational subtleties

- The Racket forms `and`, `or` may have more than two arguments.
- The form `and` has value `#true` exactly when all of its arguments have value `#true`.
- The form `or` has value `#true` exactly when at least one of its arguments has value `#true`.

## Some computational subtleties

- The Racket forms `and`, `or` may have more than two arguments.
- The form `and` has value `#true` exactly when all of its arguments have value `#true`.
- The 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`.

- The Racket forms `and`, `or` may have more than two arguments.
- The form `and` has value `#true` exactly when all of its arguments have value `#true`.
- The 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`.
- DrRacket only evaluates as many arguments of `and` and `or` as is necessary to determine the value.

- The Racket forms `and`, `or` may have more than two arguments.
- The form `and` has value `#true` exactly when all of its arguments have value `#true`.
- The 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`.
- DrRacket only evaluates as many arguments of `and` and `or` as is necessary to determine the value.

**Example**

```
(and (not (= x 0)) (<= (/ y x) c))
(or (= x 0) (> (/ y x) c))
```

These will never divide by zero.

**Exercise**

Does the following expression evaluate to `#true` or `#false`?

```
(not (and (= 1 2) (> 3 1)))
```

**Exercise**

Does the following expression evaluate to `#true` or `#false`?

```
(not (and (= 1 2) (> 3 1)))
```

**Exercise Solution**

```
(not (and (= 1 2) (> 3 1)))
⇒ (not (and #false (> 3 1)))
⇒ (not (and #false #true))
⇒ (not #false)
⇒ #true
```

**Exercise**

Does the following expression evaluate to `#true` or `#false`?

```
(and (not (> 3 2)) (and (< 1 2) (> 2 1)))
```

**Exercise**

Does the following expression evaluate to `#true` or `#false`?

```
(and (not (> 3 2)) (and (< 1 2) (> 2 1)))
```

**Exercise Solution**

```
(or (not (> 3 2))  (and (< 1 2) (> 2 1)))
⇒  (or (not #true)  (and (< 1 2) (> 2 1)))
⇒  (or #false (and #true (> 2 1)))
⇒  (or #false (and #true #true))
⇒  (or #false #true)
⇒  #true
```

**Exercise**

Can the following expressions result in a divide by 0 error? (Yes/No)

1. `(or (= x 0) (< 0 (/ y x)))`
2. `(or (< 0 (/ y x)) (= x 0))`
3. `(and (= x 0) (< 0 (/ y x)))`
4. `(or (< 0 (/ y x)) (not (= x 0)))`

**Exercise**

Can the following expressions result in a divide by 0 error? (Yes/No)

1. `(or (= x 0) (< 0 (/ y x)))`
2. `(or (< 0 (/ y x)) (= x 0))`
3. `(and (= x 0) (< 0 (/ y x)))`
4. `(or (< 0 (/ y x)) (not (= x 0)))`

**Exercise Solution**

1. No.

**Exercise**

Can the following expressions result in a divide by 0 error? (Yes/No)

1. `(or (= x 0) (< 0 (/ y x)))`
2. `(or (< 0 (/ y x)) (= x 0))`
3. `(and (= x 0) (< 0 (/ y x)))`
4. `(or (< 0 (/ y x)) (not (= x 0)))`

**Exercise Solution**

1. No.
2. Yes.

**Exercise**

Can the following expressions result in a divide by 0 error? (Yes/No)

1. `(or (= x 0) (< 0 (/ y x)))`
2. `(or (< 0 (/ y x)) (= x 0))`
3. `(and (= x 0) (< 0 (/ y x)))`
4. `(or (< 0 (/ y x)) (not (= x 0)))`

**Exercise Solution**

1. No.
2. Yes.
3. Yes.

**Exercise**

Can the following expressions result in a divide by 0 error? (Yes/No)

1. `(or (= x 0) (< 0 (/ y x)))`
2. `(or (< 0 (/ y x)) (= x 0))`
3. `(and (= x 0) (< 0 (/ y x)))`
4. `(or (< 0 (/ y x)) (not (= x 0)))`

**Exercise Solution**

1. No.
2. Yes.
3. Yes.
4. Yes.

# Predicates

Racket provides a number of built-in Boolean functions or **predicates**, such as `even?`, `negative?`, and `zero?`.

