Тема роботи: Розроблення найпростіших програм мовою Scheme.

Мета роботи: Вивчити базові конструкції мови Scheme та навчитись розробляти найпростіші програми у середовищі Dr.Racket.

Теоретичні відомості

1. Історія розвитку функційних мов програмування

Декларативні мови, на відміну від імперативних мов, описують проблему, а не визначають рішення. Функціональне і логічне програмування є підкатегоріями декларативного програмування. Прикладами декларативних мов програмування є: Candle, Lustre, MetaPost, Modelica, Prolog, Oz, RDQL, SPARQL, SQL тощо.

Функціональні мови програмування визначають програми і підпрограми як математичні функції. Серед цих мов існує ряд не лише строго функціональних мов, але і з імперативними властивостями. Не дивно, що багато з цих мов пов'язані з інструментами математичних обчислень. Функціональні мови включають в себе [1]:

Таблиця 1.

Строго- і нестрого-функціональні мови програмування

Строго функціональні мови	Нестрого функціональні мови		
Charity	APL	Hop	Nemerle
Clean	ATS	J	Opal
Coq (Gallina)	CAL	Java (з версії 8)	OPS5
Curry	C++ (з C++11)	JavaScript	Poplog
Elm	C#	**Joy**	Python
Frege	Candle	Julia	Q
Haskell	Curl	**Lisp**	**R**
Hope	Erlang	Mathematica	Ruby
Mercury	Elixir	ML	REFAL
Miranda	F#	Standard ML	Rust
Idris	**FPr**	Alice	Scala
SequenceL	Groovy	OCaml	Spreadsheets

Крім цього, сім’я Lisp включає такі діалекти: **Clojure**, **Common Lisp**, **Dylan**, **Emacs Lisp**, Little b, Logo, **Scheme**, **Racket** (також відомий як PLT Scheme), **Tea** (Тут, і в табл.1 жирним шрифтом виділено мови програмування, які базуються на списках, тобто в їх основі є спискові структури даних).

Пояснимо відмінність декларативного підходу та імперативного. Якщо потрібно отримати дані з таблиці за певним критерієм, можна використати структуровану мову запитів SQL і написати щось на зразок

Select * from MyTable where year > 2015

Таким чином ми отримаємо шуканий результат в декларативному стилі. Якщо потрібно реалізувати подібний запит в імперативному стилі, потрібно забезпечити прохід курсору бази даних по всіх записах таблиці, забезпечити інкремент значення поточного рядка тощо. На мові програмування C++ це виглядатиме вже як:

for (int i = 0; i < tableLength; i++) { if (tableRow[i].year > 2015) { … } }

Тобто в першому випадку ставиться питання «що ми хочемо отримати?», а в другому – «як потрібно це отримати?».

Першою функційною мовою програмування була мова створена американським вченим Джоном Маккарті – Lisp (List Processing). Областю її застосування стали символьні обчислення, які є типовими для задач штучного інтелекту (Artificial Intelligence, AI).

В основу мови покладено строгий математичний апарат:

- лямбда-обчислення Чьорча;
- алгебра спискових структур;
- теорія рекурсивних функцій.

Scheme – один з двох найбільш популярних в наші дні діалектів мови Lisp (другий популярний діалект – це Common Lisp). Автори мови Scheme – Гай Стіл (Guy L. Steele) і Джеральд Сассмен (Gerald Jay Sussman) з Массачусетського Технологічного Інституту – створили його в середині 1970-х років. Мова програмування **Scheme** визначається двома стандартами: стандартом де-юре в редації IEEE, та стандартом де-факто. Поточна версія описання стандарту де-факто має назву «Revised 5 Report on the Algorithmic Language Scheme» (R5RS). 28 серпня 2007 року було затверджено наступну редакцію: R6RS. 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.

Common Lisp – стандарт ANSI, найбільш відома спроба стандартизувати функційні мови. Нажаль, мова виявилася надто складною. ЇЇ опис містить більше 1000 сторінок. Доступні як комерційні (Allegro, Harlequin, Corman), так і безплатні (GnuCL, CLISP, CMUCL) реалізації.

Erlang – мова розроблена компанією Ericsson, яка характеризується виразним синтаксисом і базується на співставленні зі зразком і дечим нагадує

Prolog. Специфіка мови - орієнтація на паралельне програмування систем реального часу. Мова включає засоби для опису взаємодіючих процесів.

На сьогодні все більш набуває популярність динамічна мова програмування **Clojure**, яка базується на віртуальній машині Java Virtual Machine. Мова створена як мова загального призначення, і поєднує доступність та інтерактивну розробку скриптових мов з ефективною та надійною інфраструктурою багатопотокового програмування. Clojure є діалектом Lisp, а тому запозичує у Lisp філософію «програмний код як дані» і потужну систему макросів. Clojure є переважно функціональною мовою програмування, і має багатий набір незмінних, стійких структур даних.

«Родзинки» від використання мови програмування Lisp легко найти. Однак проблемою Lisp є те, що це має сенс лише для досвідчених Lisp-програмістів. Для інших, особливо для тих, які вагаються чи варто вивчати і використовувати Lisp – це просто сприймається як марнування часу. Тому довгий час мова Lisp використовувалася вузьким колом дослідників. Широке поширення мова отримала в кінці 70-х – на початку 80-х років з появою необхідної потужності обчислювальних машин і відповідного кола задач.

Функційне програмування – це нетрадиційне програмування, орієнтоване на комп’ютери архітектури відмінної від “фон-нейманівської”. Тому найефективніше ці мови можуть використовуватися на так званих Lisp-машинах. Оскільки проект створення Lisp-машин через свою вартісність не завершився позитивними результатами, то сьогодні ці мови мають вузьке спеціалізоване застосування в галузі комп’ютерних наук.

У своєму ессе «Як стати хакером» Ерік Раймонд каже "*Lisp є цінним навчанням за ... великий досвід просвітлення, який ви отримаєте, коли ви, нарешті отримаєте його. Цей досвід зробить вас кращим програмістом на все життя, навіть якщо ви більше ніколи не будете використовувати Lisp*". На жаль, Раймонд не пояснює в ессе як це поліпшить навички програмування в цілому. Ессе Раймодна не присвячене опису мови Lisp, однак він описує Lisp як секретну зброю, і пояснює, що мови програмування розрізняються за потужністю.

2. 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.

Не дивно, що багато відомих застосувань AI в цих областях були написані мовою **Lisp**:

- **Macsyma** – перша велика система комп'ютерної алгебри.
- **ACL2** – широко використовувана 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. Також використовується кількома іншими великими телескопами.
- **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, яке перевіряє транзакції за кредитними картками.

