CS 135 Winter 2018
Tutorial 01: Translations, Constants, and the Design Recipe
Announcements

• The times and locations of office hours are posted on the “Office & Consulting Hours” page of the course website. Please email us at cs135@uwaterloo.ca to set up an appointment outside of these hours.

• Assignment 1 is due on **Tuesday, January 16, at 9:00 pm**. Submit early and often to MarkUs! Check your basic test results after each submission. We will not be lenient about any late submissions.

• Make sure you complete Assignment 0 before the Assignment 1 deadline, if you haven’t done so already!

• Ensure that your clicker marks posted on the “View Marks” page of the course website are accurate.
Announcements

• MarkUs Basic tests:
  – Are set up for every assignment.
  – Do not thoroughly test your code.
  – Ensure we can run more thorough tests on your code after the due date.
  – The results are automatically emailed to your UWaterloo email. You can also check the results on MarkUs after each submission.
  – Are not related to the tests that you will write in your solutions.
Goals of this tutorial

You should be able to...

- Give a **direct translation** of mathematical expressions and functions in Racket.
- Understand when and how to use **constants**.
- Write the full **design recipe** for simple arithmetic functions.
How to find the help pages

- **Do not** use Google search. It will land you at the wrong language level, which is typically the full Racket help page.

- Open DrRacket: Help menu > Help Desk or Racket Documentation (this opens a browser window) > Teaching > How to Design Programs Languages > Select the appropriate language level (e.g. Beginning Student).

- Note the categorized list of functions on the left side bar.

- If you must use Google, then add the teaching language name to your query, e.g. “Racket beginning student”.
Clicker Question: Errors in Racket

Which of the following is an error-free Racket expression in “Beginning Student”?

A \((8 + 6 / 3)\)
B \((\ast (\plus 6 \minus 12 18) \minus 24)\)
C \((/ 5 (\minus 4 (\sqrt 16)))\)
D \((\ast (\plus 5 10) (15))\)
E They all have some kind of error.
Direct Translations: Tips

When given a mathematical function or expression to translate:

- Do not swap the order of parameters, or change their names.
  - For example, $2x + 2y$ should not be translated as $(+ (* 2 y) (* 2 x))$ or $(+ (* 2 a) (* 2 b))$.

- A mathematically equivalent expression is not necessarily a direct translation.
  - For example, $\frac{x}{y}$ should be translated as (/ x y), but not $(\ast x (\text{expt } y - 1))$.
  - Similarly, $2x + 2y + x$ should be translated as $(+ (* 2 x) (* 2 y) x)$, but not $(+ (3 x) (2 y))$. 
Clicker Question: Direct Translation

What is the correct Racket translation of the following mathematical expression?

\[ 15 + \frac{(6+3)^2}{10} - 18 \cdot 17 \]

A \[ (+ (- 15 (sqr (/ (+ 6 3) 10))) (* 18 17)) \]
B \[ (- (+ 15 (sqr (/ (+ 6 3) 10))) (* 18 17)) \]
C \[ (+ (- 15 (/ (sqr (+ 6 3)) 10)) (* 18 17)) \]
D \[ (- (+ 15 (/ (sqr (+ 6 3)) 10)) (* 18 17)) \]
E None of the above
Group Problem: Direct Translation

Translate the following mathematical function into Racket:

The area of a regular polygon, given the length of one side, $s$, and the number of sides, $n$, can be computed with the following formula:

$$\text{polygon-area}(s, n) = \frac{1}{4} \cdot n \cdot s^2 \cdot \frac{1}{\tan\left(\frac{\pi}{n}\right)}$$

(Hint: $\tan$ is a built-in Racket function that may be useful.)

Ensure that you are giving a direct translation of the function above.
Review: Advantages of constants

- Can give meaningful names to useful values (e.g. `interest-rate`, `passing-grade`, and `starting-salary`).
- Reduces typing and errors when such values need to be changed.
- Makes programs easier to understand.
- Constants can be used in any expression, including the body of function definitions.
- Sometimes called variables, but their values cannot be changed (until CS 136).
Review: The five Design Recipe components

**Purpose:** Describes what the function produces. You should meaningfully include parameter names in your purpose statement.

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

**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.

**Examples:** Illustrate the use of the function.

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

**Tests:** A thorough set of function arguments and expected function values.
Example: The Design Recipe components

Most countries in the world have sales tax on purchased goods and services. We will write a Racket function `calc-tax`, which consumes an amount spent on a purchase, called `purchase-cost`, and a tax percentage, called `tax-rate`, and produces the amount of tax that should be paid for that purchase. The consumed tax percentage will be an integer between 0 and 100, inclusive.

We will write all parts of the Design Recipe for the function `calc-tax`. 
Purpose Statements: Do’s

When writing a purpose statement, ensure that you do the following:

- Include the **function header**, which contains the name of the function and all its parameters.

- Have a brief description of what the function does after the function header.

- Reference the parameter names (as they appear **exactly** in the function header) in the description to explain the significance of each parameter.

- If the purpose statement exceeds one line, indent all lines of the purpose statement after the first line by 2 or 3 spaces.
Purpose Statements: Do’s

An example of a good purpose statement for `calc-tax` is:

```
;; (calc-tax purchase-cost tax-rate) produces the amount of tax that should
;; be paid on purchase-cost, based on tax-rate
```
Purpose Statements: Don’ts

When writing a purpose statement, avoid the following:

- Labeling the purpose statement with a header (e.g. Purpose: ).

- Missing/incorrect function headers. Make sure the function header in the purpose statement is identical to the function header in the function definition.

- Missing parameter references in the purpose statement. Parameters used in the function should always be referenced directly in the purpose statement.