Racket provides a number of built-in Boolean functions or **predicates**, such as `even?`, `negative?`, and `zero?`.

We can write our own predicates:

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

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

Racket provides a number of built-in Boolean functions or **predicates**, such as `even?`, `negative?`, and `zero?`.

We can write our own predicates:

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

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

Ending a predicate with a question mark is a convention.
It is not required.

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

A symbol is formed by an initial apostrophe or "quote": `'blue`

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

A symbol is formed by an initial apostrophe or “quote”: `'blue`

The symbol `'blue` is a value just like `6`, but it is more limited computationally.

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

A symbol is formed by an initial apostrophe or “quote”: `'blue`

The symbol `'blue` is a value just like `6`, but it is more limited computationally.

Using a symbol allows a programmer to avoid using numbers to represent names of colours, or of planets, or of types of music.

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

A symbol is formed by an initial apostrophe or "quote": `'blue`

The symbol `'blue` is a value just like `6`, but it is more limited computationally.

Using a symbol allows a programmer to avoid using numbers to represent names of colours, or of planets, or of types of music.

Symbols can be compared using the function `symbol=?`.

```
(define my-symbol 'blue)
(symbol=? my-symbol 'red)
⇒ #false
```

`symbol=?` is the only function we'll use in this course that is applied only to symbols.

Racket also supports strings, such as `"blue"`.

What are the differences between strings and symbols?

Racket also supports strings, such as `"blue"`.

What are the differences between strings and symbols?

- Strings are really compound data
  (a string is a sequence of characters).

Racket also supports strings, such as `"blue"`.

What are the differences between strings and symbols?

- Strings are really compound data
  (a string is a sequence of characters).
- Symbols can't have certain characters in them (such as spaces).

Racket also supports strings, such as `"blue"`.

What are the differences between strings and symbols?

- Strings are really compound data
  (a string is a sequence of characters).
- Symbols can't have certain characters in them (such as spaces).
- It is more efficient to compare two symbols than two strings.

Racket also supports strings, such as `"blue"`.

What are the differences between strings and symbols?

- Strings are really compound data
  (a string is a sequence of characters).
- Symbols can't have certain characters in them (such as spaces).
- It is more efficient to compare two symbols than two strings.
- There are more built-in functions for strings.

Racket also supports strings, such as `"blue"`.

What are the differences between strings and symbols?

- Strings are really compound data
  (a string is a sequence of characters).
- Symbols can't have certain characters in them (such as spaces).
- It is more efficient to compare two symbols than two strings.
- There are more built-in functions for strings.

Here are a few functions which operate on strings.

```
(string-append "alpha" "bet") ⇒ "alphabet"
(string-length "perpetual") ⇒ 9
(string<? "alpha" "bet") ⇒ #true
```

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

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

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

Consider the use of symbols when a small, fixed number of labels are needed (e.g. colours) and comparing labels for equality is the only operation 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: `Symbol` and `String`.

# General equality testing

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

## General equality testing

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

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

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

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 all types of data we have encountered so far (except inexact numbers), and most types we will encounter in the future.

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

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 all types of data we have encountered so far (except inexact numbers), and most types we will encounter in the future.

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

# General equality testing

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

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 all types of data we have encountered so far (except inexact numbers), and most types we will encounter in the future.

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.

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

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 all types of data we have encountered so far (except inexact numbers), and most types we will encounter in the future.

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).

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

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

Example: Computing the absolute value of $x$.

$$|x| = \begin{cases} -x & \text{when } x < 0 \\ x & \text{when } x \geqslant 0 \end{cases}$$

Conditional expressions in Racket use the form `cond`.

Conditional expressions in Racket use the form `cond`.

Each argument of a `cond` is a clause consisting of a question/answer pair.

Conditional expressions in Racket use the form `cond`.

