ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ

Лекції 3-4

ЛІСПОВИДНА МОВА RACKET

Racket - мультіпарадігменний мову загального призначення, що входить в Lisp-сімейство. Підкреслю, що Lisp - це не назва конкретного мови, а позначення групи мов, що володіють певними схожими властивостями. Racket, як і більшість інших ліспоподобних мов, не є функціональним мовою програмування. На ньому можна писати як функціонально, так і імперативно.

Lisp-мов багато, а перший з'явився в 1958 році. Його автором був Джон Маккарті - автор терміну «штучний інтелект», один з основоположників функціонального програмування, лауреат премії Тьюринга за величезний внесок в область досліджень штучного інтелекту.

Крім Racket, в сучасній розробці щодо популярні - Clojure, ClojureScript, Common Lisp, різні варіанти Scheme (читається "ським"). Цікавий факт: редактор Emacs написаний на власному діалекті, який називається Emacs Lisp. І все програмісти, які використовують emacs, в якійсь мірі є Lisp-програмістами.

Все що далі буде говоритися про Racket, майже завжди справедливо і для інших Lisp-мов. Та й сам Racket дуже схожий на інші діалекти Scheme (так, Racket - теж діалект Scheme). Цим фактом можна користуватися при роботі в https://repl.it, де немає Racket, але є Scheme.

За традицією почнемо знайомство з мовою з написання програми "Hello, World!". Ця програма буде виводити на екран текст Hello, World !. Для виведення на екран в Racket використовується функція `displayln`:

```
#lang racket

(displayln "Hello, World!")
```

У середовищі програмістів на Lisp-мовами традиційно говорять "процедура", а не "функція". Історично, процедура це блок коду, який не має повернення і як правило потрібен виключно для виконання побічних ефектів. А під функцією розуміють функцію в її математичному значенні. Функція залежить від своїх аргументів і обчислює результат, який потім повертається назовні. У деяких мовах, таких як Pascal, ці поняття рознесені на рівні синтаксису і семантики. У більшості ж мов функціями називають в тому числі процедури. У документації Racket зустрічається і те й інше. Для простоти ми будемо завжди говорити про функції.

В наведеному вище прикладі є рядок #lang racket, це так звана "Прагма". Прагма lang говорить компілятору, що даний файл, це модуль (про модулі ми поговоримо пізніше), написаний на мові racket. Справа в тому, що Racket, це не тільки мова сама по собі, але ще і платформа для створення інших мов! Більш того, в одному проекті можна використовувати відразу кілька створених в Racket мов. Саме тому нам потрібно повідомляти компілятору, якою ж мовою ми писали код модуля.

Один із прикладів "іншої мови на базі Racket" - Scribble, мова для розмітки документації, книг, статей. Програми на Scribble генерують HTML і PDF (а також і інші формати).

Racket вимагає вказівки Прагма #lang в кожному файлі з вихідним кодом (є виключення але це зараз не важливо), проте в нашому середовищі она присутній не завжди (з технічних причин). Не забудьте про цю різницю, коли вирішите запускати Racket програми у себе на комп'ютері.

Завдання

Наберіть в редакторі код з завдання символ в символ і натисніть «Перевірити».

```
; я комментарий
(displayln "Hello, World!")
```

Поради

- repl.it - тут ви можете експериментувати з кодом на Racket.
- Використовуйте готовий бойлерплейт для проходження SICP на Racket

Код як дані

Перше питання, яке виникає при читанні коду Racket-програми - чому така дивна запис виклику функції? Замість звичного: displayln ("eat me"), ми бачимо: (displayln "eat me").

