

Функціональне програмування

Лектор Ковалюк Тетяна Володимирівна
к.т.н., доцент
tkovalyuk@ukr.net



Scheme

ОСНОВИ МОВИ SCHEME – діалекту LISP

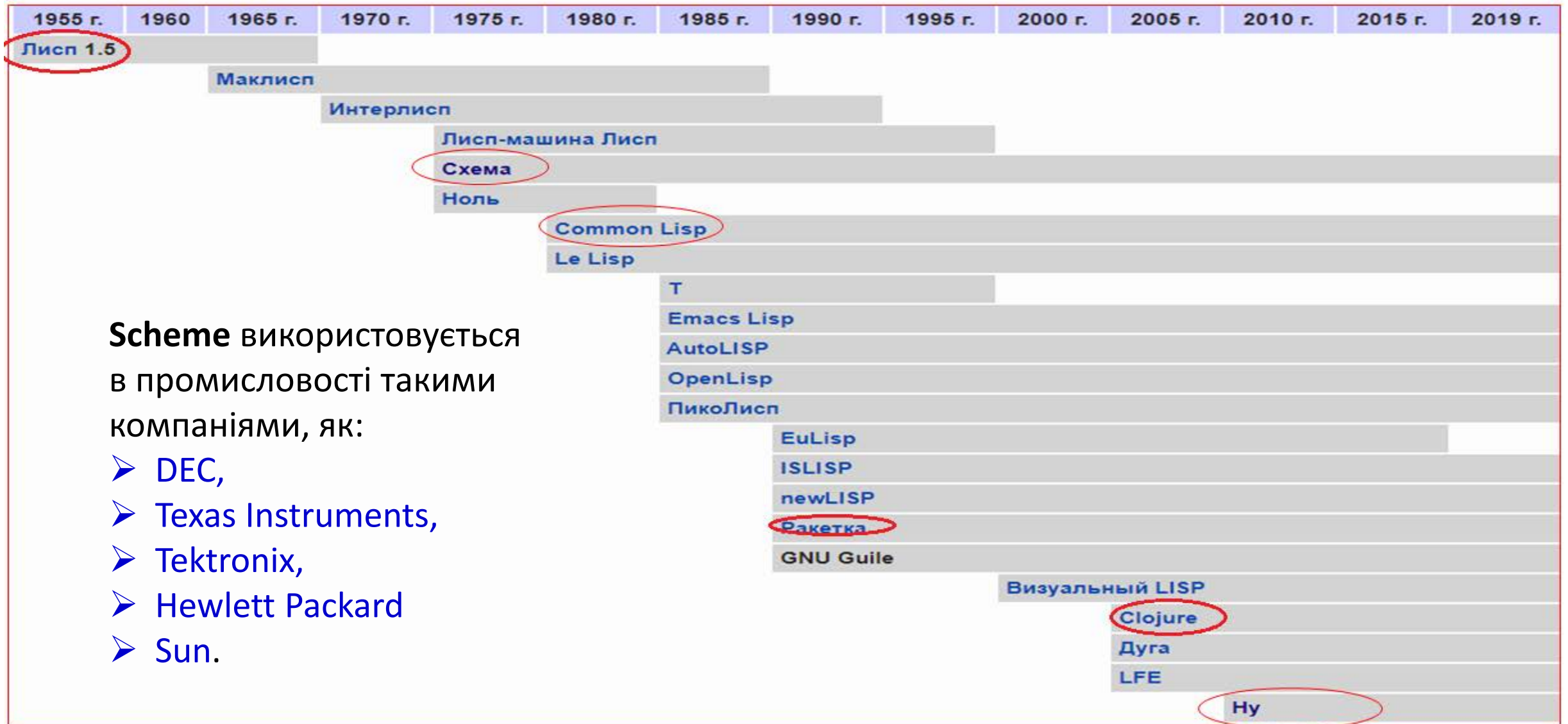
План лекції 2

1. Історія Lisp
2. Використання мови Scheme
3. Основні типи мови Scheme
4. Вирази та значення виразів
5. Комбінаційні форми
6. Інфіксний та префіксний запис арифметичних виразів
 1. Імена, зв'язування та оточення
7. Складені процедури
8. Умовні вирази та предикати
9. Елементарні предикати та операції логічної композиції
10. Приклад. Обчислення квадратного кореня методом Ньютона

Історія Lisp

- ❑ Назва мови **ЛІСП** походить від англійського «**List Processing**» (обробка списків).
- ❑ Мова створювалася для символної обробки задач диференціювання та інтегрування алгебричних виразів.
- ❑ Мова створена Джоном Маккарті у 1960 р.
- ❑ Областю її застосування стали символні обчислення, які є типовими для задач штучного інтелекту (Artificial Intelligence, AI). В основу мови покладено строгий математичний апарат:
 - лямбда-числення Чьорча;
 - алгебра спискових структур;
 - теорія рекурсивних функцій
- ❑ Сьогодні ЛІСП представляє сімейство діалектів, один з яких **SCHEME**.
- ❑ Лісп має унікальну властивість розглядати **процедури як дані**. Це робить його одною із самих зручних мов для дослідження методів проектування програм, в яких не потрібна відмінність між пасивними даними та активними процесами їх обробки.

Хронологія діалектів LISP



Scheme використовується в промисловості такими компаніями, як:

- DEC,
- Texas Instruments,
- Tektronix,
- Hewlett Packard
- Sun.

Сфери використання декларативних мов програмування

Lisp використовується для AI, оскільки він підтримує реалізацію програмного забезпечення, яке добре оперує з символами. Символи, символьні вирази і їх обчислення закладено в ядрі Lisp. Типові області штучного інтелекту для обчислення з символами були/є: комп'ютерна алгебра, доведення теорем, системи планування, діагностика, перепису системи, представлення знань і висновки, логічні мови, машинний переклад, експертні системи та ін.

Відомі застосування AI, написані мовою Lisp

Macsyma – перша система комп'ютерної алгебри.

ACL2 – система для доведення теорем, наприклад, використовується AMD.

DART – планувальник логістики, який використовувався під час першої війни в Перській затоці американськими військовими. Завдяки цьому застосуванню окупилися американські інвестиції в наукові дослідження AI в той час.

SPIKE – застосування планування та розкладу для космічного телескопу Hubble.

CYC – одна із найбільших розроблених програмних систем: представлення і обробка в галузі людських загальновідомих почуттів.

METAL – одна з перших систем перекладу мови комерційного використання.

Express Authorizer's Assistant – американське програмне забезпечення для перевірки транзакцій за кредитними картками.

Мова Lisp прийнята як одна з двох основних мов програмування для міністерства оборони США.

Систему AutoCAD розроблено мовою Lisp.

Властивості функціонального програмування

- 1) Представлення програми і даних відбувається однаково – через списки. Це дає змогу програмі обробляти інші програми і навіть саму себе.
- 2) Функційні мови, як правило, є інтерпретуючими мовами.
- 3) Функційні мови є безтиповими, це означає, що символи не зв'язуються за замовчуванню з яким-небудь типом.
- 4) Функційні мови мають незвичний синтаксис через велику кількість дужок.
- 5) Функційні програми, написані для обробки символічних даних, є набагато коротші, аніж написані імперативними мовами

Історія розвитку мови SCHEME

- Перший опис Scheme написано в [1975](#) році. Творцями мови є [Гай Стіл](#) (англ. *Guy L. Steele*) та [Джеральд Сассмен](#) (англ. *Gerald Jay Sussman*) з [Массачусетського технологічного інституту](#).
- В [1981](#) та [1982](#) роках почались три різних проекти по використанню Scheme для навчального процесу в університетах [США](#).
- Мова програмування Scheme визначається двома стандартами: стандартом де-юре в редакції [IEEE](#), та стандартом де-факто. Поточна версія описання стандарту де-факто має назву «Revised 5 Report on the Algorithmic Language Scheme». [28 серпня 2007](#) року було затверджено наступну редакцію: **R⁶RS**. Зараз доступна редакція **R⁷RS**
- Основну увагу, при створенні діалекту, було приділено **елегантності та концептуальній довершеності мови**. Як наслідок, повна специфікація мови програмування [Scheme](#) вмістилася в 50 сторінок, в той час, як специфікація [Common Lisp](#) має розмір 1300 сторінок.

Використання мови Scheme

1. Мова семантики і специфікації стилів документа (**DSSSL**), який забезпечує метод визначення таблиць стилів **SGML**, використовує підмножину **Scheme**.
2. Редактор растрової графіки з відкритим вихідним кодом **GIMP** використовує **Scheme** в якості мови сценаріїв.
3. **GNU Guile** - це краща мовна система розширень для проекту GNU, в якій реалізована реалізація мови **Scheme**. Guile використовується в таких програмах, як GnuCash , LilyPond , GNU Guix , Guix System Distribution (GuixSD) и GNU Debugger .
4. **Scheme** використовується **Synopsys** в якості мови сценаріїв для своїх технологічних інструментів CAD (TCAD).
5. У фільмі **The Spirits Within** використовувалась мова **Scheme** в якості **мови сценаріїв** для управління двигуном рендеринга в реальному часі.
6. **Google App Inventor** для **Android** використовує **Scheme** для роботи на віртуальній машині Java, що працює на пристроях Android.
7. **Ігровий движок Haxima** написаний на Scheme, і гра Nazghul, також. Обидва є безкоштовним програмним забезпеченням.

Особливості мови Scheme

1. *Scheme* є мовою програмування зі **статичними областями видимості**.
2. *Scheme* має **неявні**, а не декларативні **типи даних**. Типи пов'язані з об'єктами (значеннями), а не із змінними. Отже, **Scheme є слабо типизована мова**.
3. Усі об'єкти, створені в процесі обчислення *Scheme*, включаючи **процедури і продовження**, мають необмежений екстент. **Жоден об'єкт Scheme ніколи не знищується**.
4. **Реалізації** *Scheme* повинні **бути чистими хвіст-рекурсивними**. В чистій хвіст-рекурсивній реалізації ітерація може бути виражена за допомогою звичайного механізму виклику процедур.
5. *Scheme* була одною з перших мов, що підтримують **процедури як об'єкти** самі по собі. Процедури можуть динамічно створюватися, заноситися в структури даних, повертатися в якості результатів процедур, і так далі.
6. **Арифметична модель** *Scheme* надає багатий набір числових типів і операцій з ними. В цій моделі розрізняються **точні і неточні числові об'єкти**: точний числовий об'єкт точно відповідає числу, а неточний числовий об'єкт є результатом обчислення, що спричинило округлення або інші помилки.

Механізми програмування

Будь-яка мова програмування реалізує **три механізми**:

1. **Елементарні вирази**, що представляють мінімальні сутності, з якими мова має справу.
2. **Засоби комбінування**, за допомогою яких з простих об'єктів складаються складні.
3. **Засоби абстракції**, за допомогою яких складні об'єкти можна називати і поводитися з ними як з єдиним цілим.

Будь-яка мова має справу з **двома типами об'єктів** :

- ☐ **процедури**
- ☐ **дані**

Дані - це "матеріал", який потрібно обробити,
Процедури - це описи правил обробки даних.