Each argument of a `cond` is a clause consisting of a question/answer pair.

The question is a Boolean expression. The answer is an expression that may possibly be evaluated to produce the value of the whole conditional expression.

Conditional expressions in Racket use the form `cond`.

Each argument of a `cond` is a clause consisting of a question/answer pair.

The question is a Boolean expression. The answer is an expression that may possibly be evaluated to produce the value of the whole conditional expression.

In Racket, we can compute $|x|$ with the expression:

```
(cond
  [(< x 0)    (- x)]
  [(>= x 0)       x])
```

Conditional expressions in Racket use the form `cond`.

Each argument of a `cond` is a clause consisting of a question/answer pair.

The question is a Boolean expression. The answer is an expression that may possibly be evaluated to produce the value of the whole conditional expression.

In Racket, we can compute $|x|$ with the expression:

```
(cond
  [(< x 0)    (- x)]
  [(>= x 0)       x])
```

Square brackets are used by convention around question-answer pairs, for readability.

The general form of a conditional expression is

```
(cond
  [question1 answer1]
  [question2 answer2]
  ...
  [questionk answerk])
```

where `questionk` could be `else`.

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 evaluates to `#true`, the corresponding answer is evaluated and becomes the value of the whole expression.

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 evaluates to `#true`, the corresponding answer is evaluated and becomes the value of the whole expression.

As soon as one question is found that evaluates to `#true`, no further questions are evaluated. Only one answer is ever evaluated (the one associated with the first question that evaluates to `#true`, or associated with the `else` if that is present and reached).

**Exercise**

What is the value of `(f 9)` when `f` is defined as:

```
(define (f x)
  (cond
    [(< x 5)"small"]
    [(< x 10) "medium"]
    [else "large"]))
```

**Exercise**

What is the value of `(f 9)` when `f` is defined as:

```
(define (f x)
  (cond
    [(< x 5)"small"]
    [(< x 10) "medium"]
    [else "large"]))
```

**Exercise Solution**

`"medium"`

Example:

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

```
(define (f x)
  (cond
    [(= x 0) 0]
    [ else   (* x (sin (/ 1 x)))]))
```

Notice the second question has been simplified.

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

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

Suppose our analysis identifies three relevant intervals:

- negative numbers,
- non-negative numbers less than 100,
- numbers greater than or equal to 100.


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

```
(cond
  [(< x 0)                    (f1 x)]
  [(and (>= x 0)(< x 100))    (f2 x)]
  [(>= x 100)                 (f3 x)])
```

## Simplifying conditional functions


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

```
(cond
  [(< x 0)                    (f1 x)]
  [(and (>= x 0)(< x 100))  (f2 x)]
  [(>= x 100)                 (f3 x)])
```

We can simplify the second and third tests.

```
(cond
  [(< x 0)    (f1 x)]
  [(< x 100)  (f2 x)]
  [else       (f3 x)])
```

These simplifications become second nature with practice.

## Tests for conditional expressions

Write at least one test for each possible answer in the expression. That test should be simple and direct, aimed at testing that answer.

Write at least one test for each possible answer in the expression. That test should be simple and direct, aimed at testing that answer.

Often tests are appropriate at boundary points as well.

Write at least one test for each possible answer in the expression. That test should be simple and direct, aimed at testing that answer.

Often tests are appropriate at boundary points as well.

DrRacket highlights unused code to point out clauses that are untested.

Write at least one test for each possible answer in the expression. That test should be simple and direct, aimed at testing that answer.

Often tests are appropriate at boundary points as well.

DrRacket highlights unused code to point out clauses that are untested.

Example:

```
(cond
  [(< x 0)   (f1 x)]
  [(< x 100) (f2 x)]
  [else      (f3 x)])
```

There are three intervals and two boundary points, so five tests are required (for instance, -10, 0, 10, 100, 150).

Testing `and`, `or` expressions is similar.

## Tests for conditional expressions

Testing `and`, `or` expressions is similar.