Причини такого запису лежать в ідеї, яка стоїть за усіма Lisp-мовами. Назва "Lisp" розшифровується як LISt Processor (обробник списків). Однозв'язний список - основна структура даних в цих мовах. Більш того, будь-яка програма на Lisp-мові - сама по собі список! Подивіться ще раз на цю програму:

```
(displayln "eat me")
```

З одного боку це виклик функції displayln зі строковим аргументом "eat me". З іншого, це список з двох елементів: displayln і "eat me". Розглянемо кілька прикладів. Не намагайтеся їх зрозуміти як код, ми ще не готові до цього. Дивіться на них як на звичайні списки.

```
(+ 1 2); складання або список з трьох елементів

(Define count 0); оголошення змінної або список з трьох елементів

(+ 100 3 8 9); складання або список з 5 елементів
```

У цьому моменті проявляється одна з ключових особливостей будь-якого Lisp-мови: код на Lisp одночасно є даними Lisp-мови (кажуть "код як дані"). Це властивість називається гомоіконічностью і є візитною карткою даного сімейства мов.

Можливо вас цікавить питання, навіщо це потрібно? Гомоіконічность дає можливість писати макроси, що працюють з вихідним кодом як зі списком. Механізм макросів в Lisp-мовами - одна з найпотужніших речей в програмуванні взагалі.

Завдання

Напишіть програму, яка виводить на екран фразу “winter is coming!”

Визначення

- Гомоіконічность - властивість деяких мов програмування, в яких текст програми одночасно може розглядатися як структура даних цього ж мови.

Списки як дерево

Списки можуть бути вкладеними. У такій формі замість конкретного значення підставляється новий список ():

```
(- 1 | 8); віднімання або список з трьох елементів

(+ 1 (- 3 2)); складання або список з трьох елементів, в якому третій елемент список з трьох елементів

(- (* 3 3) (- 2 3) (+ 5 1)); віднімання або список з 4 елементів
```

Питання для самоконтролю. Скільки елементів в списку ((3) 8 (7 9))?

Список - рекурсивна структура даних. Будь-який елемент може бути списком і містити в собі елементи-списки. Нижче приклад повноцінної Racket-програми:

```
(define (square x) (expt x 2))

(define (squarer xs)
  (if (empty? xs)
      empty
      (cons (square (first xs)) (squarer (rest xs)))))

(squarer '(1 2 3 4 5))
```

Код виглядає незвично і не дуже зрозуміло, але спробуйте поміняти кут зору. Подивіться на цей код як на структуру даних. Простежте за тим, як списки вкладаються один в одного і форматуються.

Фактично виходить, що вихідний код на Racket, це деревоподібна структура.

Завдання

Напишіть програму, що виводить на екран різницю чисел 128 і 37

Форми

Про Lisp-подібні мови говорять, що у цих мов немає синтаксису. Синтаксис у них звичайно є, але максимально примітивний, фактично складається зі списків і значень. Крім того, в Ліспі відсутні ключові слова і відповідні їм конструкції. У звичайних мовах існує безліч керуючих конструкцій, таких як умови, цикли, повернення, привласнення змінних і багато іншого. У ліспоподібних мовах таких конструкцій немає (це не означає, що на Racket не можна реалізувати цикл або написати умова - можна!).

Яким же чином Racket розуміє, з чим зараз він працює і що потрібно робити? Вся справа в формах. Будь-яка коректна програма на Lisp називається формою. наприклад:

```
; форма
(displayln "i am from form")
; форма
(+ 1 2)
; и це форми
8
"hello"

; а це не форма, так як такий код завершиться з помилкою
(1 2 3)
```

Форм всього дві - нормальна і складова. Нормальною відповідають все значення (і визначення, з якими ми познайомимося пізніше), так як вони обчислюються самі в себе, наприклад, число 8 або рядок "hello". Складова форма, це список, який потрібно обробити тим чи іншим способом (обчислити).