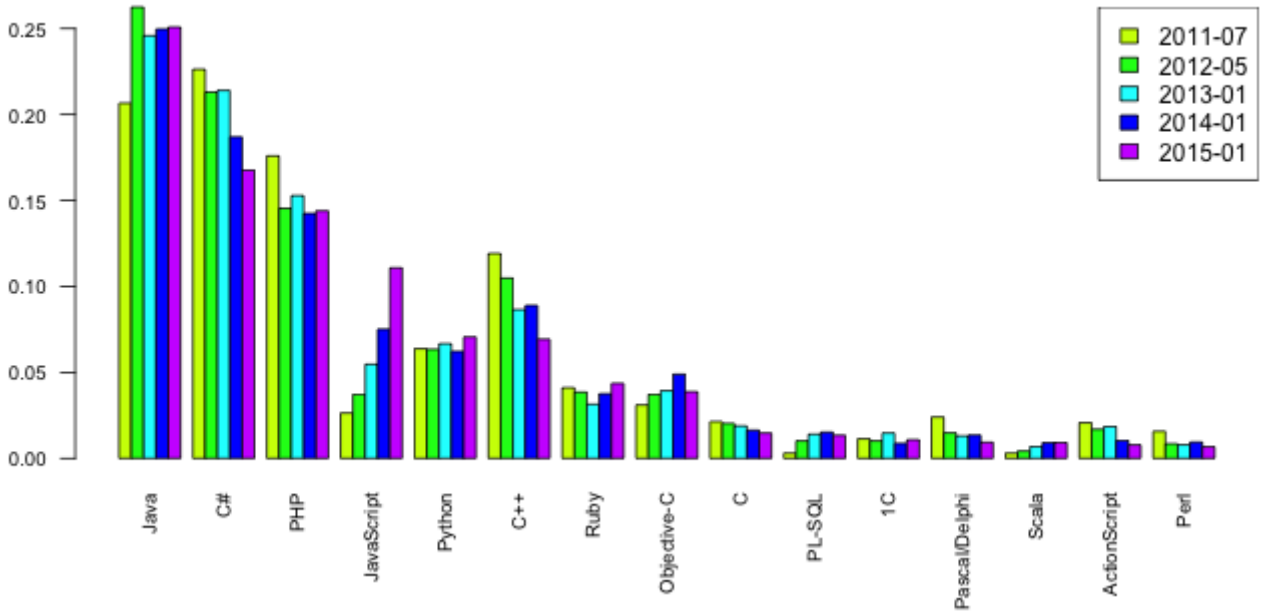

Рис. 1. Рейтинг мов програмування (січень 2015)

Поряд з тим є тисячі додатків, які написані на Lisp. Більшість з них потребують спеціальних можливостей в області обробки символів. Lisp дає змогу створювати представлення символьних даних і програм і може

реалізовувати всі види машинної роботи з цими виразами (математичні формули, логічні формули, плани, тощо).

Одне із запитань яке часто виникає у розробників-початківців на будь-якій декларативній мові програмування – «А де ця мова зараз практично використовується?». Незважаючи на те, що довкола найбільш популярними технологіями та мовами програмування є Java, C/C++, .NET, Php, Python, Ruby (рис. 1), мови базовані на декларативній парадигмі впевнено займають свою нішу у сфері розробки застосувань для штучного інтелекту.

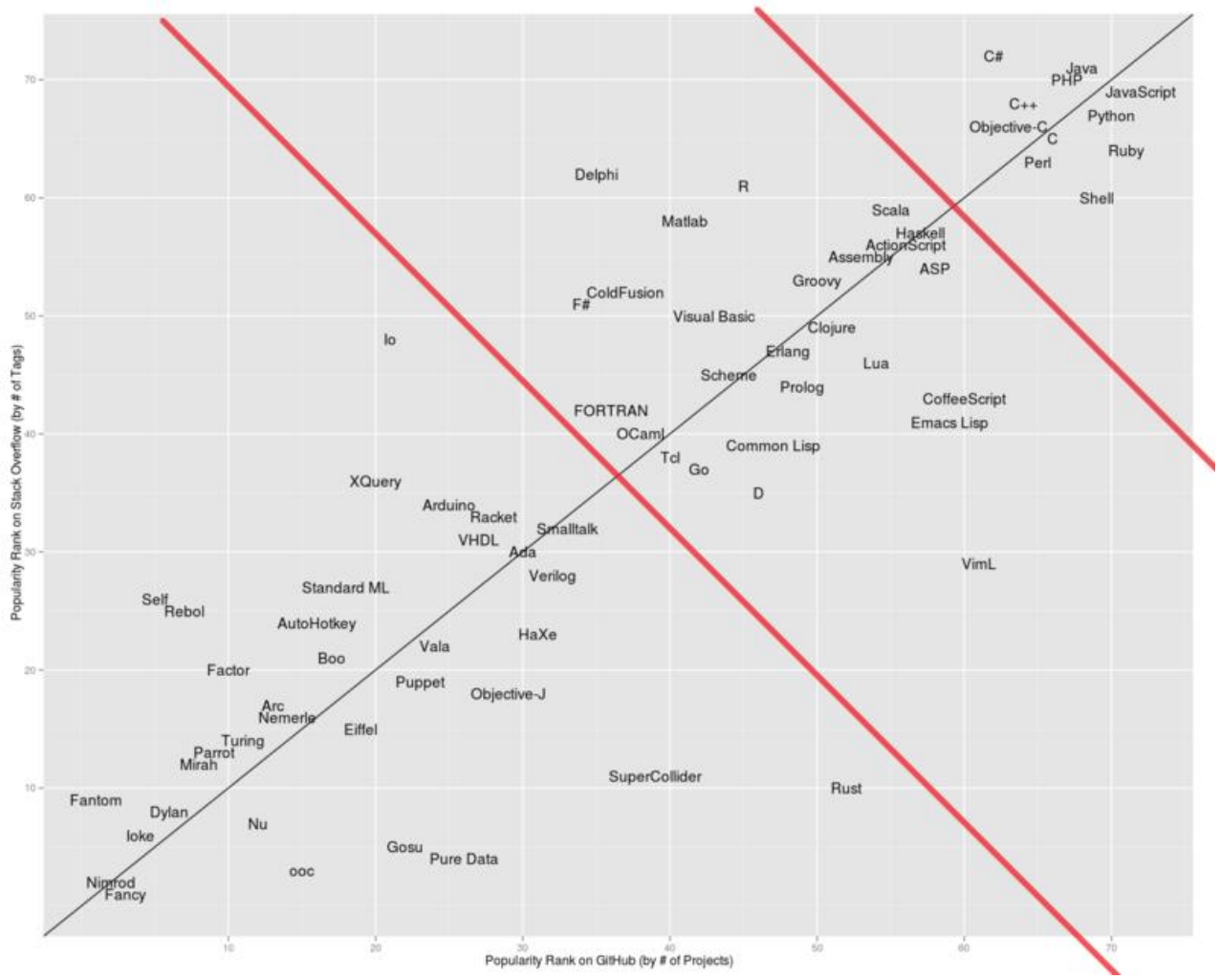

Рис.2. Співвідношення проектів на сайті GitHub до тегів на сайті StackOverflow

Зокрема, показником є кількість реальних проектів, написаних із використанням декларативних мов програмування (табл. 2, рис. 2). Останні чотири мови в табл. 2 є Lisp-подібними, отож разом створено проектів понад 35,846. Тому можна стверджувати, що Lisp на сьогодні впевнено входить в top-20 мов програмування.

Таблиця 2.