For `(and (not (zero? x)) (<= (/ y x) c))`, we need:

- one test case where $x$ is zero (first argument to `and` is `#false`)

Testing `and`, `or` expressions is similar.

For `(and (not (zero? x)) (<= (/ y x) c))`, we need:

- one test case where $x$ is zero (first argument to `and` is `#false`)
- one test case where $x$ is nonzero and $y/x > c$, (first argument is `#true` but second argument is `#false`)

Testing `and`, `or` expressions is similar.

For `(and (not (zero? x)) (<= (/ y x) c))`, we need:

- one test case where $x$ is zero (first argument to `and` is `#false`)
- one test case where $x$ is nonzero and $y/x > c$, (first argument is `#true` but second argument is `#false`)
- one test case where $x$ is nonzero and $y/x \leqslant c$. (both arguments are `#true`)

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.

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 the code, for example, to check specific answers in conditional expressions. These are known as **white-box tests**.

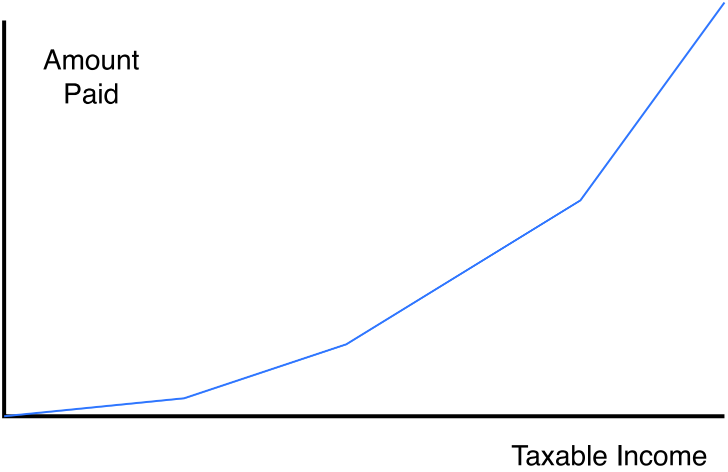
Some of your tests, including your examples, will have been defined before the body of the function was written. These are known as **black-box tests**, because they are not based on details of the code.

Other tests may depend on the code, for example, to check specific answers in conditional expressions. These are known as **white-box tests**.

Both types of tests are important.

Canada has a **progressive** tax system: the rate of tax increases with income. In 2014, the rates were:

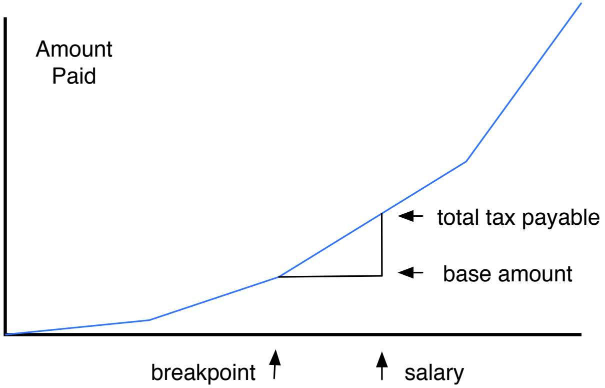
- no tax payable on negative income
- 15% from $0 up to $43,953
- 22% from $43,953 up to $87,907
- 26% from $87,907 up to $136,270
- 29% above $136,270


Amount
Paid
Taxable Income

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.



```
; breakpoints
(define bp1 43953)
(define bp2 87907)
(define bp3 136270)
; rates
(define rate1 0.15)
(define rate2 0.22)
(define rate3 0.26)
(define rate4 0.29)
```

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 rate1))
(define base2 (+ base1 (* (- bp2 bp1) rate2)))
(define base3 (+ base2 (* (- bp3 bp2) rate3)))

(define (tax-payable income)
  (cond
    [(<= income 0) 0]
    [(<= income bp1) (* income rate1)]
    [(<= income bp2) (+ base1 (* (- income bp1) rate2))]
    [(<= income bp3) (+ base2 (* (- income bp2) rate3))]
    [else (+ base3 (* (- income bp3) rate4))]))
```

## Helper functions

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

```
(define (tax-calc base low high rate)
  (+ base (* (- high low) rate)))
```

This can be used both in the definition of constants and in the main function.

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

```
(define (tax-calc base low high rate)
  (+ base (* (- high low) rate)))
```

This can be used both in the definition of constants and in the main function.