Коли код представлений як список, з'являється простір для інтерпретації. Візьмемо складання двох чисел, наприклад, 3 і 2. У вигляді списку таке додавання можна уявити трьома різними способами:

- `(3 + 2)`
- `(3 2 +)`
- `(+ 3 2)`

У Lisp-мовах використовується префиксная нотація, тобто перший елемент форми визначає поведінку (семантику). Такий спосіб має ряд переваг. Наприклад він дозволяє природним чином виконувати дію з будь-яким набором елементів:

```
; 3 + 2 + 8 + 3 + 9
(+ 3 2 8 3 9) ; 25

; 3 - 2 - 8 - 3 - 9
(- 3 2 8 3 9) ; -19
```

Інша перевага - простота реалізації динамічної диспетчеризації в порівнянні з іншими мовами. Цьому сприяють і макроси.

Завдання

Виведіть на екран значення виразу `10 - 100 - 12 - 18`

Порядок обчислень

Спробуємо перетворити вираз 5 - 3 + 1 в програму на Racket. З точки зору арифметики порядок обчислення елементів цього складеного вираження строго визначений. Спочатку обчислюється 5 - 3, потім до одержали результату додається одиниця.

Почнемо з того що обчислюється першим: 5 - 3 перетворюється в (- 5 3). Потім складемо вийшов результат з одиницею: (+ (- 5 3) 1). Так як складання, це Комутативність (пам'ятаєте, "від зміни місць доданків сума не змінюється"?), Те ж саме можна записати і в іншому порядку: (+ 1 (- 5 3)). Незмінним залишається те, що на початку кожного списку знаходиться операція.

Зауважте, вирази, що обчислюються першими, знаходяться в глибині дерева. Така поведінка властива більшості звичайних мов. Спочатку обчислюються аргументи функцій, потім викликається сама функція.

Спробуємо інший варіант: 5 - (3 + 1). У цьому виразі дужки встановлюють інший пріоритет. Це означає, що спочатку обчислити сума одиниці і трійки. Так і запишемо (+3 1) (або так (+ 1 3)). Тепер, візьмемо п'ятірку і віднімемо з неї вийшов результат: (- 5 (+1 3)).

У такі моменти проявляється ще одна відмінна риса Lisp-мов. Деревоподібна структура програми сама визначає послідовність обчислення піддерев. Відпадає необхідність використовувати додаткові дужки.

Ще один приклад: 5 + 7 + (8 - 3) - (8 * 5). Діємо за звичною схемою:

- `(* 5 8)`
- `(- 8 3)`
- `(+ 5 7 (- 8 3))`
- `(- (+ 5 7 (- 8 3)) (* 5 8))`

У деяких ситуаціях порядок обчислення елементів списку не відповідає порядку їх слідування. Таке відбувається при використанні спеціальних форм і макросів. Про це поговоримо пізніше.

Завдання

Виведіть на екран значення виразу: `100 - 34 - 22 - (5 + 3 - 10)`

Дужки

Дужки настільки лякають новачків, що весь інтернет завалений питаннями "чому в Ліспі так багато дужок?". Такі питання виникають не просто так: при звичайному способі редагування тексту Lisp-програми практично не піддаються модифікації. Забута дужка може стати причиною довгого налагодження. Подивіться на цей код:

```
(define (GET/hash #:rconn [rconn (current-redis-connection)] key
                  #:map-key [fkey identity]
                  #:map-val [fval identity])
  (let loop ([lst (HGETALL #:rconn rconn key)] [h (hash)])
    (if (null? lst) h
        (loop (cddr lst) (hash-set h (fkey (car lst)) (fval (cadr
lst)))))))
```

В самому кінці дуже багато дужок. Уявіть, що буде, якщо доведеться обернути якусь частину коду в новий список або видалити непотрібний список?

Схожі проблеми виникають при редагуванні HTML файлів, коли потрібно видалити як відкриває, так і закриває тег (або навпаки - додати).

Очевидно, що звичайний спосіб роботи в редакторі не дуже підходить для модифікації Lisp-програм. Тому в цьому світі прийняті інші підходи, про які початківці Lisp-програмісти дізнаються випадково.