Кількість проектів на GitHub

Мова програмування	**Проектів на GitHub**
JavaScript	885,467
Java	644,711
Ruby	622,088
Python	432,533
PHP	420,913
C	218,287
Clojure	21,108
Common Lisp	6883
Scheme	5128
Racket	2727

У сучасних умовах Lisp є одним з головних інструментальних засобів систем штучного інтелекту. Серед інших прикладів застосування цієї мови можна назвати такі:

- Мова Lisp прийнята як одна з двох основних мов програмування для міністерства оборони США.
- Систему AutoCAD розроблено мовою Lisp.

3. Суть функційного програмування

Як видно з назви, основним поняттям функційного програмування є функція.

Як відомо, математичні функції виражають зв’язок між параметрами (входом) і результатом (виходом) деякого процесу. Оскільки обчислення – це також процес, який має вхід і вихід, функція є у повній мірі адекватним засобом опису обчислень. Саме цей простий принцип покладено в основу функціональної парадигми і функціонального стилю програмування. Функціональна програма є набором визначень функцій. Функції визначаються через інші функції або рекурсивно – через самих себе. У процесі виконання програми функції отримують параметри, обчислюють і повертають результат, у випадку необхідності обчислюються значення інших функцій. Програмуючи функційною мовою, програміст не повинен описувати порядок обчислень. Йому необхідно просто описати бажаний результат у вигляді системи функцій.

Функціональна програма складається з сукупності визначених функцій. Функції, в свою чергу, можуть викликати інші функції. Обчислення починається з виклику деякої функції. Строго функційне програмування не має

присвоєнь та засобів передачі керування. Повторні обчислення здійснюються за допомогою рекурсії, яка є основним засобом функціонального програмування.

Серед важливих властивостей функційного програмування є такі:

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 символьних даних, є набагато коротші, аніж написані імперативними мовами.

4. Типи даних у мові Scheme

Основу Scheme, як і Lisp складають символьні вирази, які називаються S-виразами. (Symbolic expresion). S-вирази – це або **атом**, або **список**, або **пара.** Як правило, алфавіт функційної мови складається з усіх допустимих друкованих символів (латинські букви, цифри, знаки пунктуації, спеціальні символи тощо), деколи до алфавіту не входять малі латинські букви.

Атом. Найпростіші об'єкти Scheme, з яких будуються інші структури називаються атомами. Атоми бувають двох типів – символьні і числові. Символьні атоми будуються з послідовності великих букв та/або цифр, при цьому повинен бути по крайній мірі один символ, відмінний від числа. Наприклад, SHEME, AB13, B54, .., 10.A, A..B, p…..p. Серед символьних атомів є два спеціальних, які позначають логічні значення істина – #t та неістина – #f.

Символьний атом розглядається як неділимий цілий об'єкт. До символьних атомів застосовується тільки одна операція: порівняння на тотожність. Числові атоми – звичайні числа. Наприклад, 124, -344, 4.5, 3.055E8. Числа – це константи, над якими можна виконувати усі допустимі арифметичні операції та порівняння.

Схематично організацію атомів можна представити у вигляді одиночної комірки (рис. 3).

Пара. Об'єднання двох елементів в одне ціле називається парою. Елементами пари можуть бути об'єкти допустимої у функційному програмуванні структури, тобто атом, пара, список. Пара позначається в дужках, між елементами ставиться крапка, яка розділена з обох сторін пробілами. Наприклад, пара із двох атомів **A, B** позначається як **(A . B).** Схематично пара представляється у вигляді комірки поділеної на дві частини

(рис. 3). Зауважимо, що S-вираз **(A.B)**, в якому відсутні пробіли з обох сторін крапки є списком, який складається з одного елемента – атома **A.B**. Аналогічно S-вираз **(A..B)** – список, який складається з одного атома **A..B**.

Список. Послідовність зв'язаних між собою елементів, кожен з яких є або атомом, або списком, або парою називають у функційному програмуванні списком (List). Списки позначаються в дужках, елементи списку розділяються пробілами. Наприклад, (A D (G H) W) – список з чотирьох елементів: трьох атомів та списку з двох елементів; (ADW) – список з одного елемента – символьного атома; (((((FIRST) 2) SECOND) 4) 5) – список з двох елементів: перший елемент – це багаторівневий список, другий - числовий атом. У свою чергу список ((((FIRST) 2) SECOND) 4) містить 2 елемента: багаторівневий список та числовий атом. Вкладений список (((FIRST) 2) SECOND) містить 2 елемента: перший елемент список ((FIRST) 2), а другий елемент – символьний атом SECOND. Цей список ((FIRST) 2) складається з двох елементів: списку (FIRST) та числового атома. (FIRST) – це список з одного символьного атома.

Члени списку організовані в пам'яті комп'ютера у вигляді послідовностей комірок, розділених на дві частини. У першій частині комірки вказується інформація про член списку. Це може бути безпосереднє значення, якщо інформація подається атомом, або адреса комірки, де є перший елемент вкладеного об'єкту, в протилежному випадку. У другій частині комірки вказується адреса зв'язку, тобто адреса знаходження наступного члена заданого списку. Якщо член списку є останнім, то у другій частині комірки вказується символ NIL ("ніщо"), який позначається у вигляді діагоналі другої частини останньої комірки (рис. 3). Порожній список позначається ( ).

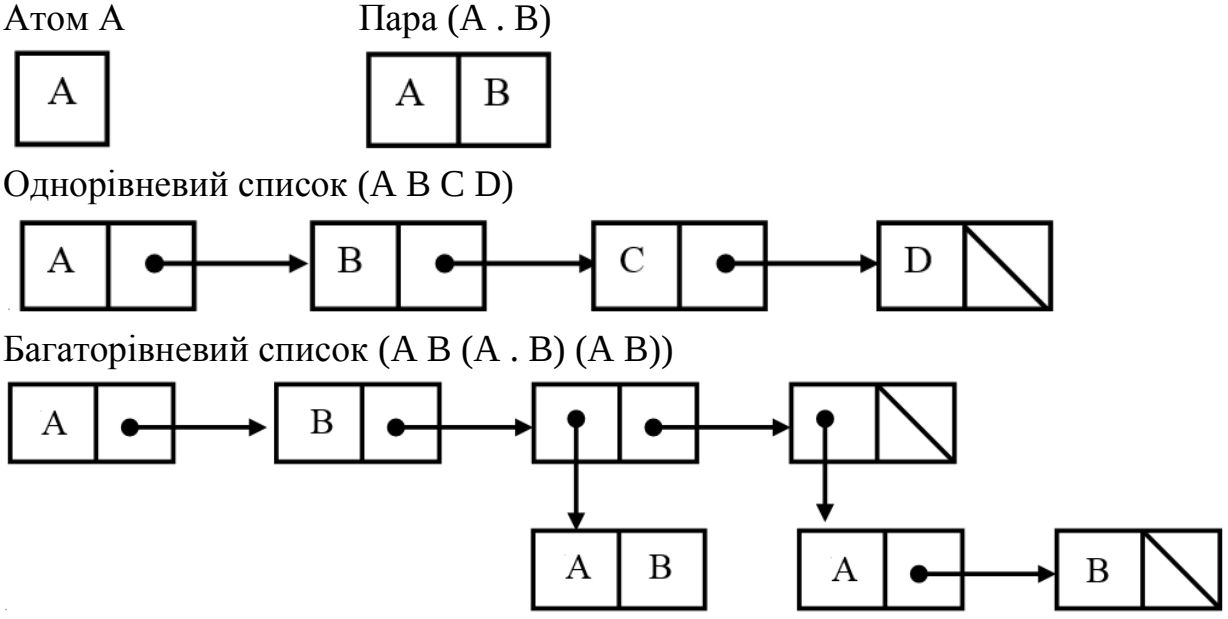

Рис. 3. Приклади схематичного зображення об'єктів

Неправильно-сформований список. Класичний список містить вкінці вказівник NIL. Натомість існує структура даних, у якій відсутній цей вказівник. Така структура називається *неправильно сформованим списком*. В графічному представленні, в такій структурі останнім елементом є пара, яка зображена без дужок, наприклад, в структурі **(1 2 . 3)** останнім елементом є пара **(2 . 3)**. Зауважимо, що в такій структурі пара може міститися лише вкінці списку. Схематичне позначення правильно- і неправильно-сформованих списків наведено на рис. 4.

