The design recipe

Readings:

- HtDP, sections 1-5

  (ordering of topics is different in lectures, different examples will be used)

- Survival and Style Guides

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.

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 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 (header and body) of the function.

Tests: A representative set of function arguments and expected function values.

Order of Execution

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

- Write a draft of the Purpose
- Write Examples (by hand, then code)
- Write Definition Header & Contract
- Finalize the purpose with parameter names
- Write Definition Body
- Write Tests

Using the design recipe

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

Purpose:

```
;; (sum-of-squares p1 p2) produces the sum of
;;     the squares of p1 and p2
```

Contract:

```
;; sum-of-squares: Num Num → Num
```

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

```
;; Examples:
(check-expect (sum-of-squares 3 4) 25)
(check-expect (sum-of-squares 0 2.5) 6.25)
```

Header & Body:

```
(define (sum-of-squares p1 p2)
  (+ (* p1 p1) (* p2 p2)))

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

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 (including 0)
- Any: any Racket value
- We will see more types soon.

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 **by hand**.

The teaching languages offer a convenient testing method.

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

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.

```
;; (sum-of-squares p1 p2) produces the sum of
;;     the squares of p1 and p2
;; sum-of-squares: Num Num → Num
;; Examples:
(check-expect (sum-of-squares 3 4) 25)
(check-expect (sum-of-squares 0 2.5) 6.25)

(define (sum-of-squares p1 p2)
   (+ (* p1 p1) (* p2 p2)))

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

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.

```
;; (my-function a b c) ...
;; my-function: Num Num Num → Num
;; requires: 0 < a < b
;;                c must be non-zero
```

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 before completing your assignments.

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).

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 #f; these will sometimes show up in basic tests and correctness tests.

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:

;; = : Num Num $\rightarrow$ Bool

Note that Boolean is abbreviated in contracts.

Complex relationships

You may have 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 "$3 \leq x < 7$", which is the same as "$x \in [3, 7)$", can be computationally tested by evaluating (and (<= 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.

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

Examples:

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

These will never divide by zero.

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-drink? age)
  (>= age 19))
```

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’: ’blue

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

It 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 CS135 that is applied only to symbols.

Other types of data

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).
- More efficient to compare two symbols than two strings
- 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
```

The textbook does not use strings; it uses symbols.

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

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).

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.

Do not overuse equal?, however.

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

Example: taking the absolute value of x.

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

- Conditional expressions use the special form cond.
- Each argument is a question/answer pair.
- The question is a Boolean expression.
- The answer is a possible value of the conditional expression.

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

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

- square brackets used by convention, for readability
- square brackets and parentheses are equivalent in the teaching languages (must be nested properly)
- 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 order; as soon as one evaluates to true, the corresponding answer is evaluated and becomes the value of the whole expression.

- 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.

  (the one associated with the first question that evaluates to true, or associated with the else if that is present and reached)

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

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

Simplifying conditional functions

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 intervals:

- students who fail CS 135 must take CS 115 (this isn't true).
- students who pass but get less than 60% go into CS 116.
- students who pass and get at least 60% go into CS 136.

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

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

We can simplify the second and third tests.

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

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.
- Often tests are appropriate at boundary points as well.
- DrRacket highlights unused code.

For the example above:

```
(cond [(< grade 50) 'cs115]
      [(< grade 60) 'cs116]
      [else 'cs136]))
```

there are three intervals and two boundary points, so five tests are required (for instance, 40, 50, 55, 60, 70).

Testing and 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 $y/x \leq c$.

  (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.

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.