```
(define base1 (tax-calc 0 0 bp1 rate1))
(define base2 (tax-calc base1 bp1 bp2 rate2))
(define base3 (tax-calc base2 bp2 bp3 rate3))

(define (tax-payable income)
  (cond
    [(<= income 0) 0]
    [(<= income bp1) (tax-calc 0 0 income rate1)]
    [(<= income bp2) (tax-calc base1 bp1 income rate2)]
    [(<= income bp3) (tax-calc base2 bp2 income rate3)]
    [else (tax-calc base3 bp3 income rate4)]))
```

Helper functions generalize similar expressions.

Helper functions generalize similar expressions.

They help avoid long, unreadable function definitions.

Helper functions generalize similar expressions.

They help avoid long, unreadable function definitions.

They eliminate repetition.

Helper functions generalize similar expressions.

They help avoid long, unreadable function definitions.

They eliminate repetition.

Use judgement: don’t go to excess in writing helper functions, but sometimes very short definitions improve readability.

# Helper functions

Helper functions generalize similar expressions.

They help avoid long, unreadable function definitions.

They eliminate repetition.

Use judgement: don't go to excess in writing helper functions, but sometimes very short definitions improve readability.

Helper functions must also follow the design recipe.

Helper functions generalize similar expressions.

They help avoid long, unreadable function definitions.

They eliminate repetition.

Use judgement: don't go to excess in writing helper functions, but sometimes very short definitions improve readability.

Helper functions must also follow the design recipe.

Give all functions (including helpers) meaningful names, not "helper".

1. You should understand the reasons for each of the components of the design recipe and the particular way that they are expressed.

## Goals of this module

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

## Goals of this module

1. You should understand the reasons for each of the components of the design recipe and the particular way that they are expressed.
2. You should start to use the design recipe and appropriate coding style for all Racket programs you write.
3. You should understand Boolean data, and be able to perform and combine comparisons to test complex conditions on numbers.

1. You should understand the reasons for each of the components of the design recipe and the particular way that they are expressed.
2. You should start to use the design recipe and appropriate coding style for all Racket programs you write.
3. You should understand Boolean data, and be able to perform and combine comparisons to test complex conditions on numbers.
4. You should understand the syntax and use of a conditional expression.

1. You should understand the reasons for each of the components of the design recipe and the particular way that they are expressed.
2. You should start to use the design recipe and appropriate coding style for all Racket programs you write.
3. You should understand Boolean data, and be able to perform and combine comparisons to test complex conditions on numbers.
4. You should understand the syntax and use of a conditional expression.
5. You should understand how to write `check-expect` examples and tests, and use them in your assignment submissions.

1. You should understand the reasons for each of the components of the design recipe and the particular way that they are expressed.
2. You should start to use the design recipe and appropriate coding style for all Racket programs you write.
3. You should understand Boolean data, and be able to perform and combine comparisons to test complex conditions on numbers.
4. You should understand the syntax and use of a conditional expression.
5. You should understand how to write `check-expect` examples and tests, and use them in your assignment submissions.
6. You should be aware of other types of data (symbols and strings), which will be used in future lectures.

## Goals of this module

1. You should understand the reasons for each of the components of the design recipe and the particular way that they are expressed.
2. You should start to use the design recipe and appropriate coding style for all Racket programs you write.
3. You should understand Boolean data, and be able to perform and combine comparisons to test complex conditions on numbers.
4. You should understand the syntax and use of a conditional expression.
5. You should understand how to write `check-expect` examples and tests, and use them in your assignment submissions.
6. You should be aware of other types of data (symbols and strings), which will be used in future lectures.
7. You should look for opportunities to use helper functions to structure your programs, and gradually learn when and where they are appropriate.