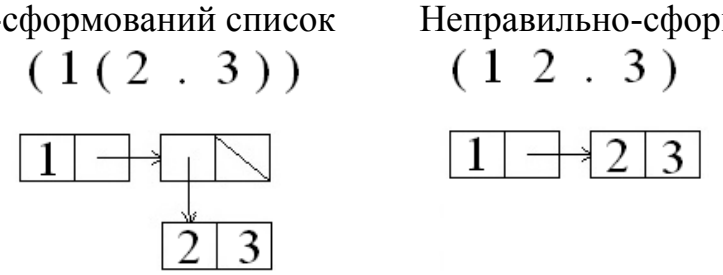

Рис. 4. Правильно- і неправильно-сформовані списки

У парах та списках допускається вкладеність, тобто пари, списки можуть бути складовими інших пар чи списків. Якщо пара чи список складається лише з атомів, то кажуть, що вони мають нульовий рівень вкладеності. Якщо у списку (парі) у вигляді елемента містить список (пара), які складаються лише з атомів, то кажуть, що вони мають перший рівень вкладеності. Якщо у списку (парі) у вигляді елемента є список (пара), які мають у вигляді елемента список (пару), які складаються лише з атомів, то кажуть, що вони мають нульовий рівень вкладеності. І так далі, допускається довільний рівень вкладеності. Наприклад, список (((((FIRST) 2) SECOND) 4) 5) має четвертий рівень вкладеності.

Приклад. Зобразити схематичне зображення S-виразу в комірках пам’яті. Визначити його тип, довжину, а також рівень вкладеності.

((A B) J (( )) (((D . E) F) . (K (U.V) S)) (M N . O) SS)

Очевидно, що тип даного S-виразу є або пара, або список або неправильно-сформований список. Для того, щоб визначити тип виразу потрібно акуратно погрупувати елементи від відкриваючої дужки до закриваючої, нумеруючи їх при цьому. Результат подано на рис. 5.

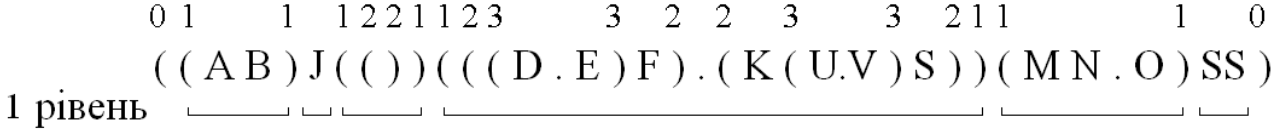

Рис. 5. Розбиття S-виразу на елементи

Отже, тип поданого S-виразу – список. Кількість елементів на 1 рівні – це довжина списку. Розберемо S-вираз по рівнях:

1. список **(A B)** – складається з:

1.1. атома **A**;
1.2. атома **B**;
2. атом **J**;
3. список **(( ))** – складається з:
3.1. порожнього списку **( )**;
4. пара **(((D . E) F) . (K (U.V) S))**, яка складається з двох елементів:
4.1. першим аргументом є список **((D . E) F)**, який містить
4.1.1. пару **(D . E)** та
4.1.2. атом **F**;
4.2. другим аргументом є список **(K (U.V) S)**, який складається з
4.2.1. атома **K**;
4.2.2. списку **(U.V)**, який містить
4.2.2.1. атом **U.V**;
4.2.3. атом **S**;
5. неправильно-сформований список **(M N . O)**, який містить
5.1. атом **M**;
5.2. пару **N . O**
6. атом **SS**.

Тому довжина S-виразу становить 6. Рівень вкладеності – 3 (максимальне число, яке позначене над дужками).

На рис. 6 подано схематичне зображення S-виразу в комірках пам’яті.

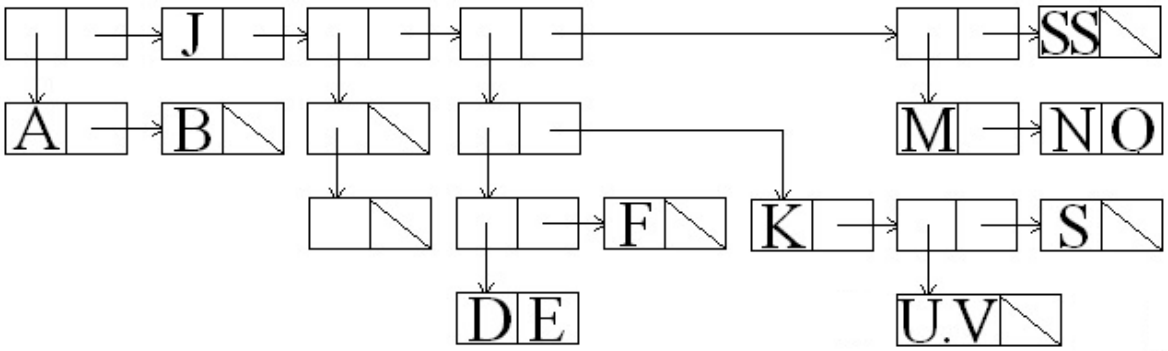

Рис. 6. Схематичне зображення S-виразу в комірках пам’яті

Таким чином, список – це багаторівнева або ієрархічна структура даних, в якій відкриваючі та закриваючі дужки знаходяться в строгій відповідності. Список, в якому немає ні одного елемента, називається пустим списком і позначається як "( )". Пустий список грає таку ж важливу роль в роботі зі списками, що і нуль в арифметиці. Пустий список може бути елементом інших списків. Кожен список має голову і хвіст. Голова списку – це перший елемент списку, а хвіст – залишок списку без голови (також список).

Отже, у мові програмування Scheme є три типи даних: атом, пара, список. Атом є простим типом. Пара і список – складеними типами, оскільки їх

елементами можуть бути об’єкти інших типів. Списки можуть бути правильно сформовані (із NIL-вказівником в кінці списку) і неправильно сформовані (без вказівника NIL). Зважаючи на це, при розгляді роботи функцій в мові Scheme будемо S-вираз конкретизувати на: атом, порожній список, непорожній список, неправильно-сформований список, пара.