Секрет приємної роботи з Lisp-кодом полягає в зміні точки зору на цей код. У той час як у звичайних мовах ми модифікуємо текст, в Lisp ми оперуємо деревом. Для зручної роботи з цим деревом нам знадобляться операції, які допомагають легко вставляти видаляти вузли, об'єднувати їх і роз'єднувати. А ще непогано було б ніколи не порушувати "баланс дужок".

Такий спосіб роботи з кодом існує і називається "структурний редагування". Історично склалося так, що розширення для редакторів прийнято називати "Paredit". Спробуйте загугли: "<ім'я редактора> paredit lisp". Скажімо, для продуктів компанії JetBrains розроблено спеціальне розширення Cursive. У документації цього розширення наочно показані можливості Paredit. Обов'язково подивіться ці гифки, вони допоможуть зрозуміти принципи управління Lisp кодом. У Paredit є альтернатива Parinfer.

При правильному підході, в якийсь момент ви раптом виявите, що структурний редагування ефективніше звичайного. Дужки перестануть бути проблемою, а при поверненні в звичайні мови ви почнете відчувати незручності.

Інший важливий аспект - правильне форматування. Довгі операції прийнято розбивати так, що операнди виявляються один під одним:

```
(+ 234
   88
   123423)
```

Більш складний приклад:

```
(- 100
   (+ 4 100)
   (- 1000
      50))
```

Такий запис теж вимагає звикання, але натомість ви зможете з легкістю орієнтуватися в акуратно оформленому коді.

Завдання

Виведіть на екран значення виразу: 4 + 2 - 3 * 5 - 8 / 7. Виконайте форматування коду, так щоб він легше сприймався (код можна розбити по-різному).

Об’ява символів

Racket - не функціональна мова програмування. У ньому є справжні змінні, які можна змінювати. Змінні створюються за допомогою конструкції define і називаються оголошеннями.

```
(Define id expr)

; id - ідентифікатор

; expr - вираз
```

наприклад:

```
; define створює "оголошення".

(Define lang "racket")

(Displayln lang); => "Racket"
```

Значенням оголошення може бути як нормальна форма (значення) так і складова:

```
(define result (+ 7 (- 4 6)))
(displayln result) ; 5
```

define пов'язує ім'я (ідентифікатор) і значення наступного за ним вирази.

Імена оголошень, що складаються з декількох слів, з'єднують за допомогою дефіса. У Lisp мовах повсюдно прийнятий так званий "kebab-case".

```
(define dangerous-year 1984)
(displayln dangerous-year) ; => 1984
```

Для зміни значення оголошення використовується функція `set!`:

```
(set! lang "scheme")
(displayln lang) ; => "scheme"
```

У загальному випадку використовувати set! не рекомендується. Racket відмінно підтримує функціональну парадигму і всіляко її заохочує. Код зі змінними практично завжди легко замінюється на код з константами.

Завдання

Створіть оголошення, що позначає "кількість учасників" (ім'я спорудити самі), надайте йому значення 100 і роздрукуйте на екран.

Створення і виклик функцій

Функції в Racket мають наступні властивості:

• У функцій немає імен. У багатьох мовах такі функції також існують і називаються анонімними функціями або лямбда-функціями.

• Функції є об'єктами першого роду. Їх можна привласнювати змінним, передавати в інші функції і повертати з функцій.

приклади:

```
; визначення функції обчислює суму двох чисел

(lambda (x y) (+ x y))
```

В наведеному вище прикладі визначається функція з двома аргументами. Визначення функції починається зі слова lambda. Другим елементом у формі визначення функції йде список аргументів. Третій і наступні елементи - тіло функції. Тобто тіло може складатися з декількох форм (як мінімум - з однієї):

```
(lambda ()
        (displayln "one")
        (displayln "two"))
```

Зверніть увагу на відсутність інструкції return. На відміну від більшості інших мов, в Lisp-мовами "інструкцій" практично немає. Все є вираз. А вираження завжди повертають результат. Якщо добре подумати, то така поведінка випливає з самої структури Lisp програми. Фактично ми маємо дерево, яке має обчислюватися в якесь значення, значить на кожному рівні повинен створюватися повернення, що піднімається вище по дереву і так до самого кореня. Повертається завжди останнім обчислене вираз.