Основні типи мови Scheme

Об'єкт \cong значення

Значення 1

Значення 2

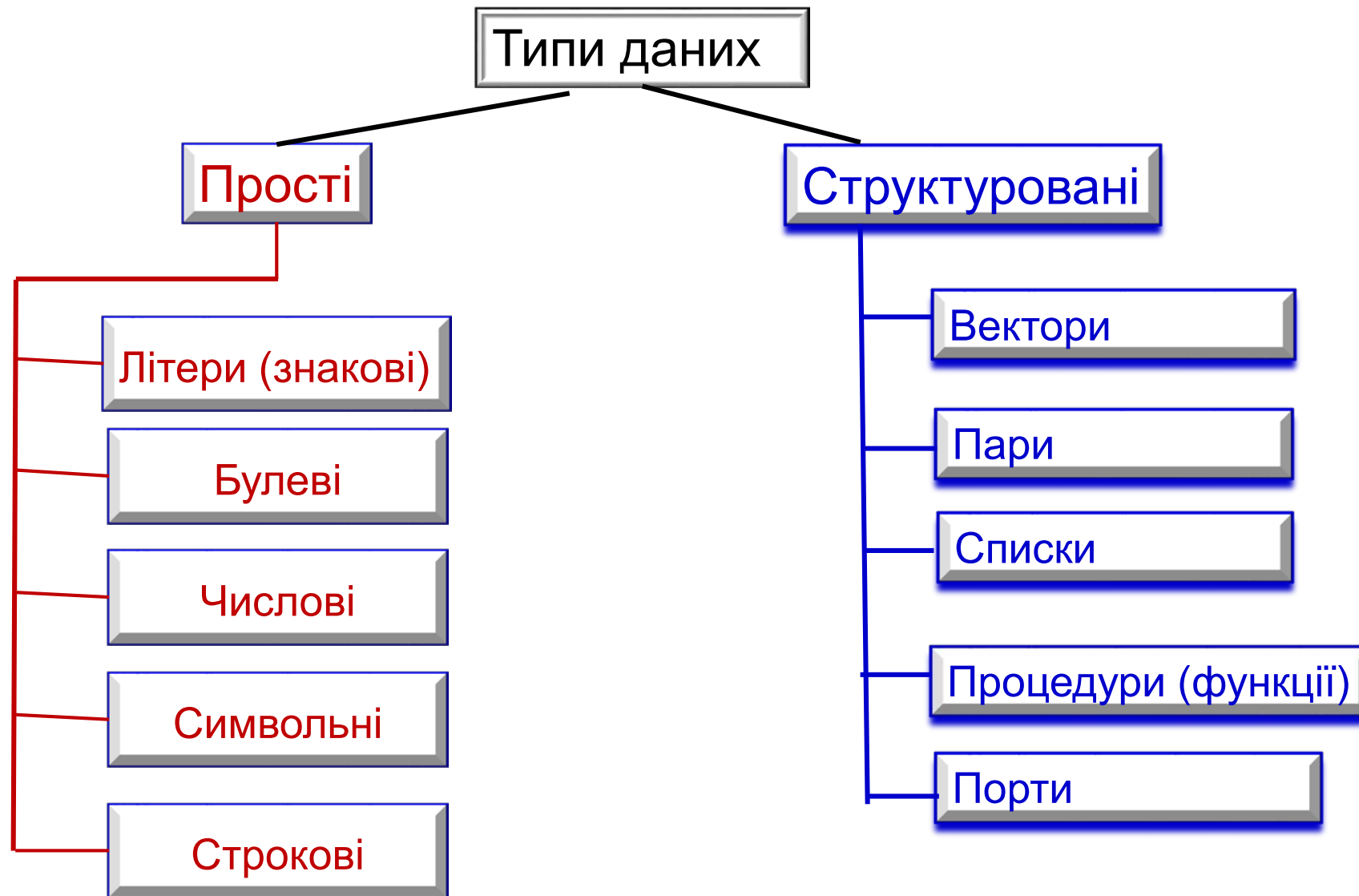
...

Значення n

Тип

- ❑ Програми мовою Scheme оперують **об'єктами**, які називаються **значеннями**.
- ❑ Об'єкти Scheme організовані в **набори значень**, які називаються **типами**.

Основні типи мови Scheme



Основні типи мови Scheme

Булеві

Булевий тип є логічним значенням і може бути **true** або **false**.

Об'єкт "false" в Scheme записується як **#f**.

Об'єкт "true" записується як **#t**.

Будь-який об'єкт, відмінний від **#f**, інтерпретітується як **true**.

Числові

Scheme підтримує безліч числових типів даних, включаючи об'єкти, що представляють цілі числа довільної точності, раціональні числа, комплексні числа і наближені числа різних видів.

47 1/3 2.3 4.3e14 1+3i



- ☐ Кожне ціле число (integer) є **дробом** (rational),
- ☐ Кожний дріб є **дійсним числом** (real),
- ☐ Дійсне число є **комплексним** (complex).
- ☐ Числа класифікуються за ознакою точності (точне або приблизне - exact / inexact):

Основні типи мови Scheme

Літери (Знакові)

Знаки Scheme в основному відповідають текстовим символам. Вони ізоморфні до скалярних значень стандарту Unicode.

`#\a` `#\A` `\#space` `#\newline`

Строкові

Рядки є кінцевими послідовностями символів фіксованої довжини і, таким чином, представляють довільні тексти Unicode.

`"рядок тексту"`

Символьні

Символ використовується для **позначення** змінних.

`this-is-a-symbol` `abc` `a32` `>adfasf23@#$%!<`

☐ Символи не чутливі до регістру букв.

☐ Символ може містити букви англійського алфавіту, цифри і літери `+ - . * / <=> ! ? : $ % _ & ~ ^`

Основні типи мови Scheme

Пари і списки

Пара є структурою даних з двома компонентами.

Списки, є лінійними структурами даних, які формуються з пар, де перший компонент ("car") є першим елементом списку, а другий компонент ("cdr") представляє іншу частину списку.

У Scheme є окремий **порожній список**, який є останнім в ланцюжку пар, що формують список.

Вектори

Вектори, як і списки, є лінійними структурами даних, що представляють кінцеві послідовності довільних об'єктів. Оскільки елементи списку доступні послідовно через ланцюжок пар, що представляють його, до елементів вектора звертаються за **цілими індексами**. Таким чином, вектори забезпечують **довільний доступ до елементів**.

Вектор може складатися з **будь-яких** об'єктів даних.

Процедури

У Scheme процедури є значеннями.

Порти

Відкриті файли або мережеві з'єднання

Вирази та значення виразів

Вирази можуть обчислюватися, породжуючи значення.

Літеральні вирази

#t 23

Така форма запису означає, що вираз **#t** обчислюється в значення **#t**, тобто значення для "true"
вираз **23** обчислюється в чисельний об'єкт, який представляє число **23**.

Складені вирази

Створюються шляхом розміщення круглих дужок навколо своїх підвиразів.
Перший підвираз ідентифікує операцію; інші підвирази є операндами операції:

(+ 23 42) ⇒ 65
(+ 4 (* 2 3)) ⇒ 10



**Складені вирази в Scheme завжди записуються
за допомогою префіксної нотації.**

7 типів виразів

1. Константи: `'foo #\Z 3 "строка"`
2. Посилання на змінні: `foo joe a-long-name @$!+-*/%<>`
3. Створення процедур: `(lambda (x) (* x x x))`
4. Застосування процедур: `(cube 37)`
5. Умова: `(if (< x 3) sqrt modulo)`
6. Завдання значень (альтернатива присвоєння): `(set! x 5)`
7. Послідовність: `(begin (write x) (write y) (newline))`

Приклади виразів та їх значень

Звичайно робота з інтерпретатором Scheme відбувається за сценарієм:

- користувач уводить вираз
- інтерпретатор обчислює значення цього вираз й друкує результат.

```
10
 10
10.5
 10.5
```

результат

Числові константи позначають числа, які є їхніми значеннями. Розмаїстість доступних типів чисел залежить від реалізації мови, але всі реалізації підтримують **цілі й дійсні числа**, а багато хто ще й **раціональні й комплексні**.

Символ – це зображення букв, цифр і спеціальних знаків (**!\$%&* /:=<>?^_ ~ +-.@**), що відрізняється від числа.

Головне призначення символів - **іменувати об'єкти**. **Тому, значенням символу є об'єкт, поименований цим символом.**

За допомогою лапок символи можна вживати автономно.

Значенням виразу **'<символ>** є сам цей символ.

```
'Hello
hello
```

Scheme нечутлива до регістра букв



Приклади виразів та їх значень

Hello

Error: undefined variable

Оскільки з ім'ям **Hello** поки не зв'язано жодним значенням, одержуємо повідомлення про помилку.

Рядкові константи записуються в подвійних лапках і представляють послідовності відображуваних знаків:

"Hello"

"Hello"

З ім'ям **+, -, *, /** зв'язана вбудована функція, що обчислює суму (різницю, добуток, частку) чисел, що їй є значенням. Однак сама функція не відображається: внутрішні подання функцій не читаються.

+

#<procedure

+>

Дві логічні константи **#t** й **#f** позначають **істину** й **хибність**

#f

#f

(= 1 2)

#f



Константи й символи носять загальне ім'я **атоми**, оскільки являють собою найпростіші елементи мови, з яких будуються вирази

Комбінаційні форми

Вирази, що представляють числа, можуть поєднуватися з виразом, що представляє елементарну процедуру (наприклад, + або *), так що **складений вираз** є застосування процедури до цих чисел.

- ❑ Вирази, що утворюються шляхом запису списку виразів в скобках з метою позначити застосування функції до аргументів, називаються **комбінаціями (combinations)**.
- ❑ Найлівіший елемент в списку називається **оператором** (operator), а інші елементи - **операндами** (operands).
- ❑ Значення комбінації обчислюється шляхом застосування процедури, що задається **оператором** до **аргументів** (arguments), які є значеннями операндів.
- ❑ Правило, за яким оператор ставиться зліва від операндів, відоме як **префіксна нотація (prefix notation)**

(+ 137 349)
486
(- 1000 334)
666
(* 5 99)
495
(/ 10 5)
2

Комбінаційні форми

Префіксний запис може поширюватися на процедури з довільною кількістю аргументів:

(+ 21 35 12 7)

75

(* 25 4 12)

1200

Префіксна нотація може розширюватися, дозволяючи комбінаціям вкладатися одна в одну:

(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))

57

Згідно з правилами форматування будь-яка довга комбінація записується так, щоб її операнди вирівнювалися вертикально:

Загальне правило обчислення значення комбінації:

1. Обчислити значення всіх підвиразів.
2. Застосувати функцію, що є значенням оператора, до аргументів, які є значеннями операндів.

3. Правило обчислення рекурсивне за своєю природою

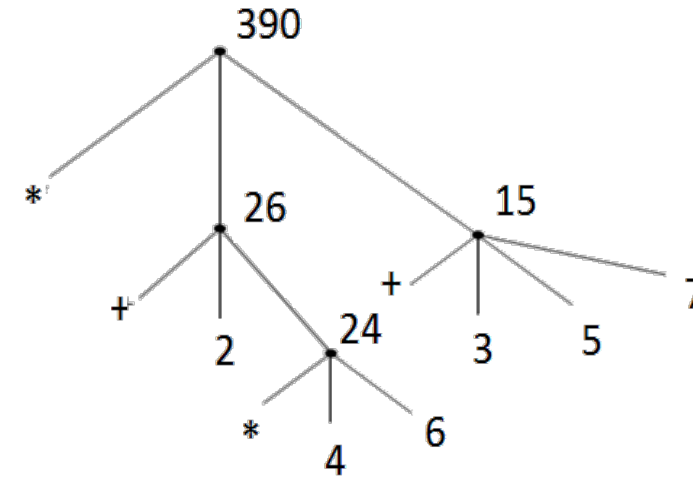
(+ (* 3
 (+ (* 2 4)
 (+ 3 5)))
 (+ (- 10 7)
 6))

Рекурсивне обчислення комбінацій

Рекурсивний процес обчислення комбінації зобразимо, намалювавши комбінацію у вигляді дерева:

Завдання:

$(* (+ 2 (* 4 6))$
 $(+ 3 5 7))$



Порядок обчислення по дереву:

$$4 * 6 = 24$$

$$3 + 5 + 7 = 15$$

$$2 + 24 = 26$$

$$26 * 15 = 390$$

- ❑ Кожна **комбінація** представляється у вигляді вершини, а її **оператор і операнди** - у вигляді гілок, що виходять з цієї вершини.
- ❑ Кінцеві вершини представляють **оператори або числа**.
- ❑ Значення операндів розповсюджуються від кінцевих вершин вгору і потім комбінуються на все вищих рівнях.
- ❑ Правило обчислення "**розповсюдити значення вгору**" є прикладом загального типу процесів, відомого як **накопичення по дереву**

Інфіксний та префіксний запис арифметичних виразів

Математичний запис - інфіксний	Запис на Ліспі (Scheme) префіксний
$f(x)$	$(f\ x)$
$g(x, y)$	$(g\ x\ y)$
$h(x, g(y, z))$	$(h\ x\ (g\ y\ z))$
$\sin x$	$(\sin\ x)$
$x + y$	$(+\ x\ y)$
$x + y \cdot a$	$(+\ x\ (*\ y\ a))$
x^y	$(\text{expt}\ x\ y)$
$ x $	$(\text{abs}\ x)$
$x = y$	$(=\ x\ y)$
$x + y < a$	$(<\ (+\ x\ y)\ a)$

Види виразів

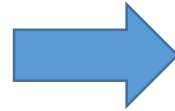
Можливі сім видів виразів:

1. Константи: `'foo # \ x 3 "рядок"`
2. Посилання на змінні: `foo joe a-long-name @ # $! + - * /% <>`
3. Створення функції: `(lambda (x) (* x x x))`
4. Застосування процедури: `(cube 3 7)`
5. Умова: `(if (<x 3) sqrt modulo)`
6. Присвоювання значення: `(set! X 5)`
7. Послідовність: `(begin (write x) (write y) (newline))`

Імена та зв'язування

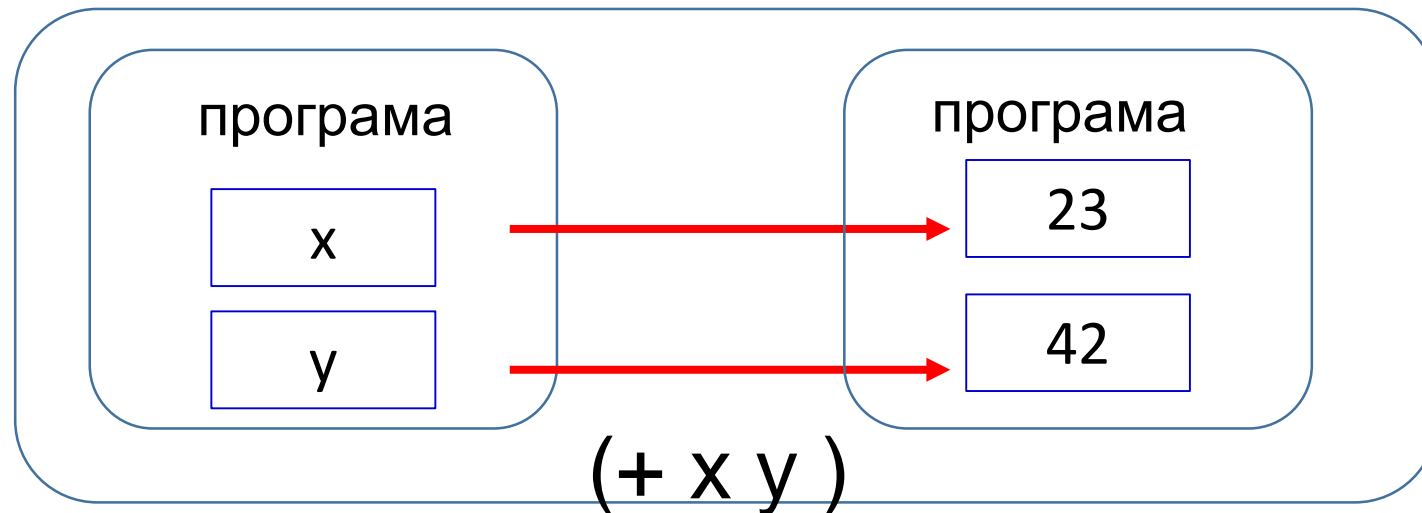
- ❑ Ім'я означає **змінну**, чиїм значенням є **об'єкт**
- ❑ Імена використовуються для вказівки на обчислювальні об'єкти.
- ❑ Scheme дозволяє **зв'язувати** ідентифікатори та значення.

```
(let ((x 23)  
      (y 42))  
  (+ x y))
```



65

х,у - локальні змінні



Імена та оточення

Scheme дозволяє створювати зв'язування верхнього рівня для ідентифікаторів шляхом створення нових об'єктів

Для визначення нових об'єктів, застосовуються *визначальні форми*:

(define name value) зв'язує ім'я **name** зі значенням виразу **value**.

(define (f a1...an) body) визначає функцію з ім'ям **f**.

a1...an - **формальні параметри**, тобто імена, що використовуються усередині тіла функції для посилань на відповідні параметри.

body - тіло функції - вираз, що визначає її значення.

(define size 2)

примушує інтерпретатор зв'язати значення 2 з ім'ям size

Після того, як ім'я **size** пов'язано зі значенням 2, можемо вказувати на значення 2 за допомогою імені **size** :

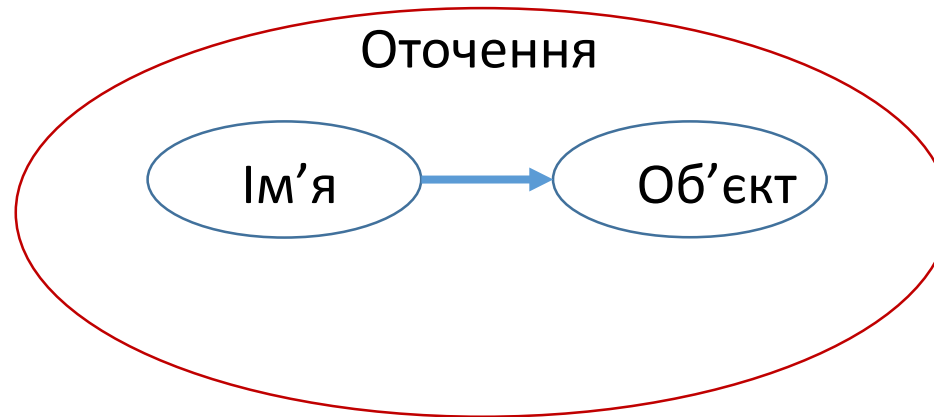
```
size  
2  
(* 5 size)  
10
```

Імена та оточення

Раз інтерпретатор здатний асоціювати значення з символами і потім згадувати їх, то він повинен мати деякого роду пам'ять, що зберігає пари

ім'я-об'єкт.

Ця пам'ять називається **оточенням** (environment).



Імена та оточення

```
(define pi 3.14159)
```

```
(define radius 10)
```

```
(* pi (* radius radius))
```

```
314.159
```

```
(define circumference (* 2 pi radius)) circumference
```

```
62.8318
```

Слово **define** служить найпростішим **засобом абстракції**, тому що воно дозволяє використовувати прості імена для позначення результатів складних операцій

Правило обчислення комбінацій не обробляє визначень.

Наприклад, обчислення **(define x 3)** не означає застосування **define** до двох аргументів, один з яких є значення символу **x**, а інший дорівнює **3**, оскільки сенс **define** якраз і полягає в тому, щоб **зв'язати x** зі значенням **3**.

Програмування виразів мовою SCHEME

Потрібно обчислити значення виразу

$$X = (A \sin B + B \cos A) / (1 - \sin C * |B + D|)$$

при заданих значеннях операндів A, B, C, D

Запис виразу мовами C/C++/C#

$$X = (A * \sin(B) + B * \cos(A)) / (1 - \sin(C) * (|B + D|))$$

```
=====
===== Ex1/ приклад розрахунку виразу=====
=====
; X = (A sin B + B cos A) / (1 - sin C * |B + D|)
(define A 1)
(define B -12)
(define C 3)
(define D 4)
(/ (+ (* A (sin B))
      (* B (cos A))
    )
   (- 1 (* (sin C) (abs (+ B D)))))
)
```

Інтерпретатор або компілятор здійснює розрахунок без додаткового оформлення виразу в процедуру. Після виконання команди Run отримуємо результат. Пробіли між іменами sin, cos, abs та аргументами **обов'язкові**

Складені процедури

У мові Лісп/Scheme присутні:

- **Числа і арифметичні операції** є елементарними даними і процедурами.
- **Вкладення комбінацій** дає можливість комбінувати операції.
- **Визначення**, які зв'язують імена зі значеннями, дають обмежені можливості абстракції.

Лісп має **визначення процедур** (procedure definitions) - значно потужніший метод абстракції, за допомогою якого **складеній операції можна дати ім'я і потім посилатися на неї як на єдине ціле.**

Загальна форма визначення процедури така:

((define (ім'я) (параметри) (тіло)))

Складені процедури

Загальна форма визначення процедури така:

(define ім'я) (формальні-параметри) (тіло)

(Ім'я) - це той символ, з яким треба зв'язати в оточенні визначення процедури.

(формальні-параметри) - це імена, які в тілі процедури використовуються для посилання на відповідні аргументи процедури.

(тіло) - це вираз, який обчислює результат застосування процедури, коли формальні параметри будуть замінені аргументами, до яких процедура буде застосовуватися.

(Ім'я) і (формальні- параметри) взяті в дужки, як це було б при виклику процедури

Запис мовою Лісп:

```
(define (square x) * x x)
```

Ключове
слово

ім'я
функції

Формальний
параметр

Тіло
процедури

Складені процедури

Запис мовою Лісп:

```
(define (square x) * x x)
```

Ключове ім'я Формальний Тіло
слово функції параметр процедури

Семантика визначення процедури `square`

`(define` ↑ щоб
`(square` ↑ піднести у квадрат
`x)` ↑ що-небудь, помножте
`*` ↑ це
`x` ↑ само на себе
`x` ↑

Приклад виклику процедури `square`

```
(square 21)  
441  
(square (+ 2 5))  
49  
(square (square 3))  
81
```

Визначення процедури для x^2+y^2

```
(define (sum-of-squares x y)  
  (+ (square x) (square y)))
```

Приклад виклику процедури `sum-of-squares`

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

Складені процедури

Тепер **sum-of-squares** можна використовувати як будівельний блок при подальшому визначенні процедур:

Визначення процедури

```
(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
```

```
(define (sum-of-squares x y)
  (+ (square x) (square y)))
```

Виклик процедури

```
(f 5)
136
```

Виконання процедури (f 5)

$5+1=6$

$5*2=10$

$6^2+10^2=136$

Результат=136

Щоб застосувати складену процедуру до аргументів, потрібно обчислити тіло процедури, замінивши кожен формальний параметр відповідним аргументом.

Аплікативний і нормальний порядки обчислення

1. Інтерпретатор спочатку обчислює **оператор** і **операнди**, а потім застосовує отриману **процедуру до аргументів**.
2. Інша модель обчислення не обчислює аргументи, поки не знадобиться їх значення. Замість цього вона підставляє на місце параметрів вирази-операнди, поки не отримає вираз, в якому присутні тільки елементарні оператори, і лише потім обчислює його.

Наприклад, для обчислення (f 5) потрібна послідовність підстановок:

```
(define (sum-of-squares x y)
  (+ (square x) (square y)))
```

```
(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
```

```
(sum-of-squares (+ 5 1) (* 5 2))
(+ (square (+ 5 1)) (square (* 5 2)) )
(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
```

Після підстановок слідує редукції:

```
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

Аплікативний і нормальний порядки обчислення

Метод «**повна підстановка, потім редукція**» відомий під назвою **нормальний порядок обчислень (normal-order evaluation)**.

Метод «**обчислення аргументів, потім застосування процедури**» називається **аплікативним порядком обчислень (applicative-order evaluation)**.

При **строгому (аплікативному)** порядку обчислення заздалегідь підраховуються значення всіх аргументів перед обчисленням самої функції.

При **нестрогому підході (нормальний порядок обчислення)** значення аргументів не обчислюються до тих пір, поки їх значення не знадобляться при обчисленні функції.

У Ліспі/Scheme використовується **аплікативний порядок обчислень** через додаткову ефективність, яка дає можливість не обчислювати багаторазово вирази.

Як правило, нестрогий підхід реалізується у вигляді **редукції графа (спрощення)**. Нестроге обчислення використовується за замовчуванням в декількох чисто функційних мовах ([Miranda](#), [Clean](#) і [Haskell](#))

Умовні вирази та предикати

Для виконання дій відповідно до певної умови використовують форма, яка називається **розбором випадків (case analysis)**. Наприклад, для обчислення модуля числа в Ліспі є конструкція **cond** (англ. *conditional*, «умовний»)

Математичне визначення

$$|x| = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{if } x = 0 \\ -x, & \text{if } x < 0 \end{cases}$$

Загальна форма умовного виразу така:

```
(cond (<p1> <e1>
      <p2> <e2>)
  ...
      <pn> <en>))
```

умова

вираз, що обчислюється

Конструкція в SCHEME

```
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        (< x 0) (- x))))
```

Конструкція **cond** складається з символу **cond**, за яким слідує в дужках пари виразів (<p> <e>), що називаються **гілками**.

<p> - **предикат**, тобто вираз, значення якого інтерпретується як істина або хибність.

<e> - **вираз**, що обчислюється

Умовні вирази та предикати

Загальна форма умовного виразу така:

```
(cond (<p1> <e1>
      <p2> <e2>)
  ...
  <pn> <en>))
```

Умовні вирази обчислюються так:

1. Спочатку обчислюється предикат <p1>.
2. Якщо його значенням є хибним, обчислюється предикат <p2>.
3. Якщо значення <p2> також хибне, обчислюється <p3>.
4. Цей процес триває доки, поки не знайдеться предикат, значенням якого буде істина, і в цьому випадку інтерпретатор повертає значення відповідного виразу-слідства (consequent expression) в якості значення всього умовного виразу.
5. Якщо жоден з предикатів <p> не виявиться істинним, значення умовного виразу не визначено.

Предикат це процедура, яка повертає істину або хибність, а також висловлювання, які мають значенням істину або хибність.

Умовні вирази та предикати

Альтернативні умовні вирази та процедури для прикладу обчислення модуля числа

«Якщо x менше за нуль,
повернути $-x$;
інакше повернути x ».