5. Набір базових функцій-примітивів мови Scheme

Вважається, що строго функційна мова програмування має дуже обмежене число базових функцій, на основі яких можна побудувати всі інші функції. Як правило, вибирається сім примітивних функцій. Ця обмеженість є важливою в теоретичному плані в галузі програмування, оскільки вирішується проблема розв’язання довільної задачі з допомогою обмеженого набору примітивних функцій.

Виклик функції у функційних мовах програмування здійснюється у формі списку і має такий формат:

(function_name arg1 arg2 ... argN),

де **function_name** – ім'я функції,

arg1,**arg2**,...**argN** – її аргументи.

Тобто список має значення, якщо він є викликом функції.

Базові функції функційних мов програмування Lisp та Scheme наведено в табл. 3. В таблиці наведено по рядках функцію мови Lisp та її аналоги у мові Scheme.

Таблиця 3.

Базові функції функційних мов програмування

Мова Lisp	**Мова Scheme**
QUOTE	QUOTE
CAR	CAR
CDR	CDR
CONS	CONS
EQL	EQ?
ATOM	LIST?
	PAIR?
COND	COND

Отож, в Scheme примітивними функціями є такі:

1. QUOTE – функція блокування обчислень, яка також позначається як одинарна лапка «'». Функція використовується для блокування обчислень, що призводить до інтерпретації частини S-виразу не як програми, а як даних.

2. (CAR object) – вибирає голову непорожнього списку, пари або неправильно-сформованого списку.
3. (CDR object) – вибирає хвіст непорожнього списку, пари або неправильно-сформованого спику..
4. (CONS object1 object2) – функція-конструктор - об'єднює об'єкти 1 та 2.
5. (EQ? atom1 atom2) – порівнює два об’єкта на тотожність.
6. (LIST? object) – перевіряє чи є object списком.
7. (PAIR? object) – перевіряє чи є object парою.
8. (COND (condition1 action1) (condition2 action2)… (conditionN actionN)) – набуває значення з множини значень action1, action2, ..., actionN в залежності від значень умовних виразів condition1, condition2, …, conditionN.

Ще одною специфікою мови є використання крапки та круглих дужок в S-виразах. Синтаксис мови Scheme не визначений безпосередньо в термінах потоків символів. Замість цього, синтаксис визначається двома шарами:

- ***шар для читання***, який перетворює послідовність символів в списки, символи та інші константи;
- ***розширений шар***, який обробляє списки, символи та інші константи для розбору їх як вираз.

Правила для друку і читання використовуються разом. Наприклад, список друкується з дужками, при читанні пари з круглими дужками створюється список. Крім того, пара не-список друкується з точковою нотацією, а точка на вході ефективно виконує правила dot-нотації в зворотному напрямку, щоб отримати пару.

Одним з наслідків шару читання виразів є те, що ви можете використовувати dot-нотацію у виразах, які не оформлені з використання функції блокування: вираз **(+ 1 . (2))** повертає число **3**.

Це працює, тому що **(+ 1 . (2))** це просто ще один спосіб написання **(+ 1 2)**. Це практично погана ідея, щоб писати вирази застосування через dot-нотацію, однак це просто наслідок такого визначення синтаксису мови Scheme.

Як правило «крапка» допускається для читання тільки з наступною послідовністю круглих дужок, і тільки перед останнім елементом послідовності.

Таким чином, якщо подати вираз **(A . ( ))** як вхідні дані, то він сприйматиметься інтерпретатором як вираз **(A)**, а вираз **(A . (B))** сприйматиметься інтерпретатором як **(A B)**. На рис. 7 наведено схематичне зображення цих виразів у комірках пам’яті: спочатку із винесеними атомами в окремих комірках, потім із внесеними атомами.

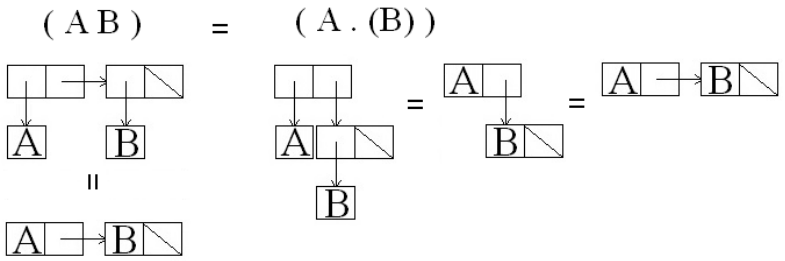

Рис. 7. Порівняння схематичного зображення комірок пам’яті двох S-виразів

Як видно з рис. 7 структури **(A B)** та **(A . (B))** у пам’яті представляються однаково. Тому про таку інтерпретацію вхідних даних потрібно пам’ятати під час розроблення застосувань мовою Scheme.

У функційному програмуванні діє важливий принцип композиції функцій. Тобто у виклиці будь-якої функції можливе використання на місці будь-якого її аргументу виклику іншої відомої функції.

Функція QUOTE – використовується для блокування обчислень. Частина S-виразу, яку потрібно передати як дані блокується до обчислень функцією QUOTE або символом «'».

Зауважимо, що функція QUOTE викликається згідно правил виклику функцій в мові Scheme – із круглими дужками. Натомість синтаксис використання функції «'» простіший – символ ставиться перед S-виразом, який потрібно заблокувати. Блокування діє в межах «глобальних» відкриваючої і закриваючої круглої дужки. Приклади використання функцій блокування наведено в табл. 4.

Таблиця 4.

Приклади викликів функцій блокування на мові Scheme

Текст програми		**Відповідь системи**
із використанням функції QUOTE	**аналог із використанням функції «'»**	
(QUOTE a)	'a	a
(QUOTE (cdr a b c))	'(cdr a b c)	(cdr a b c)
(QUOTE (a . b c))	'(a . b c)	Помилка використання «.»
(QUOTE ((test test test)))	'((test test test))	((test test test))
(QUOTE (QUOTE ((test))))	''((test))	'((test))
(QUOTE '(test))		'(test)

У заблокованому S-виразі можна розміщувати будь-який набір даних, однак оскільки інтерпретатор аналізує посимвольно код програми, то слід дотримуватися правил використання крапки в межах дужок, навіть якщо сам S-вираз є заблокованим.

Протилежною до функції блокування є функція-інтерпретатор **eval**, яка знімає блокування з S-виразу і обчислює його. Ця функція буде розглянута пізніше.

Функції-селектори. CAR та CDR називаються селекторними функціями, оскільки вони дають змогу вибирати частину S-виразу. Результатом функції (CAR object) завжди є перший елемент об’єкта object, якщо він непорожній і викликає помилку інтерпретатора в іншому випадку. Результат функції (CDR object) різниться у залежності від типу вхідного параметра object:

- ✓ для непорожнього і неправильно-сформованого списку функція CDR повертає список, який містить всі елементи object крім першого.
- ✓ для пари функція CDR повертає другий елемент пари.

Приклади викликів функцій-селекторів на мові Scheme наведено в табл. 5.

Таблиця 5.