Пара прикладів для закріплення:

```
; печать на екран
(lambda () (displayln "hello!"))
; квадрат числа
(lambda (n) (* n n))

; середнє між двома числами
(lambda (num1 num2) (/ (+ num1 num2) 2))
```

Визначення функції, саме по собі мало корисно, особливо якщо ми захочемо використовувати її кілька разів. Для повторного використання потрібно створити оголошення, в яке запишеться функція. Таке можливо завдяки тому, що форма визначення функції, цей вислів, яке повертає саму функцію.

```
(define square (lambda (n) (* n n)))
```

Тепер спробуємо викликати:

```
(square 7) ; 49
(square 5) ; 25
```

Завдання

Створіть функцію з ім'ям cube, яка обчислює куб переданого числа

```
(cube 3) ; 27
```

Визначення

- Об'єкт першого роду - сутність в мові, яка розглядається як дані. Це означає, що її можна записувати в змінну, передавати в функції і повертати з функцій.

Виклик функції без define

Згадаємо функцію `square` :

```
(define square (lambda (n) (* n n)))
(square 5) ; 25
```

Спробуйте відповісти на питання, чи можна викликати функцію зразу після оголошення без використання define?

Звичайно можна, як і у всіх інших мовах з підтримкою анонімних функцій. Для цього досить, у формі виклику, підставити замість імені саму функцію:

```
((lambda (n) (* n n)) 5)
```

Незвичайність цієї структури в тому, що тут перший елемент не ідентифікатор, а вираз, що повертає функцію (а визначення анонімної функції повертає саму функцію). Через це виникає подвійна дужка. Найпростіше це зрозуміти, якщо завжди сприймати дужки як виклик. Надалі при роботі зі списками, як з даними, ми побачимо, що це не завжди так, але на даному етапі дане спрощення корисно.

У формі виклику одразу після функції перераховуються параметри. В даному прикладі параметр тільки один, це 5. Так буде виглядати приклад виклику функції з двома аргументами:

```
((lambda (x y) (+ x y))
  8 7) ; 15
```

Виклик функції - звичайний вираз, це означає що його можна використовувати в усіх місцях, де можлива поява виразу. У Lisp-мовах це майже будь-які частини програми:

```
(displayln ((lambda (x y) (+ x y))
            8 7)) ; => 15
```

Завдання

1. Визначте (без створення змінної) і викличте функцію, яка знаходить середнє арифметичне між двома числами. Як чисел підставте 2 і 4.
2. Запишіть результат в змінну.
3. Виведіть змінну на екран.

Скорочений синтаксис створення функції

Створення функцій - настільки часта операція, що в Racket була додана скорочена версія визначення (з одночасним оголошенням) функції за допомогою define. Візьмемо для прикладу визначення функції зведення в квадрат:

```
(define square (lambda (n) (* n n)))
```

А тепер подивимося скорочену версію цього ж визначення:

```
(define (square n) (* n n))
(square 3) ; 9
```

Перше що кидається в очі - відсутність слова lambda. Замість нього, після define указиватся список, в якому перший елемент це ім'я функції, інші - параметри. Потім йде тіло функції. Оголошена вище функція зведення в квадрат приймає один аргумент - n. Приклад оголошення функції з двома аргументами:

```
(define (sum x y) (+ x y))
(sum 3 5) ; 8
```

Незважаючи на наявність такого виду записи оголошень функцій, потрібно не забувати, що в Racket немає іменованих функцій. define, це завжди зіставлення імені зі значенням, але в ролі останнього може виступати і функція.