Writing Boolean tests

The textbook writes tests in this fashion:

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

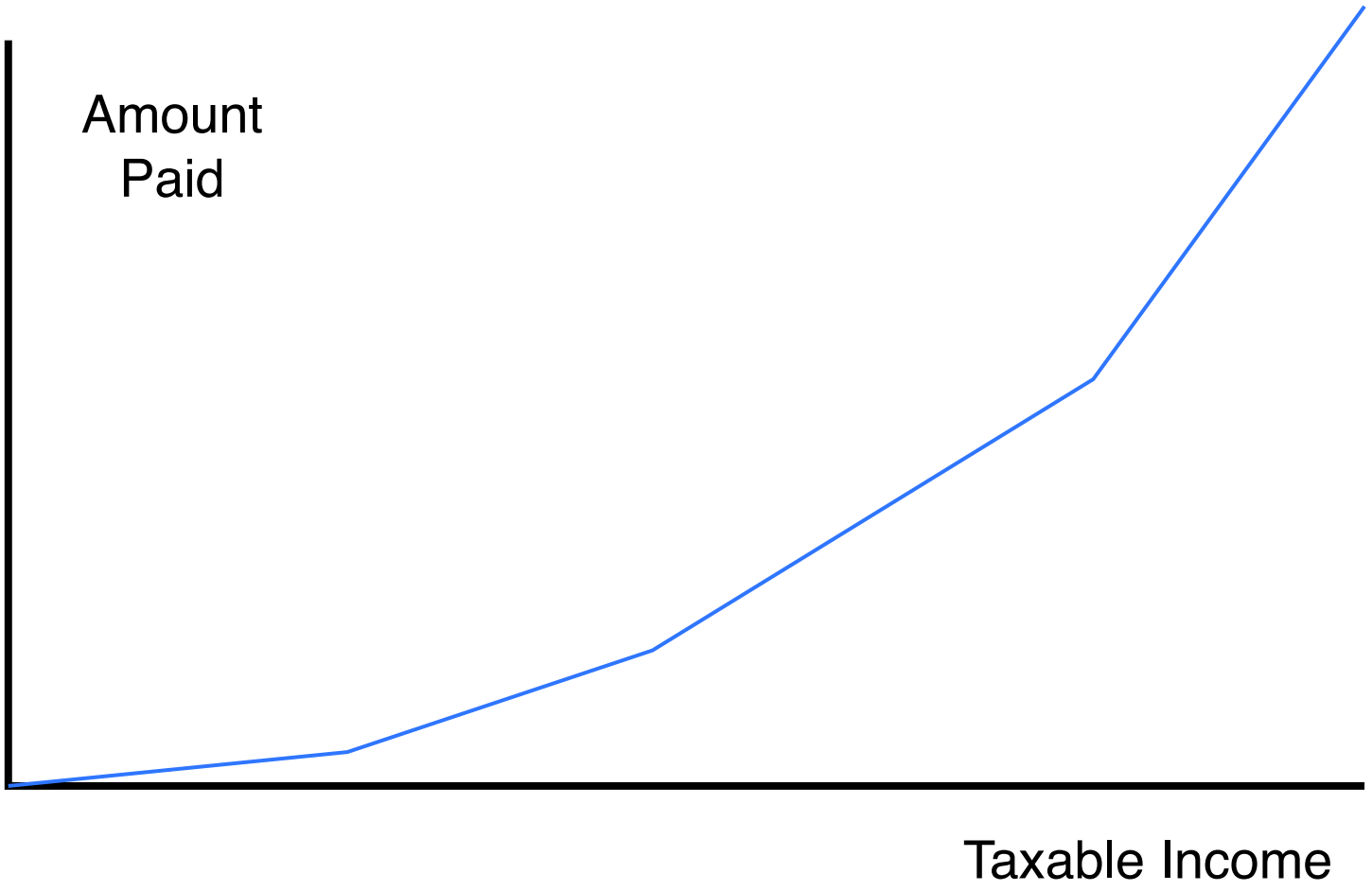
which works outside the teaching languages.

check-expect was added to the teaching languages after the textbook was written. You should use it for all tests.