Приклади викликів функцій-селекторів на мові Scheme

Функція	**Текст програми**	**Відповідь системи**
CAR	(CAR 'a)	помилка
	(CAR '( ))	помилка
	(CAR '(( )))	( )
	(CAR '(a))	a
	(CAR '(q w e r t y))	q
	(CAR '((one 1) (two 2)))	(one 1)
	(CAR '((a . b) c . d))	(a . b)
	(CAR '(a . b))	a
	(CAR '(a . ( )))	a
	(CAR '(a b . c))	a
	(CAR '((a b . c) d e . f))	(a b . c)
CDR	(CDR 'a)	помилка
	(CDR '( ))	помилка
	(CDR '(( )))	( )
	(CDR '(a))	( )
	(CDR '(q w e r t y))	(w e r t y)
	(CDR '((one 1) (two 2)))	((two 2))
	(CDR '((a . b) c . d))	(c . d)
	(CDR '(a . b))	b
	(CDR '(a . ( )))	( )
	(CDR '(a b . c))	(b . c)
	(CAR '((a b . c) d e . f))	(d e . f)

З табл. 5 видно, що область визначення і область значення функцій-селекторів CAR та CDR у Scheme є такою: вхідними аргументами може бути будь-який S-вираз крім атома і пустого списку, результатами функцій – будь-який S-вираз (табл. 6).

Таблиця 6.

Область визначення та область значень функцій CAR та CDR в мові Scheme

Область визначення	**Область значень**
1. Непорожній список 2. Пара 3. Неправильно-сформований список	1. Атом 2. Порожній список 3. Непорожній список 4. Неправильно-сформований список 5. Пара

Послідовне використання функцій-селекторів дає можливість вибрати будь-який елемент S-виразу. Дозволяється використовувати функції, які є комбінаціями CAR та CDR. Імена таких функцій починаються на C і закінчуються на R, а між ними знаходиться послідовність літер A та D (але не більше 4 літер, як правило в більшості реалізаціях Lisp). Ця послідовність вказує шлях обчислення: спочатку виконується обчислення в дужках, розміщених в «глибині» S-виразу, після чого сам під вираз заміняється результатом виконання і виконується «зовнішня» функція (рис. 8).

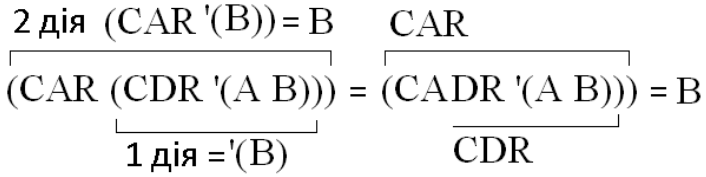

Рис. 8. Послідовність обчислення виразів

Приклади викликів комбінацій функцій-селекторів у мові Scheme подано у табл. 7.

Таблиця 7.

Приклади викликів комбінацій функцій-селекторів на мові Scheme

Текст програми		**Відповідь системи**
без комбінацій	**аналог із комбінаціями**	
(CAR (CDR (CDR '(q w e r t y))))	(CADDR '(q w e r t y))	e
(CAR (CDR (CDR '((q 1) (w 2) (e 3)))))	(CADDR '((q 1) (w 2) (e 3)))	(e 3)
(CDR (CDR '((q 1) (w 2) (e 3))))	(CDDR '((q 1) (w 2) (e 3)))	((e 3))
(CAR (CAR '((q w))))	(CAAR '((q w)))	q

Функція-конструктор. У мові Scheme функція CONS має два аргументи, кожен з яких може бути будь-яким з типів даних. Функція використовується для додання об'єкту, який задається першим аргументом, до об’єкту, заданого другим аргументом. Тобто об'єкт, який додається, стає головою S-виразу.

В залежності від типу другого аргумента можна отримати в результаті виконання функції список, пару або неправильно сформований список. Розглянемо три випадки виклику функції CONS.

1. *Другий аргумент є атом*. В результаті виконання функції утворюється пара, де першим елементом пари є перший аргумент, а другим елементом пари є другим аргумент.
2. *Другим аргументом є список*. В результаті виконання функції CONS, перший аргумент стає головою списку, заданого другим аргументом.
3. *Другий аргумент є пара або неправильно-сформований список*. В результаті виконання функції утвориться неправильно-сформований список, де перший аргумент буде головою неправильно-сформованого списку, а хвостом буде другий аргумент.

Приклади викликів конструктора наведено в табл. 8.

Таблиця 8.

Приклади викликів конструктора на мові Scheme

Тип другого аргумента	**Текст програми**	**Відповідь системи**	**Тип результуючого S-виразу**
Список	(CONS '() '())	(())	Список
	(CONS 'a '())	(a)	
	(CONS 'a '(b))	(a b)	
	(CONS '(a) '(b))	((a) b)	
	(CONS '(a . b) '(c d))	((a . b) c d)	
Атом	(CONS '() 'a)	(() . a)	Пара
	(CONS 'a 'b)	(a . b)	
	(CONS '(a) 'b)	((a) . b)	
	(CONS '(a . b) 'c)	((a . b) . c)	
Пара	(CONS '() '(a . b))	(() a . b)	Неправильно-сформований список
	(CONS 'a '(b . c))	(a b . c)	
	(CONS '(a . b) '(c . d))	((a . b) c . d)	
	(CONS 'a..b '(c . d))	(a..b c . d)	
Неправильно-сформований список	(CONS '() '(e f . g))	(() e f . g)	
	(CONS '(a . b) '(e f . g))	((a . b) e f . g)	
	(CONS '(a b . c) '(e f . g))	((a b . c) e f . g)	

Очевидно, що конструктор не може утворити атом і порожній список.

З табл. 8 видно, що області визначення і значення функції CONS є такими: вхідні аргументи можуть бути довільним типом даним, результат функції – непорожній список, пара або неправильно-сформований список (табл. 9).

Таблиця 9.

Область визначення та область значень функції CONS в мові Scheme

Область визначення		**Область значень**
перший аргумент	**другий аргумент**	
1. Атом 2. Порожній список 3. Непорожній список 4. Пара 5. Неправильно-сформований список	1. Атом 2. Порожній список 3. Непорожній список 4. Пара 5. Неправильно-сформований список	1. Непорожній список 2. Пара 3. Неправильно-сформований список

Предикати. Функції, які призначені для визначення чи їх аргументи володіють певними властивостями і як результат видають логічні константи, називаються предикатами. В мові Scheme прийнято позначати предикати символом «?», який ставиться в кінці назви функції. В мові Scheme предикатами є такі функції: EQ?, EQV?, EQUAL?, EMPTY?, LIST?, PAIR? тощо.

Функція порівняння EQ?. Функцією порівняння двох об’єктів є функція EQ?. Вона приймає два параметри і здійснює перевірку, чи ці два параметри представляють той самий об’єкт в пам’яті, тобто порівнюються вказівники. Функція є корисною для перевірки на тотожність атомів і повертає значення істини **#t** або хибності **#f**. У випадку аргументів непорожніх списків, навіть у разі їх тотожності, функція повертає значення хибності.

Природньо очікувати результат функції **EQ?** як **#t** для того ж символу, логічної константи або об’єкта та **#f** для значень, які мають різні типи даних або різні значення. Деякі реалізації також використовувати незмінні константи. **(EQ? '(1 2 3) '(1 2 3))** може бути **#f** при інтерпретації але **#t** при компіляції, оскільки вони можуть отримати ту ж адресу. Оскільки функція **EQ?** порівнює на представлення одного об’єкта в пам’яті, то, загалом кажучи, багато виразів за участю **EQ?** невизначені, а відповідь цієї функції під час порівняння двох однакових атомів (**#t** або **#f**) - залежить від її реалізації в конкретному діалекті.