```
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```



```
(define (abs x)
  (cond ((< x 0)
        (- x))
        (else x)))
```

Загальна форма виразу if така:

(If <предикат> <наслідок> <альтернатива>)

Щоб обчислити вираз if, інтерпретатор спочатку обчислює його <предикат>.

Якщо <предикат> дає істинне значення, інтерпретатор обчислює <наслідок> і повертає його значення. В іншому випадку він обчислює <альтернативу> і повертає її значення.

Елементарні предикати та операції логічної композиції

Елементарні предикати

<, =, >

Операції логічної композиції:

and or not

1. Інтерпретатор обчислює вирази <e> по одному, зліва направо.
2. Якщо яке-небудь з <e> дає помилкове значення, значення всього виразу **and** - хибність і інші <e> не обчислюються.
3. Якщо усі <e> дають істинні значення, значенням вирази **and** є значення останнього з них.
(or <e1>... <en>)

1. Інтерпретатор обчислює вирази <e> по одному, зліва направо.
2. Якщо яке-небудь з <e> дає істинне значення, це значення повертається як результат виразу, а решта <e> не обчислюються.
3. Якщо всі <e> виявляються помилковими, значенням вирази **or** є брехня.
(not <e>)

Значення виразу **not** - істина, якщо значення виразу <e> хибне, і хибне в іншому випадку.

Елементарні предикати та операції логічної композиції

Приклади :

Число x знаходиться в
діапазоні $5 < x < 10$

Одне число більше або
дорівнює іншому

Предикат в Ліспі/Scheme

```
(and (> x 5) (< x 10))
```

```
(define (>= x y)  
  (or (> x y) (= x y)))
```

```
(define (>= x y)  
  (not (< x y)))
```

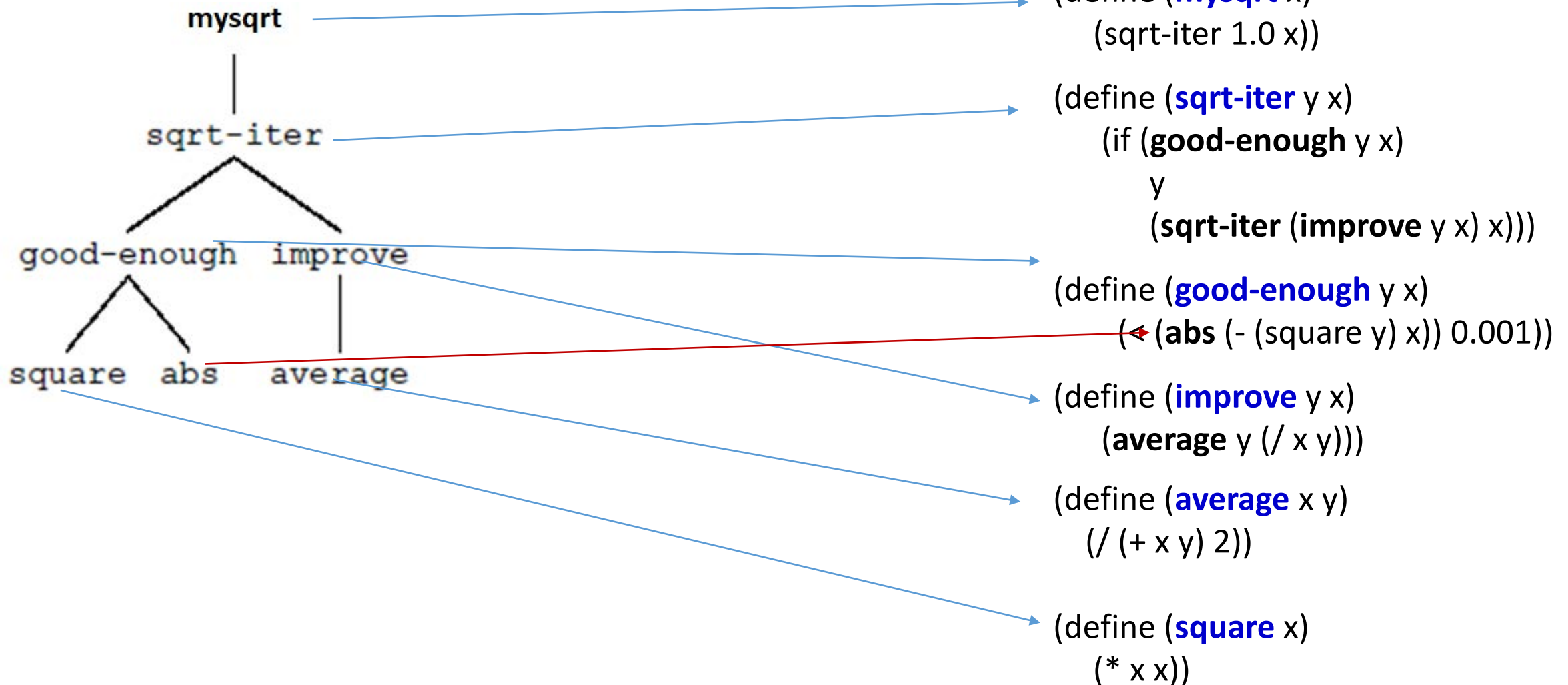
Приклад. Обчислення квадратного кореня методом Ньютона

Застосовуємо метод Ньютона послідовних наближень, який заснований на тому, що маючи деяке неточне значення y для квадратного кореня з числа x , можна отримати більш точне значення (ближче до справжнього квадратного кореня), якщо взяти **середнє** між y і x / y

Наближення	Частка x/y	Середнє
1	$\frac{2}{1} = 2$	$\frac{2 + 1}{2} = 1.5$
1.5	$\frac{2}{1.5} = 1.3333$	$\frac{1.3333 + 1.5}{2} = 1.4167$
1.4167	$\frac{2}{1.4167} = 1.4118$	$\frac{1.4167 + 1.4118}{2} = 1.4142$
1.4142

Приклад. Обчислення квадратного кореня методом Ньютона

Процедурна декомпозиція програми `mysqrt`



Приклад. Обчислення квадратного кореня методом Ньютона

```
(define (average x y)
  (/ (+ x y) 2))
```

 $(x+y)/2$

```
(define (improve y x)
  (average y (/ x y)))
```

Під час розрахунку слід поліпшувати наближення до тих пір, поки його квадрат не співпадає з підкоренним числом в межах наперед заданого допуску, наприклад, 0.001

```
(define (good_enough y x)
  (< (abs (- (square y) x)) 0.001))
```

```
(define (sqrt_iter y x)
  (if (good-enough y x)
      y
      (sqrt-iter (improve y x)
                  x)))
```

```
(define (square x) (* x x))
```

```
(define (mysqrt x)
  (sqrt-iter 1.0 x))
```

```
(mysqrt 9)
3.00009155413138
```

Приклад. Обчислення квадратного кореня методом Ньютона

```
=====
; Ex2 розрахунок кореня квадратного за методом Ньютона
; процедура розрахунку кореня квадратного з x
(define (mysqrt x)
  (sqrt-iter 1.0 x)) ;завдання початкового значення кореня з x
; рекурсивне покращення поточного значення у кореня з x
(define (sqrt-iter y x)
  (if (good-enough y x) ;якщо досягнуто задовільне значення
      y ; отримати значення кореня
      (sqrt-iter (improve y x) x))) ; інакше повторити процедуру
; перевірка досягнення прийнятного результату обчислення кореня
(define (good-enough y x)
  (< (abs (- (square y) x)) 0.001)) ; перевірка результату на точність
(define (square x) ;квадрат числа x
  (* x x))
;покращення наближення обчислення кореня з числа x
(define (improve y x)
  (average y (/ x y)))
; середнє арифметичне значення кореня та числа, з якого береться корінь
(define (average x y)
  (/ (+ x y) 2))
```

```
-----виклик процедури(функції)-----
(display "розрахунок кореня
квадратного методом Ньютона\n")
(display "sqrt(9)=")
(sqrt 9) ;виклик вбудованої
;(елементарної) процедури
(display "mysqrt(9)=")
(mysqrt 9) ;виклик написаної
;користувачем процедури
```

Результат виконання програми

```
Welcome to DrRacket, version 8.2 [cs].
Language: racket, with debugging; memory limit: 128 MB.
розрахунок корня квадратного методом Ньютона
sqrt(9)=3
mysqrt(9)=3.00009155413138
>
```

Формальні параметри процедур

- ☐ Формальні параметри процедури **локальні** по відношенню до тіла цієї процедури.
- ☐ Це означає, що за межами процедури ці параметри недоступні
- ☐ Визначення процедури **зв'язує** (binds) свої формальні параметри.
- ☐ Самі параметри називаються **зв'язаними змінними**.
- ☐ Множина виразів, для яких зв'язування визначає ім'я, називається **областю дії (scope) цього імені**.
- ☐ У визначенні процедури зв'язані змінні, оголошені як формальні параметри процедури, мають свою область дії **тіло процедури**
- ☐ Якщо змінна не зв'язана, то вона **вільна** (free).

Середовище розробки Dr.Racket

Чому Scheme, чому Dr. Racket?

Ось лише декілька тез в підтримку цих потужних мов та середовища розробки:

1. Все є виразами. Більшість мов програмування є комбінацією двох синтаксично різних частин:

- ☐ Вирази (об'єкти, які обчислюються для отримання значення) і
- ☐ Вислови (процедури, функції, оператори, тобто об'єкти, які виражають дію).

Наприклад, у мові Python, $x = 1$ є висловом, а $(x + 1)$ – виразом. Вислови і вирази різняться, оскільки вирази можуть бути вкладені один в одне, а вирази і вислови разом – ні. **Мова Lisp знімає це обмеження: так як вирази можуть бути вкладеними, так і все в мові може бути поєднане поруч зі всім.**

1. Кожен вираз є або одиночним значенням або списком. Одиночними значеннями в мові Scheme є атоми, числа, рядки.