Example: computing taxes

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

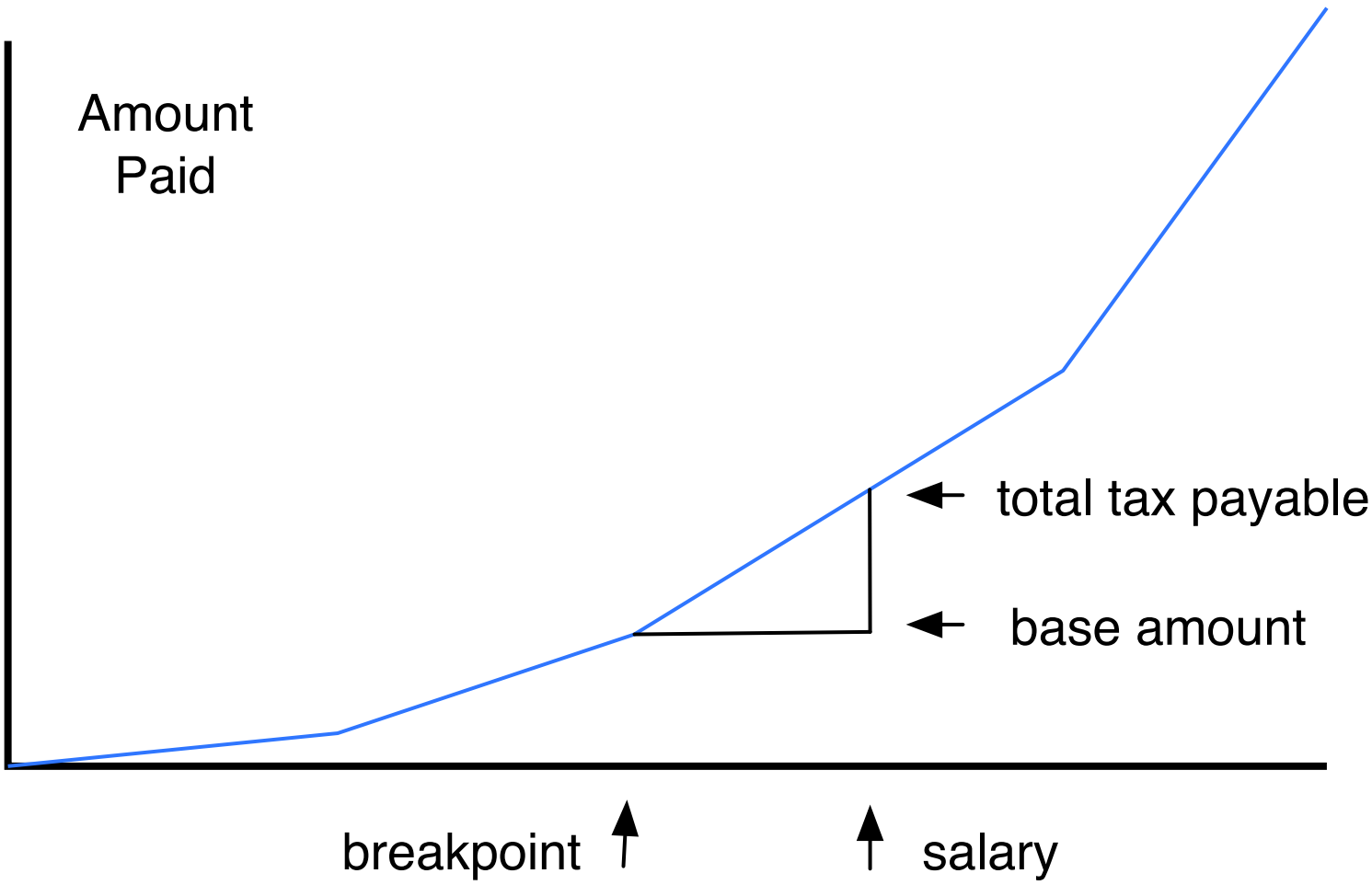
- no tax payable on negative income
- 15% on the amount from $0 to $43,953
- 22% on the amount from $43,953 to $87,907
- 26% on the amount from $87,907 to $136,270
- 29% on the amount from $136,270 and up


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.


Amount
Paid
total tax payable
base amount
breakpoint
salary

```
;; 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.

```
(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)]))
```

Good example: movie theatre (section 3.1 of HtDP).

Helper functions generalize similar expressions.

They avoid long, unreadable function definitions.

Use judgement: don't go overboard, but sometimes very short definitions improve readability.

Helper functions must also follow the design recipe

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

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.

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.