Для того, щоб продемонструвати роботу функції **EQ?** для двох змінних, які вказують в пам’яті на один і той самий об’єкт, розглянемо функцію для присвоєння змінній певного значення. В мові **Scheme** для цього використовується функція **DEFINE**, яка має такий формат:

(DEFINE <name> <value>),

де **<name>** - ідентифікатор, **<value>** - його значення.

Наприклад, задамо ідентифікатору **a** значення **10**: **(DEFINE a '(a b c))**. Надалі можна використовувати ідентифікатор у тексті програми. Наприклад: **(car a)**.

Приклади викликів функції **EQ?** у мові **Scheme** подано у табл. 10.

Інша функція мови Scheme **EQV?** працює аналогічно до функції **EQ?** за винятком того, що вона завжди повертає істину для двох однакових примітивних значень її аргументів, навіть якщо дані є занадто великі щоб вміститися у вказівник. Для таких даних функція робить додаткову перевірку, що обидва параметри є однакового типу даних, і цільові значення обох об'єктів є однаковими. Приклади викликів функції **EQV?** подано в табл. 10.

Фактично, функція **EQ?** аналогічна до функції **EQV?**, але має змогу відрізнити тонкі відмінності, і може бути реалізованою більш ефективно. Відповідно до специфікації, це може бути реалізовано у вигляді швидкого та ефективного порівняння вказівників, на відміну від більш складних операцій для **EQV?**.

Нарешті функція **EQUAL?** працює за принципом функції **EQV?**, але вона може додатково коректно порівнювати порожні і непорожні списки, пари, неправильно-сформовані списки. Практично, функція EQUAL? здійснює рекурсивне порівняння на тотожність окремих елементів у двох складних S-виразів. Приклади використання функції **EQUAL?** подано в табл. 11.

Таблиця 10.

Приклади викликів функції EQ? На мові Scheme

Текст програми		**Відповідь системи**
функція EQ?	**Функція EQV?**	
(EQ? 5 5)	(EQV? 5 5)	#t
(EQ? #t #t)	(EQV? #t #t)	#t
(EQ? 2.5 2.5)	(EQV? 2.5 2.5)	#t
(EQ? -2 -2)	(EQV? -2 -2)	#t
(EQ? '( ) '( ))	(EQV? '( ) '( ))	#t
(EQ? '(1) '(1))	(EQ? '(1) '(1))	#f
(EQ? '(( )) '(( )))	(EQV? '(( )) '(( )))	#f
(EQ? '(a) '(a . ( )))	(EQV? '(a) '(a . ( )))	#f
(EQ? '(2 3) '(2 3))	(EQV? '(2 3) '(2 3))	#f
(DEFINE x '(2 3)) (DEFINE y '(2 3)) (EQ? x y)	(DEFINE x '(2 3)) (DEFINE y '(2 3)) (EQV? x y)	#f
(DEFINE x '(2 3)) (DEFINE y x) (EQ? x y)	(DEFINE x '(2 3)) (DEFINE y x) (EQV? x y)	#t
(EQ? (car '(q w)) 'q)	(EQV? (car '(q w)) 'q)	#t
(EQ? (car '(q w)) '())	(EQV? (car '(q w)) '())	#f

Таблиця 11.

Приклади викликів функції EQUAL? на мові Scheme

Текст програми	Відповідь системи
(EQUAL? 1 1)	#t
(EQUAL? '( ) '( ))	#t
(EQUAL? '(( )) '(( )))	#t
(EQUAL? '(a) '(b))	#f
(EQUAL? '(a) '(a))	#t
(EQUAL? '(a) '(a . ( )))	#t
(EQUAL? '(a (b)) '(a (b)))	#t
(EQUAL? '(a b c) '(a b c . ( )))	#t
(EQUAL? '(a . b) '(a . b))	#t
(EQUAL? '(a . b) '(a . (b)))	#f
(EQUAL? '(a b) '(a . (b)))	#t
(EQUAL? '(a (b) (( )) c . d) '(a (b) (( )) c . d))	#t
(EQUAL? '(a (b) (( )) c . d) '(a (b) ((( ))) c . d))	#f

В загальному:

1. Використовуйте функцію **EQ?** для порівняння двох числових атомів, атома і порожнього списку, перевірки на тотожність порожніх списків.
2. Використовуйте функцію **EQV?** для перевірки нечислових значень на тотожність.
3. Використовуйте функцію **EQUAL?** для перевірки на тотожність двох списків, векторів тощо.
4. Не використовуйте функцію EQ? якщо ви не знаєте типи аргументів.

Функції перевірки типу даних. При написанні програм мовою Scheme часто виникає запитання: чи є даний об'єкт атомом? чи парою? чи списком? У більшості функційних мов це питання вирішує предикат ATOM, який має один аргумент. Ця функція повертає #t, якщо об'єкт є атомом і NIL в іншому випадку. Порожній список NIL є атомом. У мові Scheme натомість існують предикати **LIST?** та **PAIR?**

Функція **LIST?** перевіряє чи аргумент є списком. Функція **PAIR?** перевіряє чи аргумент є парою. Зауважимо, що візуально це буде означати наступне – якщо в S-виразі є крапка – то тип такого S-виразу є пара. Тому неправильно-сформований список теж вважається парою. Окрім цього, зважаючи на специфіку обробки списків та дот-нотацію, відомо, що об’єкт **(a)**

можна представити у вигляді **(a . ( ))**. Тобто непорожні списки – також вважаються парою для інтерпретатора.

Приклади використання функцій LIST? та PAIR? подано в табл. 12.

Таблиця 12.

Приклади викликів функції LIST? та PAIR? на мові Scheme

Текст програми	**Відповідь системи**	
	function_name = **LIST?**	function_name = **PAIR?**
(function_name 'a)	#f	#f
(function_name '( ))	#t	#f
(function_name '(a))	#t	#t
(function_name '(a . b))	#f	#t
(function_name '(a . ( )))	#t	#t
(function_name '(a b . c))	#f	#t

Використовуючи ці дві функції можна однозначно визначити чи даний об’єкт є атомом, списком, порожнім списком, непорожнім списком, парою або неправильно-сформованим списком (табл. 13).

Найчастіше для реалізації програм потрібною буде функція atom?, яку потрібно буде написати із застосуванням функцій LIST? та PAIR?

Таблиця 13.

Визначення типу об’єкта із використанням функцій LIST? та PAIR? на мові Scheme

Тип S-виразу	**Який результат мають повернути функції?**	
	(LIST? x)	**(PAIR? x)**
Атом	#f	#f
Список (порожній або непорожній)	#t	–
Порожній список	#t	#f
Непорожній список	#t	#t
Пара або неправильно-сформований список	#f	#t

Іноді корисно конкретизувати який саме тип S-виразу – пара або неправильно-сформований список. Для вирішення такого завдання потрібно проаналізувати, що повертає функція **CDR** для цих типів даних.