Завдання

Створіть функцію з ім'ям sum-of-squares (використовуючи короткий синтаксис), яка знаходить суму квадратів двох чисел.

```
(sum-of-squares 2 3) ; 13
```

Модулі

Система модулів в Racket схожа на подібні системи в інших мовах. Кожен файл зазвичай містить рівно один модуль:

```
; math.rkt
#lang racket

(define (sum a b) (+ a b))
```

За замовчуванням всі оголошення, які робляться в модулі, залишаються всередині модуля. Імпорт оголошень з інших модулів (читай "файлів") відбувається за допомогою форми `require`:

```
(require "math.rkt")
```

Однак, ви не зможете використовувати оголошення іншого модуля, якщо модуль не експортує їх явно. Для експорту оголошень використовується форма `provide`:

```
; math.rkt
#lang racket

(provide sum)

(define (sum a b) (+ a b))
```

В `provide` перераховуються імена оголошень, які потрібно експортувати. Будь-який інший модуль автоматично отримує доступ до всіх експортованих оголошенням при імпорті:

```
#lang racket

(require "math.rkt")

(define result (sum 5 3))
```

При експорті можна так само вказати форму (provide (all-defined-out)). Таким чином ми експортуємо всі оголошення модуля.

Зверніть увагу на наступні деталі:

• require працює з шляхами, йому можна передати як абсолютний так і відносний шлях. В наведеному вище прикладі передбачається що обидва модуля лежать в одній директорії.

• require автоматично робить доступним все, що в імпортованому модулі зазначено в provide. Це буває незручно: по-перше, можливі конфлікти імен, по-друге, хочеться явно розуміти що звідки береться. Тому існує альтернативний спосіб виклику require:

```
(require (only-in "math.rkt" sum))
```

`only-in` каже що треба включити з модуля "math.rkt" змінну sum і більше нічого. Для включення додаткових оголошень досить додати їх імена в кінець списку.

Завдання

Експортуйте оголошення, яке є в завантаженому модулі

Локальні об’яви

`define` в Racket може використовуватися як на рівні модуля, так і всередині функцій:

```
(define (f)
  (define text "lorem")
  (displayln text))

(f) ; => "lorem"
; у define локальна область видимості
(displayln text) ; error
```

Але з ним пов'язано кілька тонких моментів:

• Хоча Ліспі схожі між собою, конкретно define поводиться абсолютно по-різному в різних діалектах Lisp. У деяких оголошення залишаться локальними для поточної області видимості, в інших же оголошення завжди буде глобальним.

• Оголошення змінних має йти на самому початку функції, до будь-яких інших виразів.

Існує й інший спосіб оголосити локальні змінні, набагато більш популярний і передбачуваний:

```
; створюється локальне оголошення x і потім його значення множиться на два
(let ([x 5]) (* x 2))
```

Всі оголошення всередині let доступні тільки у виразах, які викликаються всередині самого let після списку оголошень (оголошення не бачать один одного! Детальніше - нижче). Ось більш складний приклад, з кількома оголошеннями:

```
(let ([x 2]
      [y (+ 4 3)])
  (+ x y)) ; 9
```

Кожне оголошення в let - це список з імені і вирази, обчислене значення якого, буде асоційоване з ім'ям. У наших прикладах такі оголошення записуються в квадратних дужках виключно для вашої зручності - інтерпретатора практично завжди зрозуміло, що ми маємо на увазі, незалежно від того, які дужки ми використовували (так-так, в Racket практично скрізь можна використовувати квадратні дужки замість круглих!).