2. Функціональний стиль програмування. Функціональне програмування не означає програмування функціями. Функціональне програмування відноситься до більш строгого стилю, де функції отримують на вхід певні дані, обробляють лише ці дані і повертають результат. У функціональному програмуванні функції уникають двох властивостей, наявних в інших 25 мовах:

- ☐ **мутація даних** (зміни даних на місці, а не повернення значення)
- ☐ **спирання на стан** (додатковий контекст, який не передбачений в якості вхідних даних, наприклад глобальні змінні). Це є «фальшиві друзі», оскільки вони суперечать істотному поняттю функції, яке полягає в інкапсуляції даних та алгоритму. Коли функція залежить від стану або мутації, вона працює поза цими межами.
- ☐ Функціональне програмування є складним з точки зору побудови правильного алгоритму, але водночас це заклик структурувати програму, що окуповується під час тестування і відлагодження

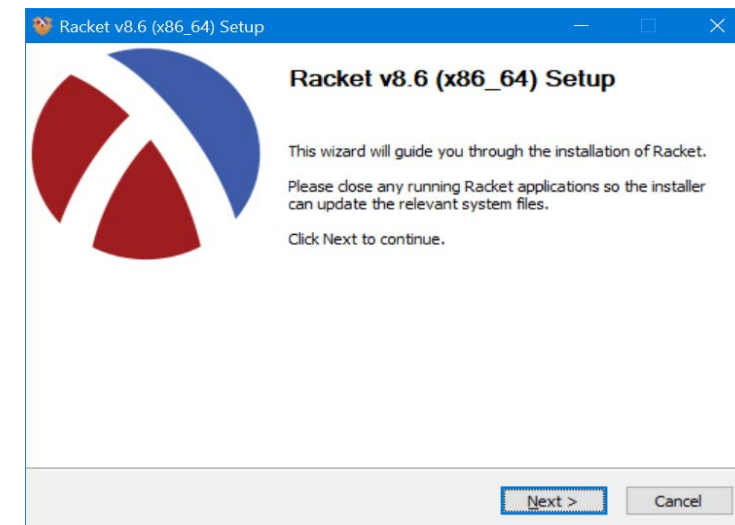
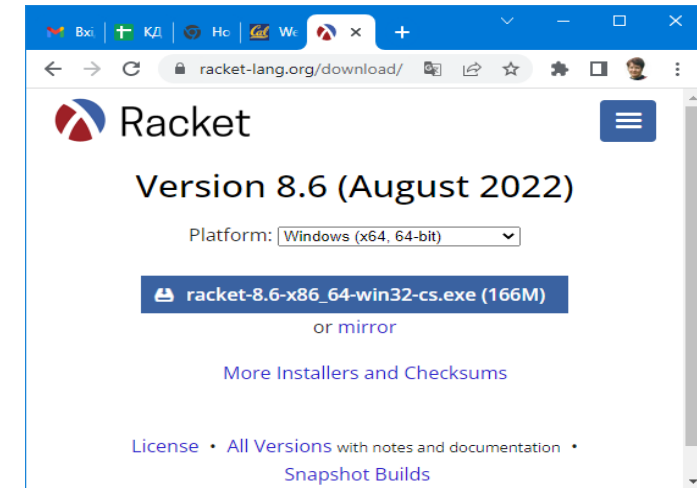
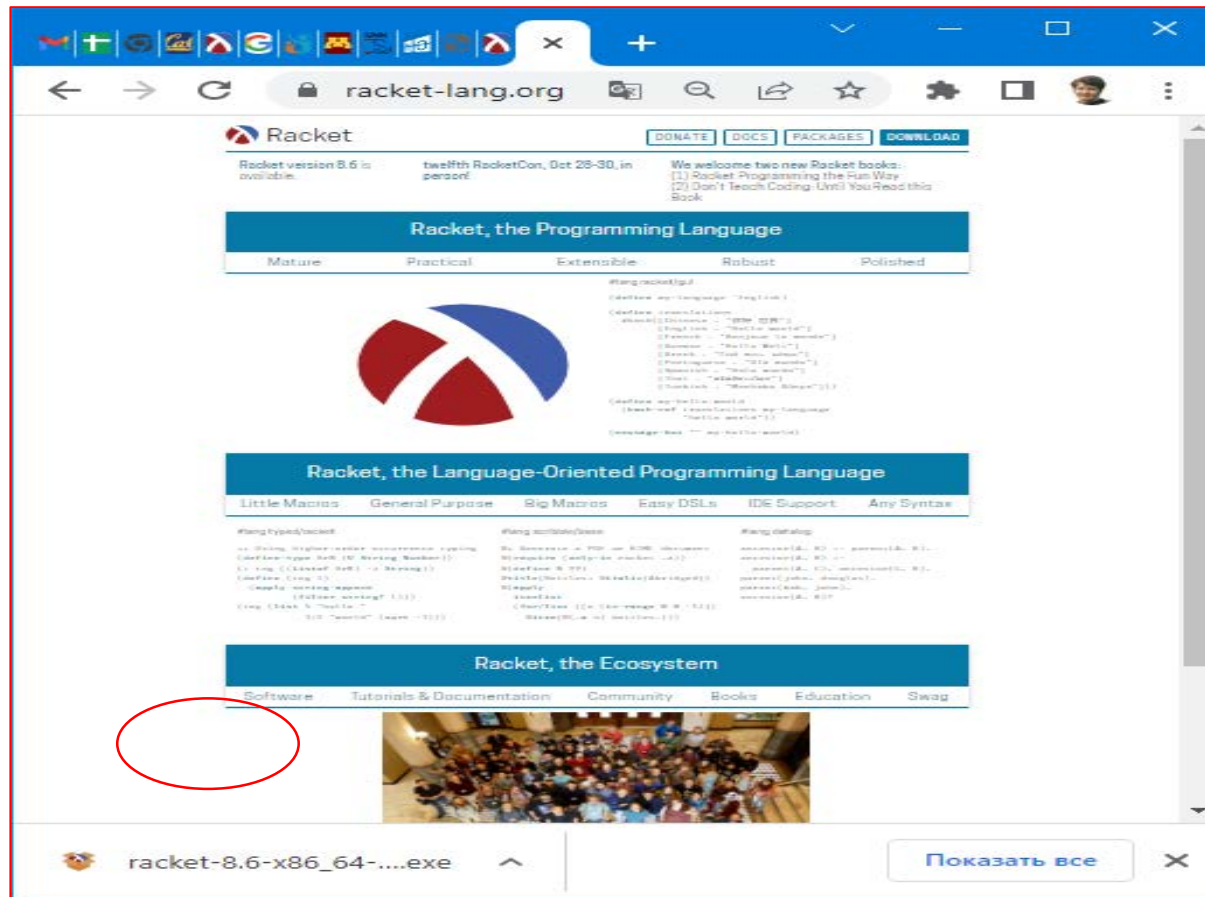
Середовище розробки Dr.Racket

4. **Бібліотеки і документація.** Дослідження ведуться протягом багатьох років, а команда розробників ядра Racket – це в більшості Computer Science-професори: бібліотеки і документація Racket продумані і наповнені відповідними прикладами використання.
5. **Dr. Racket.** Racket надає крім командного рядка добре продумане візуальне інтерактивне крос-платформне середовище розробки.
6. **X-вирази.** X-вирази – це спеціальна структура даних, яку Lisp використовує для представлення HTML і інших XML-даних, що використовуються у веб-програмуванні.
7. **Scribble.** Scribble – діалект Racket, який перевертає звичайний зв'язок між звичайним текстом і кодом: замість того, щоб вбудовувати текстові рядки в код, документ Scribble складається з виразів програмного коду, вбудованих в звичайний текст.
8. **Синтаксичні перетворення.** Синтаксичні перетворення в Racket – це еквівалент **макросам**. Макроси в Common Lisp – це функції, які працюють під час компіляції, приймаючи символи на вході і підставляють їх в шаблон для утворення нового коду. Синтаксичні перетворення додають ще один шар можливостей виразів.
9. **Створення нової мови програмування.** Крім виразів, списків і синтаксичних перетворень, Racket надає велику кількість семантичної гнучкості. Ви можете використовувати ці можливості для створення свого діалекту Racket, або створити абсолютно іншу мову програмування з власними правилами. Ви маєте змогу використовувати цю мову в середовищі Dr.Racket для власних проектів. Такі спеціалізовані мови часом називають предметно-орієнтованими мовами (**domain-specific languages, DSL**). Scribble є DSL-базованою на мові Racket.
10. **Можливості для участі.** Оскільки Racket є проектом з відкритим кодом, то кожен має змогу взяти участь у вдосконаленні цього проекту

Інтерфейс середовища Dr.Racket

Для початку необхідно завантажити файл для інсталяції за посиланням:

<https://racket-lang.org/> або <http://racket-lang.org/download/>
<https://soft.mydiv.net/win/files-Racket.html>



← → ↺ soft.mydiv.net/win/files-Racket.html 🔍 📄 ☆ ⚙️ 🖱️ 👤 ⋮

mydiv Новости и Статьи Программы Игры Драйверы Техника Форум 🔍 👤 Login

Windows ▾ ☰ Разработка > Языки программирования/компиляторы > Racket

Обзор, официальные файлы

Racket

📄 Скачать Версия для 🖱️ Windows

Обзор Отзывы 🗨️ Вопросы Обсуждение (1) Ссылки (2)

- Инструкция по скачиванию
- Инструкция по установке программ для новичков

📄 Наши новости в Telegram 🗨️ Группа технической помощи в Telegram

Предположительно у вас система Windows 10, разрядность: 64 Bits ? Подробнее

Выберите файл для скачивания:

Версия для 64-битных систем Windows

Скачать Racket (64 бита) с нашего сайта

Этот файл мы отметили как основной. Если вы не знаете что скачивать, то скачивайте его.

📄 Скачать 📄 QR Код 166.04 МБ ✓ Проверено антивирусами 🍎 0 🍌 0 Подробнее ▾

Версия для 32-битных систем Windows

Скачать Racket (32 бита) с нашего сайта

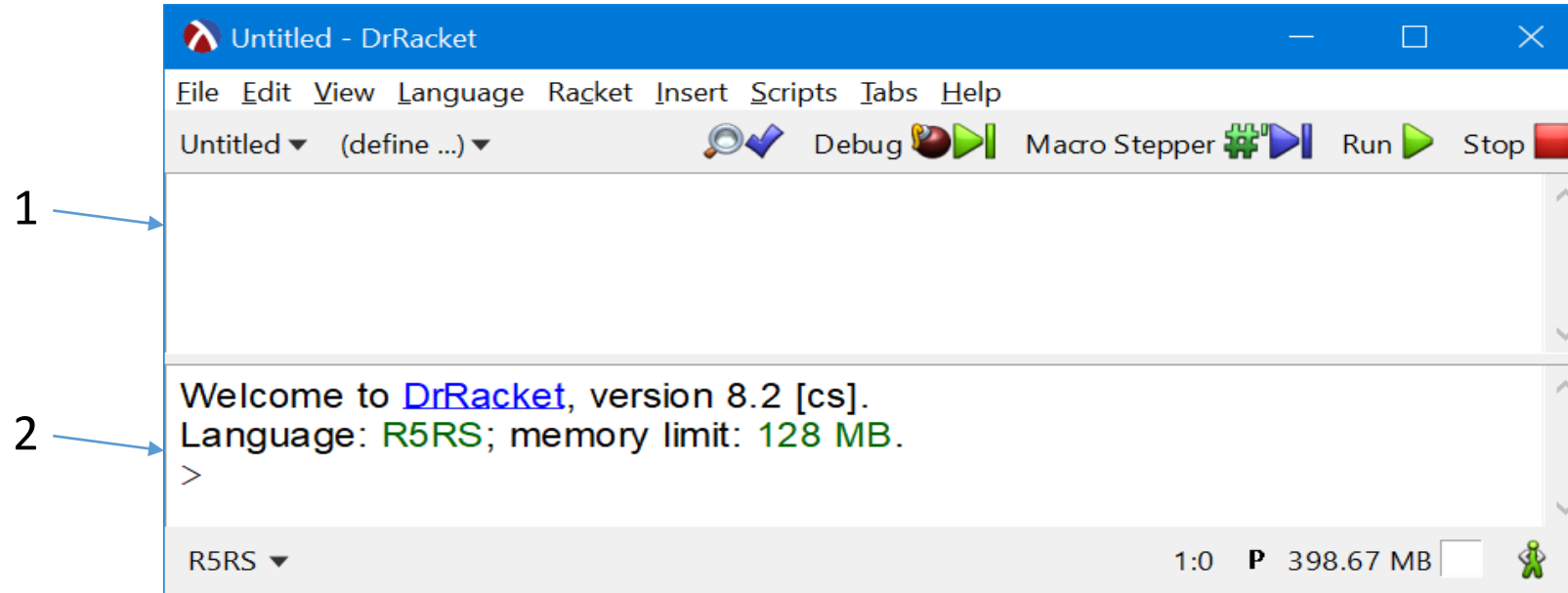
📄 Скачать 📄 QR Код 81.82 МБ ✓ Проверено антивирусами 🍎 0 🍌 0 Подробнее ▾

Внимание! Мы стараемся тщательно проверять все программы, но не гарантируем абсолютную безопасность скачиваемых файлов. Администрация сайта не несет ответственности за содержание файлов, программ и возможный вред от их использования.

<https://soft.mydiv.net/win/files-Racket.html>

Інтерфейс середовища Dr.Racket

Оскільки в середовищі Dr.Racket підтримується ряд діалектів та стандартів, оберемо класичний стандарт Scheme R5RS. Для цього потрібно обрати меню Мова-> Обрати мову ->Other languages -> R5RS.



Головне вікно середовища розробки поділене на дві частини:

1 – область для вводу тексту програми;

2 – область командного рядка.

В першій частині зручно писати текст програми і всі виклики, в другій – виклики функцій.