- Using the terms input and output. In this course, we say that a function consumes arguments and produces a value.
Purpose Statements: Don’ts

When writing a purpose statement, avoid the following:

- Writing purpose statements of the form “...consumes a number n, an integer x, etc., and produces...”. This does not adequately explain the significance of the parameters in the function, and simply repeats information that is communicated in the contract.

- Including the data types of the function’s parameters (unless the purpose statement would flow better), or any requirements for specific parameters in the purpose statement. That is what the contract is for.

- Explaining how the code works. A good indication of this is when the purpose statement exceeds 2 or 3 lines.
Purpose Statements: Don’ts

An example of a bad purpose statement for `calc-tax` is:

```plaintext
;; Purpose: (calc-tax) consumes a number, purchase-cost, and a positive integer, tax-rate, and produces the amount of tax that should be paid by multiplying the amount that was spent by the given tax percentage, and then dividing the result by 100. The amount of money spent should be greater than or equal to zero, and the tax percentage is also less than or equal to 100.
```
Contracts: Do’s

When writing a contract, ensure that you do the following:

- Begin the contract with **only** the name of the function, followed by a colon (e.g. `fn-name: `). Do **not** include any parameter names.

- List out the data types of the parameters in the same order as they are written in the function header, with exactly one space in between the data types of each parameter.

- Include an arrow ( → ) following the data types of the parameters.

- After the arrow, indicate the data type of the value that the function produces.
Contracts: Do’s

- The data types should match those listed in section 4 of the Style Guide. Data types should **always** start with a capital letter.

- If your function has any additional restrictions on the parameters **that are not already implicit in their type**, include a “requires:” section directly underneath the contract.

An example of a good contract for `calc-tax` is:

```;
;; calc-tax: Num Nat → Num
;; requires: purchase-cost ≧ 0
;;        tax-rate ≦ 100
```
Contracts: Don’ts

When writing a contract, **avoid** the following:

- Labeling the contract with a header (e.g. `Contract: `).

- Putting unneeded requirements that are already communicated in the contract types (e.g. `nat-param \geq 0`).

- Indicating any restrictions on the value that the function **produces**.

- Adding restrictions that can be avoided by choosing a more appropriate and specific data type. For example, writing

  “my-fn: Int → Num
  requires: int-param \geq 0” is the same as writing Nat in the contract.
Contracts: Don’ts

An example of a bad contract for `calc-tax` is:

```plaintext
;; Contract: (calc-tax purchase-cost tax-percentage) Number int → num
;; requires: purchase-cost ≥ 0
;;          0 ≤ tax-percentage ≤ 100
;;          tax-percentage is an integer
;;          num produced from calc-tax ≥ 0 and ≤ purchase-cost
```
Examples

- Unlike the purpose and contract, the Examples section should start with a header (Examples: ).

- For each function, provide 1-3 examples, illustrating typical uses of the function.

- Write your examples using check-expect. The first argument given to check-expect is a function application, and the second argument given is the expected result of that function application.

;; Examples:

(check-expect (calc-tax 50 10) 5)
(check-expect (calc-tax 100 13) 13)
CQ: For the function `expt`, which of the following has the best purpose, contract, and example?

A
;; (expt arg1 arg2) produces the first
;; number to the the power of the
;; second number
;; expt: num num → num
;; Example:
;(check-expect (expt 3 2) 9)

B
;; (expt arg1 arg2) consumes two
;; numbers, arg1 and arg2, and
;; produces the result of the first
;; number to the power of the second
;; number
;; (expt arg1 arg2): Num Num → Num
;; Example:
;(check-expect 9 (expt 3 2))

C
;; Contract: expt: Number Num → Num
;; Purpose: (expt x y) produces arg1
;; to the power of arg2
;; Example:
;(check-expect (expt 4 2) 16)

D
;; (expt arg1 arg2) produces arg1 to
;; the power of arg2
;; expt: Num Num → Num
;; requires: if arg1 is not zero, then the
;; produced number is also
;; not zero
;(check-expect 8 (expt 2 3))

E
;; (expt arg1 arg2) produces arg1 to
;; the power of arg2
;; expt: Num Num → Num
;; Example:
;(check-expect (expt 3 2) 9)

Note that `expt` is a built-in Racket function.
Definition

- Do not label the function definition (e.g. Definition: ).

- The function definition consists of the function header and the function body.

- Ensure that the function header and the function body are not on the same line.

- There should be one blank line before and after the function definition.

(define (calc-tax purchase-amt tax-rate) ; function header
    (/ (\ast purchase-amt tax-rate) 100)) ; function body
Tests

- The Tests section should start with a header (Tests: ).
- As with examples, write your tests using check-expect.
- Do not repeat your examples here - they also count as tests.
- You will learn in the next few lectures how to design a set of tests for each function you write.

;; Tests:
(check-expect (calc-tax 0 50) 0)
(check-expect (calc-tax 13.27 8) 1.0616)
(check-expect (calc-tax 180.17 0) 0)
Group Problem: total-prize-money

On a game show, there are three contestants who are asked 50 questions each. Prize money is awarded according to the following rules:

- The contestant that answers the most questions correctly will receive $15 for every correctly answered question from the game show host.
- The contestant that answers the second-most number of questions correctly will receive $7.50 per correct answer.
- The contestant that answers the least number of questions correctly will receive $5 per correct answer.
Group Problem: total-prize-money

Write a function `total-prize-money`, that consumes three parameters: the number of questions that the first, second, and third contestants answer correctly. The function produces the total amount of money that the game show host must pay to the contestants, according to the rules on the previous slide. For simplicity, you may assume that all players will answer a different number of questions correctly. Include the full design recipe for this function.

```scheme
(total-prize-money 17 28 32) ⇒ 775
(total-prize-money 43 12 31) ⇒ 937.5
```