Напишемо функцію, яка вважає суму квадратів двох чисел, використовуючи локальні оголошення:

```
(define (sum-of-squares x y)
  (let ([x-square (* x x)]
        [y-square (* y y)])
    (+ x-square y-square)))
```

Можна піти ще далі і оголосити локальну функцію, що зводять аргумент в квадрат

```
(define (sum-of-squares x y)
  (let ([square (lambda (n) (* n n))])
    (+ (square x) (square y))))
```

Оголошення, створені в рамках одного використання let, не помітні один одному. Тому ми не можемо зробити так:

```
(let ([x 2]
      [y (* x 10)]) ; тут - помилка, тому що
                    ; змінна x ще не оголошена.
  (+ x y))
```

Форма let так влаштована, що кожне оголошення зі списку вона створює "з чистого аркуша", тому оголошення не залежать одне від одного, що іноді зручно. Скажімо, ми можемо вільно міняти місцями оголошення в межах одного списку. Якщо ж нам неодмінно потрібно використовувати в наступних оголошеннях попередні, ми можемо скористатися формою `let*`:

```
(let* ([x 2]
       [y (* x 20)]
       [z (+ x y)])
  z)                ; 42
```

Завдання

Реалізуйте функцію square-of-sum, яка спочатку складає числа, а потім зводить в квадрат. Скористайтеся локальними оголошеннями для зберігання проміжних результатів обчислення.

```
(square-of-sum 2 3) ; 25
```

Логічні оператори

True і False в Racket представлені значеннями #t і #f. Запис незвична, але в мовах створених багато років тому зустрічається і не таке. Більшість операцій в Racket розглядають як брехня тільки #f. Все інше вважається істиною. Пара прикладів перевірки на рівність:

```
(equal? 42 42) ; #t
(equal? 42 24) ; #f
```

Рівність значень перевіряється через функцію equal ?. Іноді може знадобитися порівняння по посиланню, в такому випадку використовують eq ?.

Напишемо функцію gt ?, яка повертає #t, якщо перше число більше другого і #f в іншому випадку. У Racket імена предикатів закінчуються знаком. При цьому до них не додається префікс “is”.

```
(define (gt? x y) (> x y))
(gt? 3 2) ; #t
(gt? 10 15) ; #f
```

Ось так розробники на Ruby дізналися чому в їхній мові предикати виглядають як питання :)

Тепер напишемо предикат, який визначає парність числа. Для цього нам знадобиться функція remainder, яка обчислює залишок від ділення.

```
(define (even? n) (= (remainder n 2) 0))
(even? 3) ; #f
(even? 4) ; #t
```

Логічні оператори в Racket не мають символьних позначень, замість цього використовуються функції and, or, not і інші.

```
(not "moon") ; #f
(and (odd? 3) (even? 4)) ; #t
```

Як і у випадку з арифметичними операціями, ми отримуємо два бонуси:

1. Префіксна нотація дозволяє комбінувати будь-яке число умов: (and <one> <two> <three> <...>).

2. Завдяки структурі дерева вихідного коду, пріоритет завжди точно визначений.

Завдання

Реалізуйте функцію same-parity ?, яка приймає на вхід два числа і повертає #t в тому випадку якщо їх парність збігається. В іншому випадку повертається `#f`.

```
(same-parity? 3 7) ; #t
(same-parity? 4 8) ; #t
(same-parity? 4 7) ; #f
(same-parity? 3 10) ; #f
```

Умовна конструкція

(if test-expr then-expr else-expr)

Умовна конструкція в Racket - це спеціальна форма, в якій 4 елементи. Перший - if, потім вираз-предикат (test-expr). Якщо предикат повернув істину, то виконується then-expr інакше else-expr. У Racket, if завжди містить гілку `else`.

```
(if (> 3 2) "yes" "no") ; yes
```

Підкреслю що форма if цей вислів, а значить у неї є результат. Це різко відрізняється від більшості звичних мов, в яких if це спеціальна інструкція. Вирази роблять код простіше, а його можливості ширше. Всі подальші форми, які ми розглянемо, також будуть виразами.

Чому if називається особливою формою? Давайте розберемо на прикладі:

```
(if (> 3 2) (displayln "yes") (displayln "no"))
```

Що напечает на екран ця програма? У нормальних формах, спочатку обчислюються всі елементи форми, а потім уже сама форма, це, так званий, аплікативного порядок обчислення. Він використовується в більшості мов і звичний для більшості програмістів. Ми очікуємо що аргументи функції обчислюються до того як потраплять в функцію.