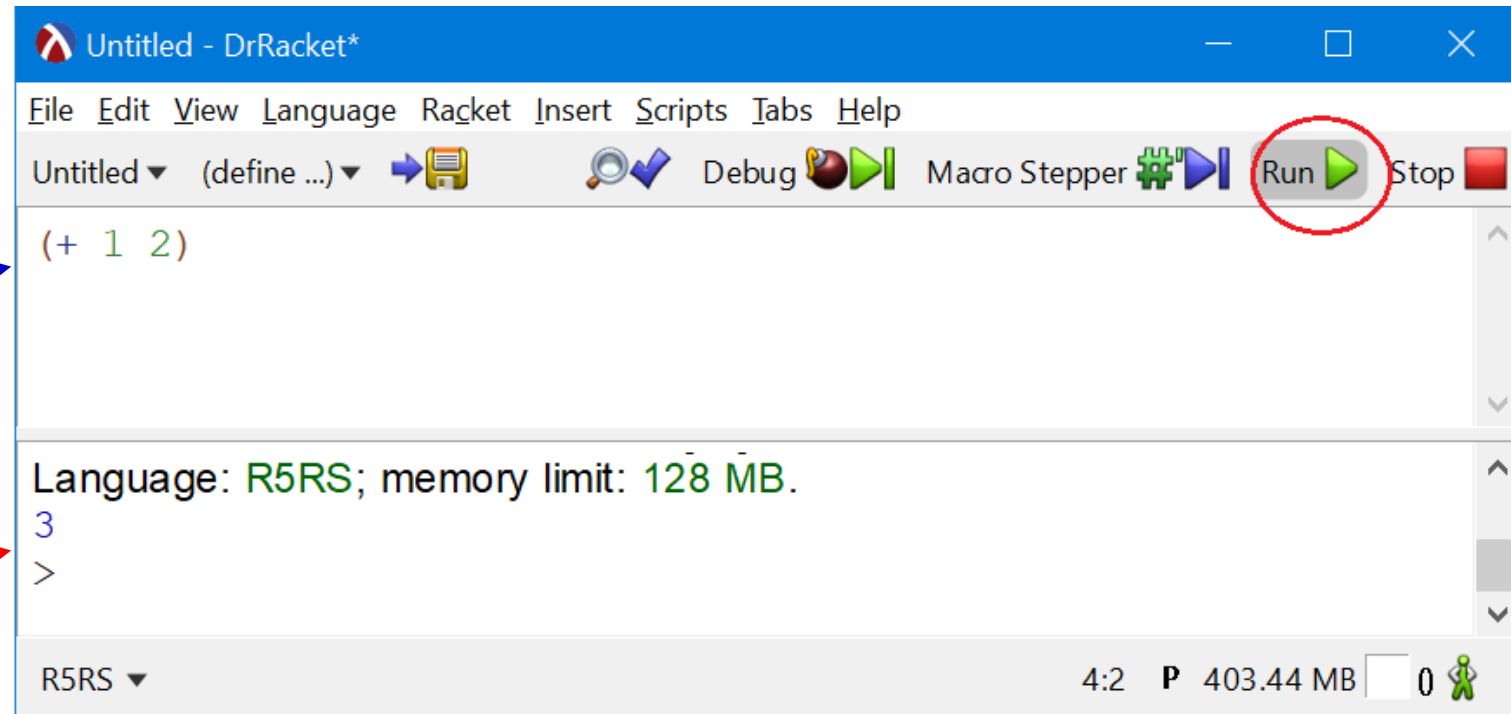
Інтерфейс середовища Dr.Racket

Приклад 1.

Обчислимо суму $1 + 2$. Мовою Scheme потрібно написати `(+ 1 2)`.

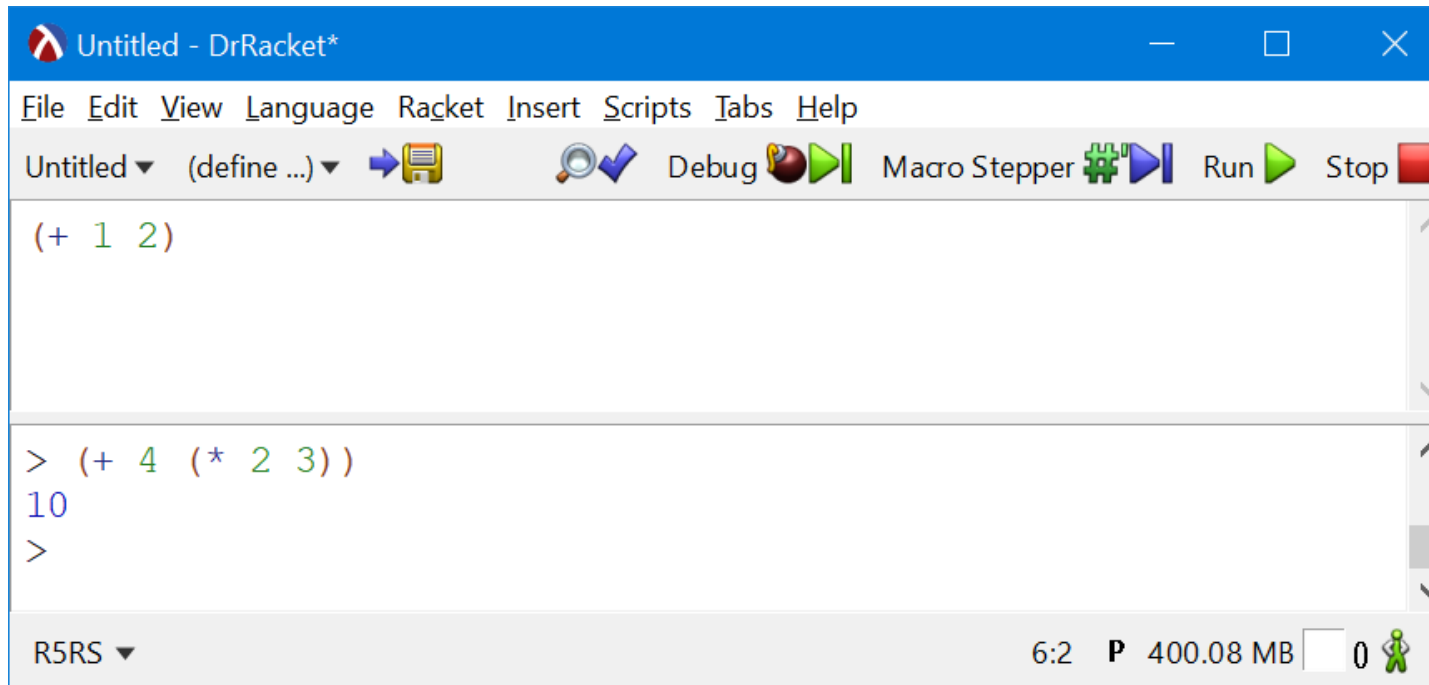
Після цього запустимо програму на виконання клавішою F5 або кнопкою «Run».

Результат виконання програми відобразиться в другій половині екрану

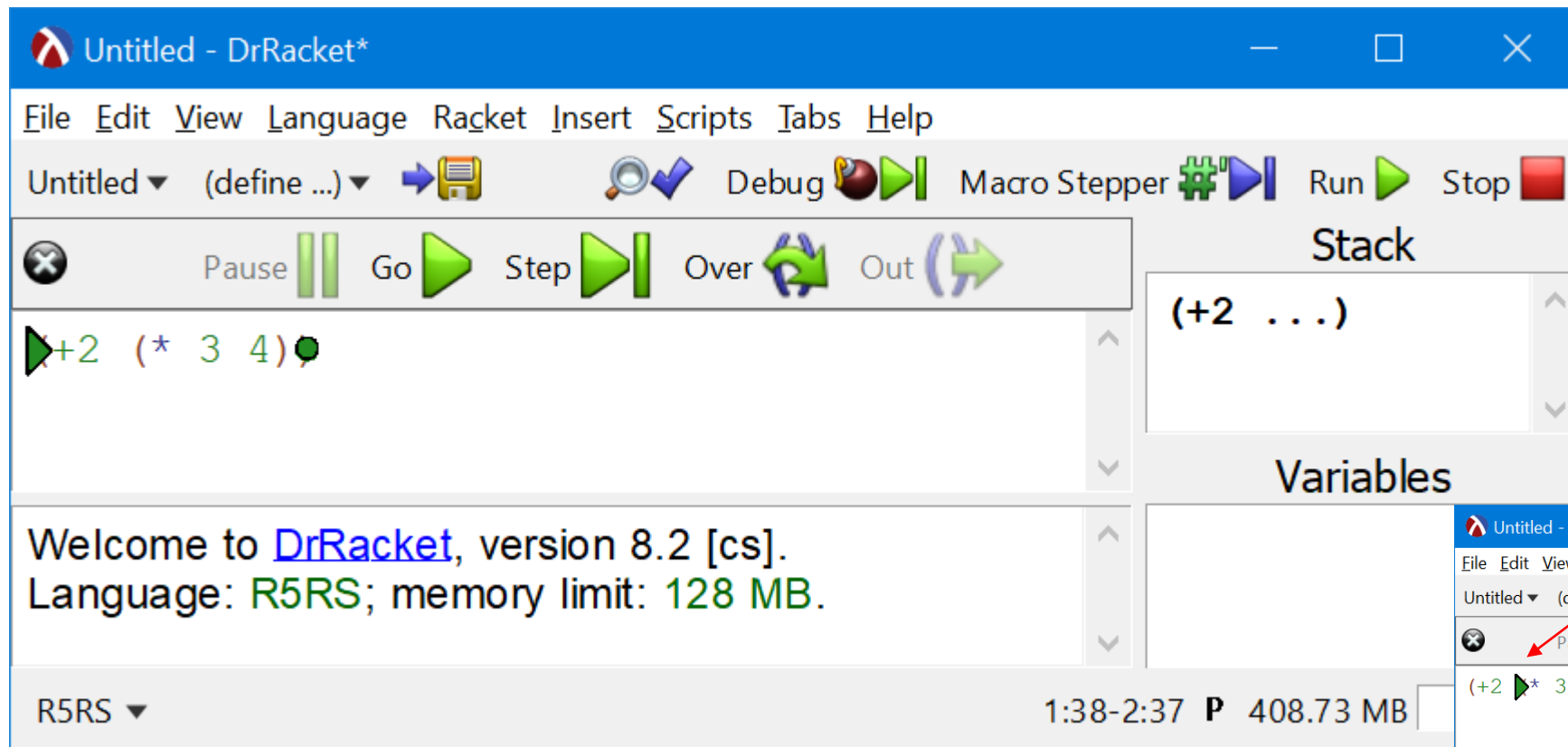


Інтерфейс середовища Dr.Racket

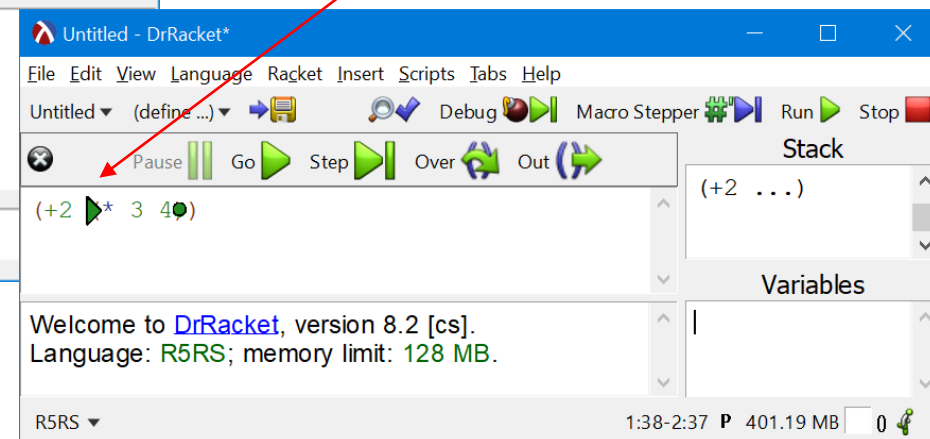
Є можливість виконувати команди безпосередньо в командному рядку в режимі інтерпретатора: після введеного тексту програми потрібно натиснути клавішу Enter.



Приклад 2. Обчислимо такий вираз: $(+ 2 (* 3 4))$. Замість кнопки «Run» натиснемо кнопку «**Debug**»

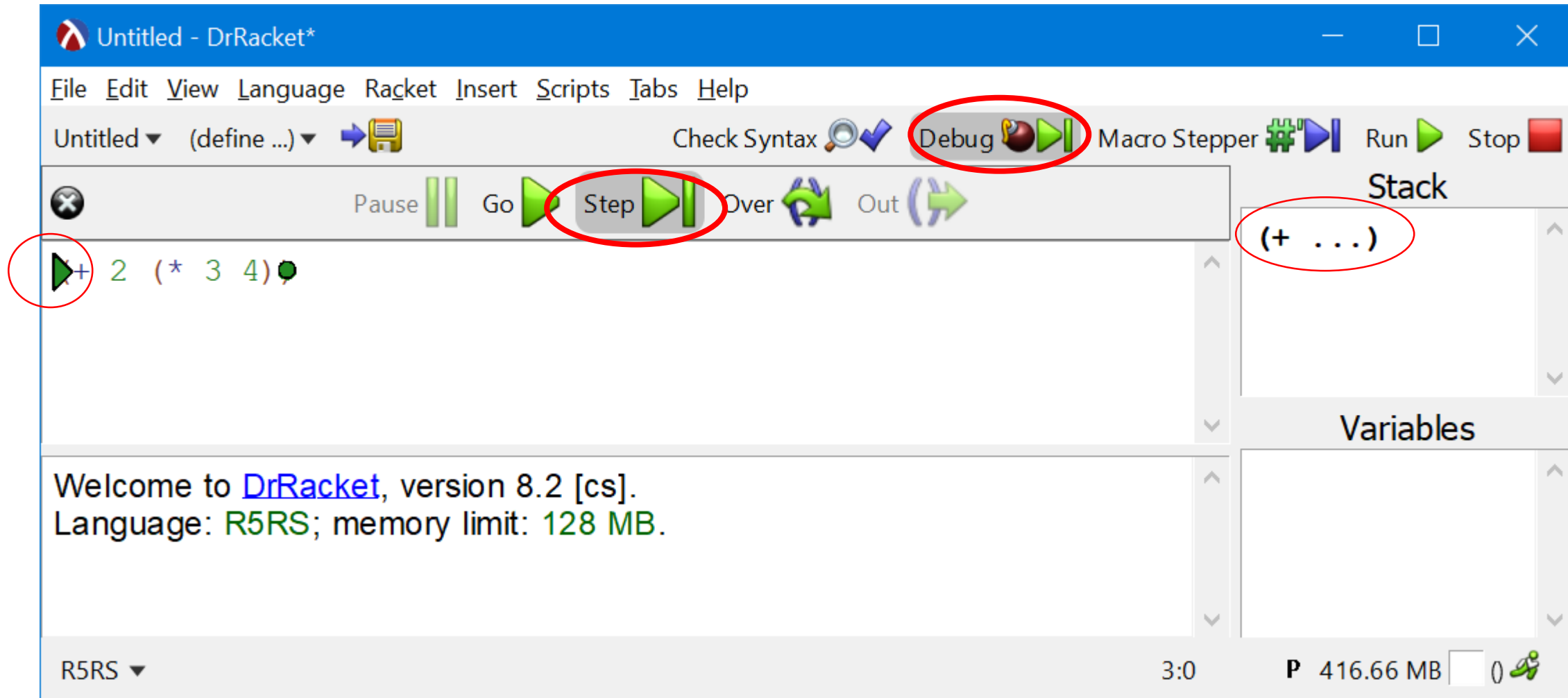


Спочатку в стеку є наявна процедура додавання. Курсор налагодження вказує на цю процедуру.
Натиснемо кнопку «**Step**».
В стеку з'явилась процедура множення

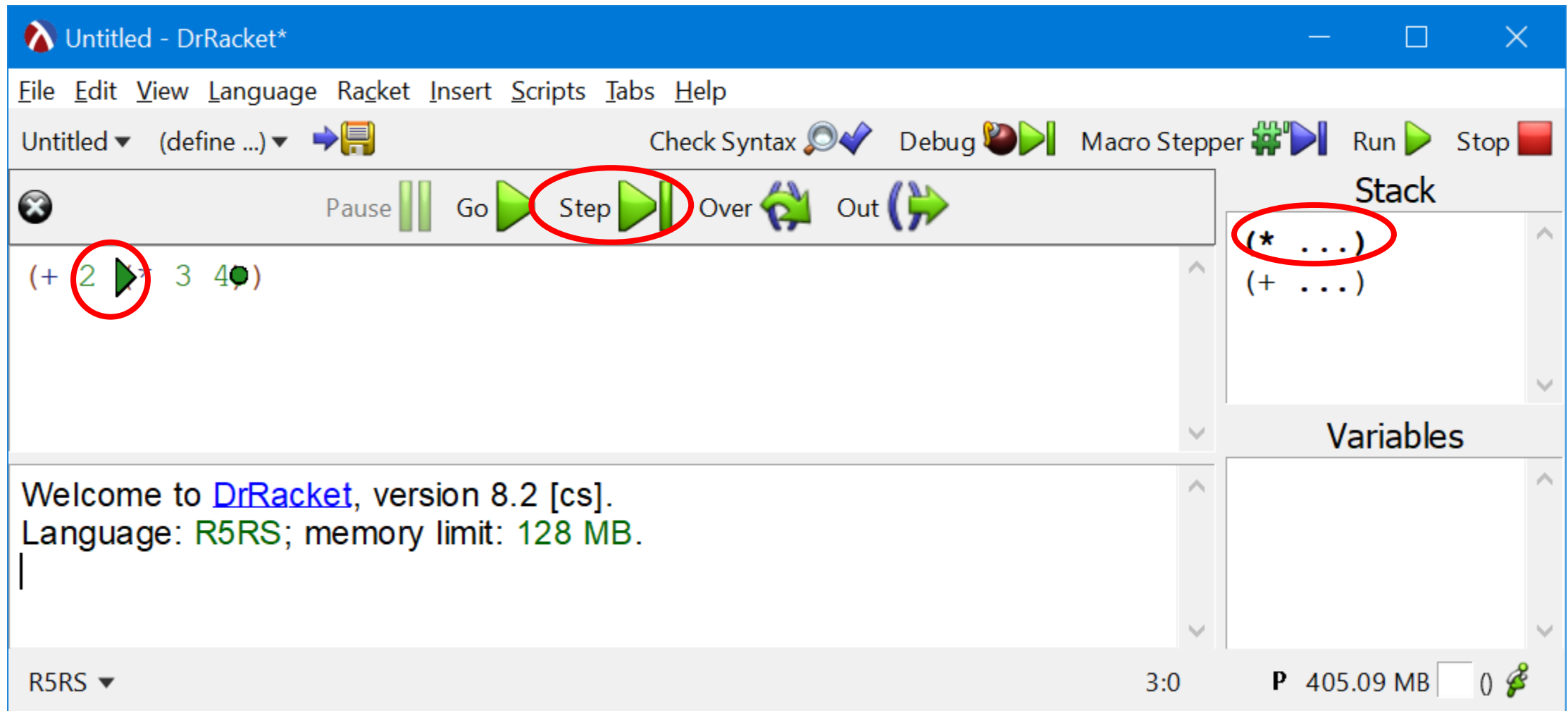


Процес налагодження коду в середовищі Dr.Racket

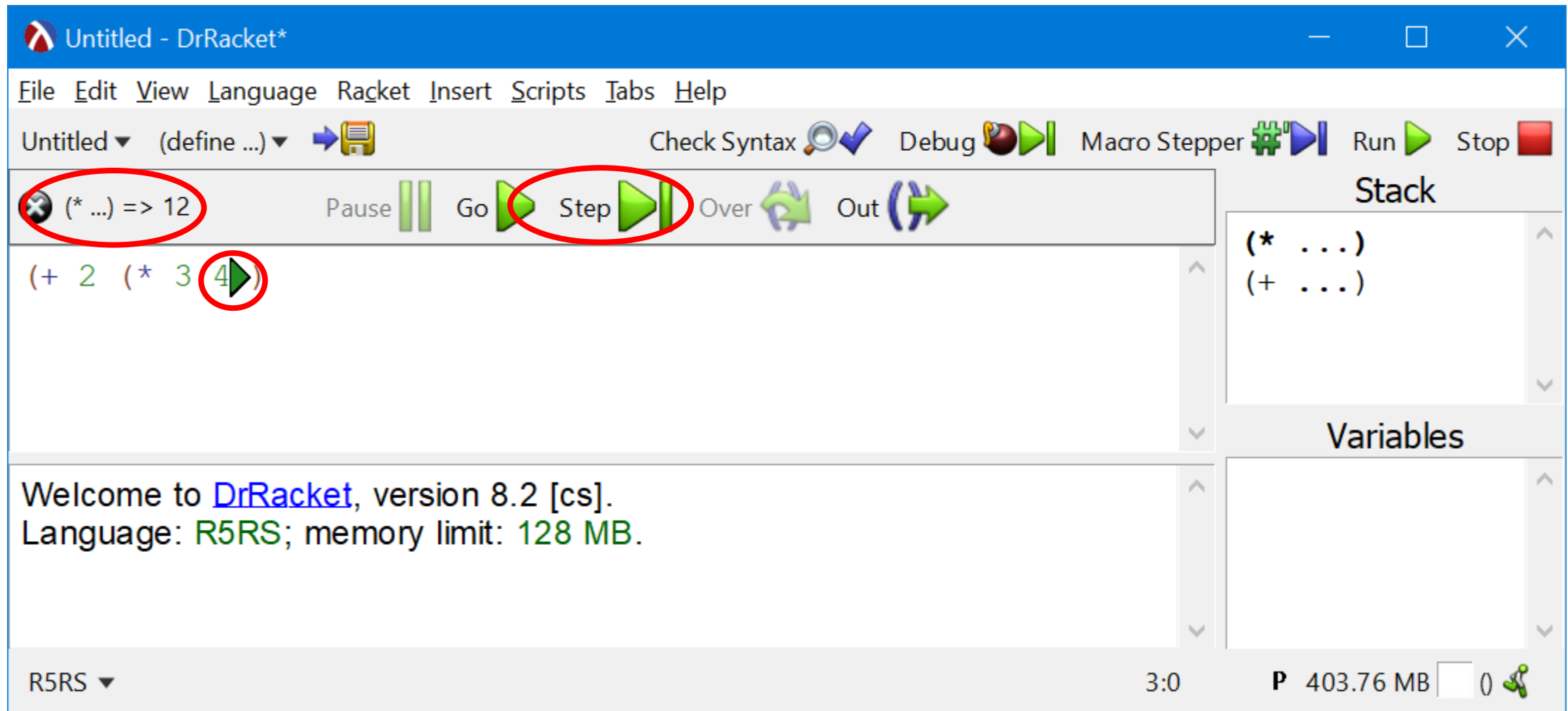
Приклад 2. Обчислимо такий вираз: $(+ 2 (* 3 4))$. Замість кнопки «Run» натиснемо кнопку «**Debug**»