Але у випадку з прикладом вище, поведінка іншого. Воно залежить від результату предиката. Тобто в формі if виконується рівно то вираз, яке повинно виконуватися за логікою форми if: перше, якщо предикат повернув істину, і друге, якщо брехня. Саме тому форма є особливою, подібну форму неможливо реалізувати на самій мові без механізму макросів (а з макросами можна).

Существует и другой порядок, так называемый нормальный порядок вычисления. Самый известный язык, использующий его - Haskell. В этом языке ничто не вычисляется до тех пор, пока не понадобится результат вычисления.

Завдання

Реалізуйте функцію sentence-type, яка повертає рядок "cry" якщо передані текст написаний великими літерами і повертає рядок "common" - в інших випадках.

```
(sentence-type "HOW ARE YOU?") ; "cry"
(sentence-type "Hello, world!") ; "common"
```

Для перекладу рядка в верхній регістр, використовуйте функцію string-upcase.

When i Unless

Звичайного if без else в Racket немає, але є дві спеціальні форми: when і unless, призначені для цієї мети.

When

(when test-expr body …+)

Якщо результат `test-expr` істина, то обчислюється тіло.

```
(when (positive? -5)
  (display "hi"))

(when (positive? 5)
  (display "hi")
  (display " there"))
```

Unless

Теж саме, що і (when (not test-expr) body …+).

unless працює навпаки. Тіло обчислюється в тому випадку, якщо test-expr - брехня. Unless хоч і буває зручний, але різко стає читаним коли в test-expr з'являються складові умови.

```
(unless (positive? 5)
  (display "hi"))
(unless (positive? -5)
  (display "hi")
  (display " there"))
```

Завдання

Реалізуйте функцію say-boom, яка повертає рядок Boom! якщо її викликали з параметром `"go"`

```
(say-boom "hey")
(say-boom "go") ; "Boom!"
```

Case

Замість switch в Racket іпользуются case. У загальному випадку, case, за своїми можливостями ширше, ніж switch в більшості мов програмування. Його використання в якості switch, це найбільш простий спосіб познайомитися з ним:

```
(let ([v 0])
  (case v
    [(0) "zero"]
    [(1) "one"]
    [(2) "two"]
    [(3 4 5) "many"]))
; "zero"
```

Кожна гілка в case описується квадратними дужками, де в лівій частині список з одного або декількох елементів. Ці елементи і є очікувані значення. У правій частині гілки знаходиться повертається значення.

Для поведінки за замовчуванням, в самому кінці, використовується частина `else`:

```
(case 6
  [(0) "zero"]
  [(1) "one"]
  [(2) "two"]
  [else "many"])
; "many"
```

Задання

Реалізуйте функцію humanize-permission, яка приймає на вхід символьне позначення прав доступу в Unix системах і повертає текстовий опис:

```
(humanize-permission "x") ; execute
(humanize-permission "r") ; read
(humanize-permission "w") ; write
```

Cond

Для найскладніших випадків, там де зазвичай застосовується if-else_if в Racket є ще одна форма: `cond`:

```
(cond
 [(positive? -5) "first return"]
 [(zero? -5) "second return"]
 [(positive? 5) "third return"]
 [else "boom!"])
```

Ця форма нагадує case, тільки в лівій частині всередині квадратних дужок знаходиться предикат. Якщо його результат істина, то виконується права частина і її результат повертається з cond.

Якщо необхідно, в кінці додається else, який веде себе аналогічно else в інших мовах.

Завдання

Реалізуйте функцію programmer-level, яка приймає на вхід кількість балів, і повертає рівень розробника на основі цього числа. Якщо балів від 0 до 10, то це junior, від 10 до 20 - middle, все що вище 20 - senior.

```
(programmer-level 10) ; middle
(programmer-level 0) ; junior
(programmer-level 40) ; senior
```