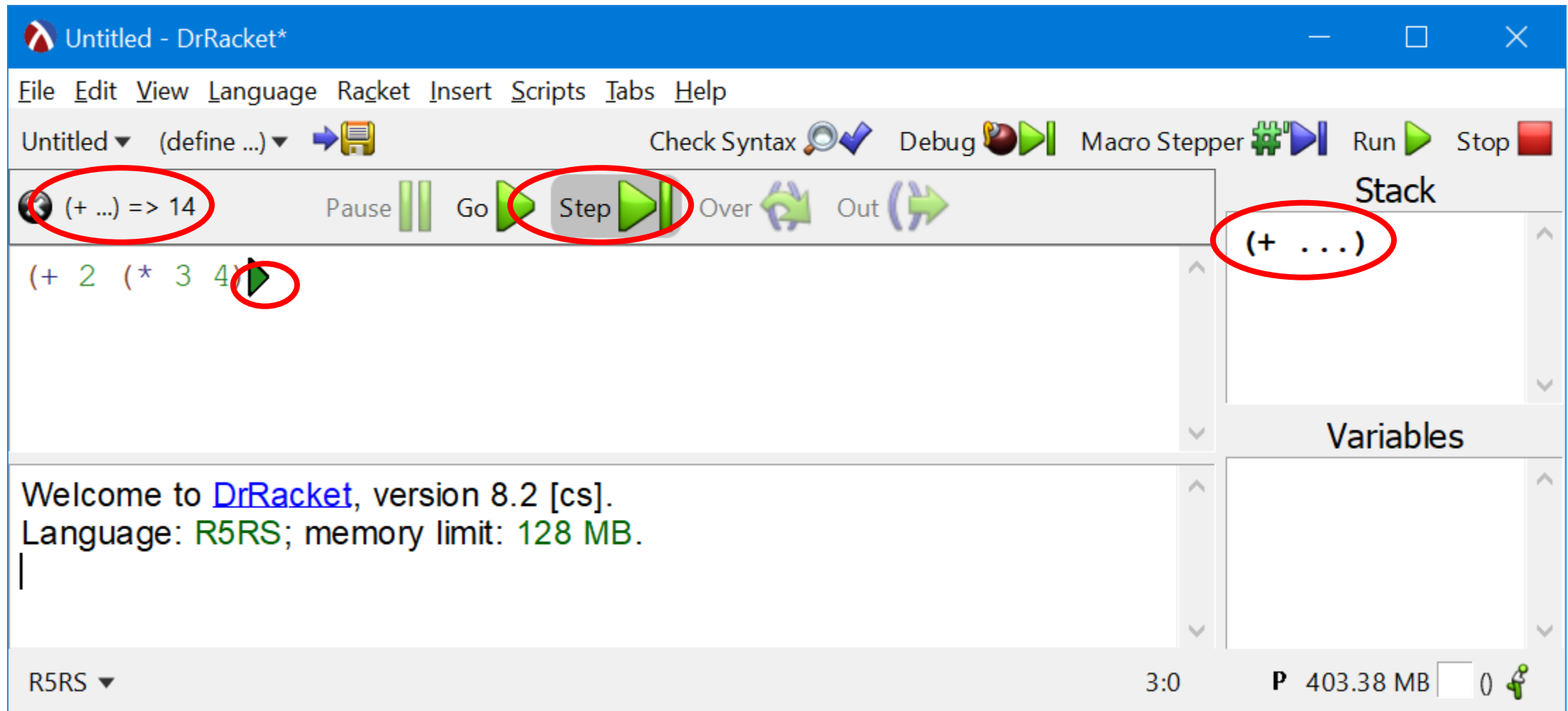
Процес налагодження коду в середовищі Dr.Racket

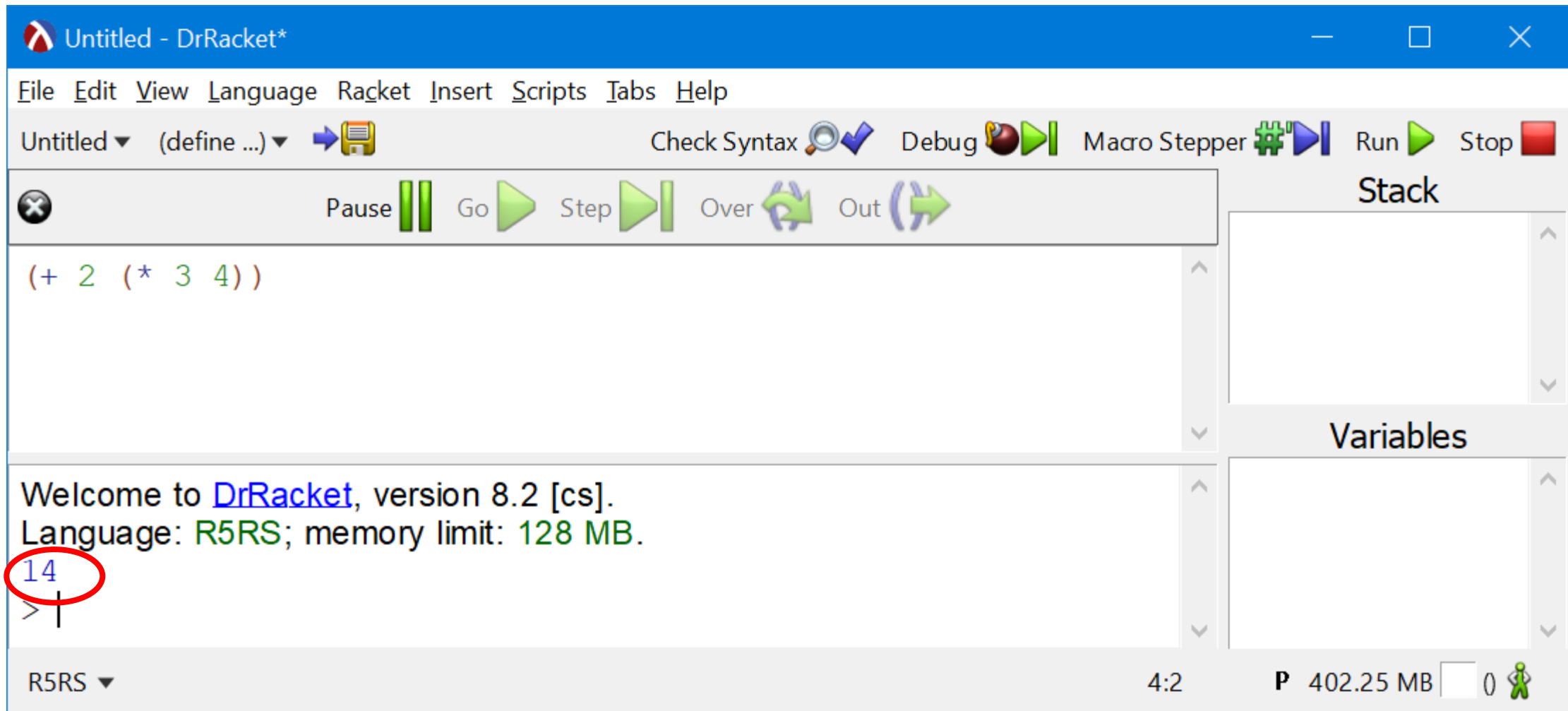


Процес налагодження коду в середовищі Dr.Racket



Процес налагодження коду в середовищі Dr.Racket





Література з програмування на Scheme

1. Навчальні матеріали Ковалюк Т.В. <https://github.com/tkovalyuk/>
2. Стандарт Scheme, версія 6.
http://www.r6rs.org/final/html/r6rs/r6rs-Z-H-2.html#node_toc_start
3. Стандарт Scheme, версія 7. Revised7 Report on the Algorithmic Language Scheme.
<http://www.larcenists.org/Documentation/Documentation0.98/r7rs.pdf>
4. Абельсон Гарольд, Сассман Джеральд Джей, Сассман Джули. Структура и интерпретация компьютерных программ. <https://www.twirpx.com/file/81061/>
<https://library.kre.dp.ua/Books/2-4%20kurs/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%B8%20%D1%96%20%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B8%20%D0%BE%D0%B1%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D1%8C/%D0%94%D0%BE%D0%B4%D0%B0%D1%82%D0%BA%D0%BE%D0%B2%D1%96%20%D0%BC%D0%B0%D1%82%D0%B5%D1%80%D1%96%D0%B0%D0%BB%D0%B8/%D0%90%D0%B1%D0%B5%D0%BB%D1%8C%D1%81%D0%BE%D0%BD%2C%20%D0%A1%D0%B0%D1%81%D1%81%D0%BC%D0%B0%D0%BD%20-%20%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0%20%D0%B8%20%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D1%80%D0%B5%D1%82%D0%B0%D1%86%D0%B8%D1%8F%20%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D1%8B%D1%85%20%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC.pdf>
5. R. Kent Dybvig. The Scheme Programming Language. <https://www.scheme.com/tspl4/>
6. Кристиан Кеннек. Интерпретация Лиспа и Scheme. <http://blog.ilammy.net/lisp/index.html>
7. Майлингова О. Л., Манжелей С. Г., Соловская Л. Б. Прототипирование программ на языке Scheme.
<https://docplayer.ru/71381060-Prototipirovanie-programm-na-yazyke-scheme-metodicheskoe-posobie-po-praktikumu.html>

Література з програмування на Scheme

Programming in MacScheme	Eisenberg, A. Cambridge, MA: The MIT Press, pps. 387.
Structure and Interpretation of Computer Programs	Abelson, H. and Sussman, J. (1985) New York: McGraw-Hill
The Scheme Programming Language	Dybvig (1987) Englewood Cliffs, NJ: Prentice-Hall
The Little LISPer	Friedman, D. and Felleisen, M. (1988) Cambridge, MA: MIT Press
An Introduction to Scheme	Smith, (1988) Prentice-Hall
PC Scheme User's Software	Texas Instruments (1988) MIT Press, ISBN: 0-262-70040-9
Scheme and the Art of Computer Programming	Springer and Friedman (1989) MIT Press and McGraw-Hill
PC Scheme: User's Guide & Language Reference Manual	Eisenberg, M. (1990) Cambridge, MA: The MIT Press, pps. 228. ISBN: 0-262-70040-9.
Programming in Scheme	Eisenberg, M. (1990) Cambridge, MA: The MIT Press, pps. 304. ISBN: 0-262-55017-2.
The Schemer's Guide	Ferguson, I. and Martin, E. (1990) Fort Lauderdale, FL: Schemers Inc
Simply Scheme: Introducing Computer Science	Harvey, B. and Wright, M., (1994) Cambridge, MA: MIT Press
Scheme Programming Language, The: ANSI Scheme	Dybvig, R. (1996) Prentice Hall, pps. 272. ISBN 0134546466

Методичні матеріали з програмування мовою Scheme

№	Адреса	Опис
1	http://www.racket-lang.org/	DrRacket
2	http://inst.eecs.berkeley.edu/	STk-ucb
3	http://www.gnu.org/software/mit-scheme/	MIT-GNU Scheme
4	http://www.htdp.org/	Felleisen M. How to Design Programs. An Introduction to Programming and Computing / Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi. – Cambridge, Massachusetts : The MIT Press, 2003.
5	https://web.mit.edu/6.001/6.037/sicp.pdf	Abelson H. Structure and Interpretation of Computer Programs / Harold Abelson, Gerald Jay Sussman, Julie Sussman. – Cambridge, Massachusetts : The MIT Press, 1996.
6	http://www.scheme.com/tspl3/	Dybvig R. K. The Scheme Programming Language, 3 rd Edition / R. Kent Dybvig. – Cambridge, Massachusetts : The MIT Press, 2003.
7	http://www.cs.hut.fi/Studies/T-93.210/schemetutorial/	Scheme Tutorial [Electronic resource] / [Catharina Candolin]
8	https://doc.lagout.org/programmation/Lisp/Scheme/Teach%20Yourself%20Scheme%20in%20Fixnum%20Days%20-%20Dorai%20Sitaram.pdf	Teach Yourself Scheme in Fixnum Days [Electronic resource]

Методичні матеріали з програмування мовою Scheme в університетах США

№	Адреса	Опис
9	https://ocw.mit.edu/courses/6-001-structure-and-interpretation-of-computerprograms-spring-2005/	Курс «Structure and Interpretation of Computer Programs», Массачусетський технологічний інститут, м. Кембридж, штат Массачусетс, США
10	https://www.cs.unm.edu/~williams/cs357/springerfriedman.pdf https://www.cs.trinity.edu/~bmunsing/csci2322-fall/index.html	Курс «Principles of Functional Languages», Трініті університет, м. Сан-Антоніо, штат Техас, США
11	http://www.classes.cs.uchicago.edu/archive/2009/fall/15100-1/	Курс «Introduction to Computer Science I», Чиказький університет, м. Чикаго, штат Іллінойс, США
12	http://zoo.cs.yale.edu/classes/cs201/	Курс «Introduction to Computer Science», Єльський університет, м. Нью-Хейвен, штат Коннектикут, США

Література з програмування на Haskell, Lisp, Common Lisp, ML

Інші мови функціонального програмування

1. Антон Холомьёв. Учебник по Haskell.
<https://docplayer.ru/25937980-Uchebnik-po-haskell-anton-holomyov.html>
2. John Harrison. Введение в функциональное программирование.
<https://nsu.ru/xmlui/bitstream/handle/nsu/8874/Harrison.pdf;jsessionid=7BDBFCF0EA05BFD026052B868E6DAEDF?sequence=1>
3. Лидия Городняя. Введение в программирование на языке Лисп.
http://window.edu.ru/resource/684/41684/files/prog_lisp.pdf
4. Практический Common Lisp. <http://lisper.ru/pcl/pcl.pdf>



Дякую за увагу

Доц. кафедри ПСТ к.т.н. Ковалюк Т.В.
tkovalyuk@ukr.net