- Для пари, функція **CDR** повертає завжди тип атом: **(cdr '(a . b)) = b**.
- Для неправильно-сформованого виразу – функція **CDR** для останнього елементу S-виразу повертає пару: **(cdr '(a b . c)) = (b . c)**.
- Для непорожніх списків функція **CDR** повертає список: **(cdr '(a)) = ( ).**
- Для атома і порожнього списку функція **CDR** не застосовується.

На основні цих правил можна реалізувати покроковий алгоритм визначення конкретного типу для заданого S-виразу.

Арифметичні операції та операції порівняння. Для проведення деяких елементарних обчислень необхідним є застосування елементарних арифметичних операцій та операцій порівняння. Виконання цих операцій у середовищах функційного програмування здійснюється у префіксній формі у вигляді правильного S-виразу, який може бути обчислений.

У Scheme арифметичні операції та операції порівняння позначаються як звичайно. Наприклад, * – операція множення, < – операція порівняння “менше”. Приклади виконання арифметичних операцій мовою Scheme наведено в табл. 14.

Таблиця 14.

Приклади виконання арифметичних операцій мовою Scheme

Текст програми	**Відповідь системи**
(+ 3)	3
(+ 5 6 8 9)	28
(* (+ 4 8) (- 6 2))	48
(/ 2 3 10)	2/30

Приклади виконання операцій порівняння наведено в табл. 15.

Таблиця 15.

Приклади виконання операцій порівняння мовою Scheme

Текст програми	**Відповідь системи**
(> 3 4 5)	#f
(< 3 4 5)	#t
(< 3 4 2)	#f

Умовна конструкція. Функція вибору COND має змінну кількість параметрів-пар типу (condition1 action1), (condition2 action2), …, (condition actionN)). У парі параметрів перший задає значення істина / неістина, другий задає можливе результуюче значення функції. Схема роботи складається з таких дій:

- визначення значення condition1, у випадку істинності вибирається значення action1 як результат функції;
- у випадку хибності відбувається перехід до наступної пари аргументів і т.д;
- процес завершується обчисленням значенням вибраного виразу.

Якщо ж жоден з логічних виразів condition1, condition2, ..., conditionN не набув значення істина, результуюче значення функції COND не визначено. У цьому випадку бажано останніми параметрами задавати вираз (#t f^*), де f^* –

деяке визначене значення, якого набуває функція COND, якщо попередні усі вирази condition1, condition2, ..., conditionN були значення "неістина".

Приклад. Нехай дано змінну **a**. Потрібно перевірити:

- якщо значення змінної більше 0 – вивести текст «a > 0»;
- якщо значення змінної менше 0 – вивести текст «a < 0»;
- інакше – вивести текст «a = 0».

Код програми буде таким:

```
(DEFINE a 5)
(cond  ((> a 0) '(a > 0))
       ((< a 0) '(a < 0))
       (#t      '(a = 0)))
```

Логічні операції. Для проведення логічних операцій у мові Scheme наявні такі функції як **AND**, **OR**, **NOT**, які відповідно є функціями логічного «і», «або» та «заперечення». Функції **AND** та **OR** використовуються аналогічно до арифметичних операцій в мові Scheme, тобто, на відміну від класичних бінарних функцій в інших мовах програмування, в мові Scheme дозволяється використовувати декілька параметрів, між якими функція буде проводити необхідні обчислення. Однак функція **NOT** приймає лише 1 аргумент.

Приклади використання логічних операцій подано в табл. 16.

Таблиця 16.

Приклади виконання логічних операцій мовою Scheme

Текст програми	**Відповідь системи**
(AND #t #t #t)	#t
(AND #t #f #f)	#f
(OR #f #f #f (EQ? 'a 'a))	#t
(NOT #f)	#t

Інші додаткові функції мови Scheme. Мова Scheme багата на вбудовані функції, а ті, яких не передбачено у стандартному варіанті діалекту – легко отримуються із підключених до програми бібліотек.

Наведемо декілька додаткових функцій мови Scheme:

NUMBER? – предикат, який перевіряє чи аргумент є числом

POSITIVE? – предикат, який перевіряє чи аргумент є додатнім числом.

NEGATIVE? – предикат, який перевіряє чи аргумент є від’ємним числом.

EVEN? – предикат, який перевіряє чи число є парним.

ODD? – предикат, який перевіряє, чи число є непарним.

MAX arg1 … argN – функція знаходить максимум з аргументів.

MIN arg1 … argN – функція знаходить мінімум з аргументів.

ABS – функція знаходить модуль числа.

EXP – функція, яка повертає e^x.

LN – функція, яка повертає натуральний логарифм аргумента.

SIN – функція, яка повертає синус аргумента.

COS – функція, яка повертає косинус аргумента.

SQRT – функція, яка повертає квадратний корінь аргумента.

NULL? – предикат, який перевіряє чи аргумент є порожнім списком.

LIST arg1 … argN – функція, яка формує список, який складається із аргументів.

LENGTH – функція, яка повертає довжину списку.

APPEND arg1… argN – функція, яка приймає аргументами списки і повертає список, який складається почергово із елементів кожного аргумента.

REVERSE – функція, яка обертає на верхньому рівні елементи списку.

6. Визначення нової функції

У функційному програмуванні нові функції створюються з допомогою базових на основі принципів композиції та рекурсії. Композиція – це правило створення нової функції, коли одна функція виступає аргументом для іншої. Наприклад, (CDR (CAR ‘((a d g) r t y))).

Для визначення нової функції мовою Scheme використовується така конструкція:

(lambda (<arguments>) (<body>)),

де **<arguments>** - список аргументів (через пробіл);

<body> - правильний S-вираз, який набуває значення.

Такий запис функцій називається лямбда-функцією, оскільки функція немає імені для її зручного виклику.

Наприклад, визначемо функцію одного аргументу, яка додає до переданого аргументу число 10:

(lambda (x) (+ x 10))

Згідно концепції Lisp, всі функції викликаються у форматі

(function_name param1 param2 … paramN)

Якщо функція є без імені, то можна підставити замість **function_name** сам опис функції.Наприклад, виклик вищеописаної функції для параметру 20:

((lambda (x)(+ x 10)) 20)

Недоліком такого опису функції є складність її виклику, оскільки для цього потрібно завжди використовувати повний опис функції.

Для створення функції з іменем потрібно використати один із форматів опису функції:

(DEFINE <name> (lambda (<arguments>) (<body>))),

або

(DEFINE (<name> <arguments>) (<body>)),

де **<name>** - ім’я функції;

<arguments> - список аргументів (через пробіл);

<body> - правильний S-вираз, який набуває значення.

Приклад визначення функції. Визначити функцію двох аргументів-списків, яка будує новий список з першого елемента першого списку і залишку після видалення першого елемента другого списку.

Нехай назва функції FUN1, вона має два аргументи U і V. Для побудови списку – результату необхідно використати функції-селектори і функцію-конструктор. Визначення функції є таким:

(DEFINE FUN1 (LAMBDA (U V) (CONS (CAR U) (CDR V))))

або таким

(DEFINE (FUN1 U V) (CONS (CAR U) (CDR V)))

Тоді виклик функції у вигляді

(FUN1 ‘(G H J) ‘(GG HH JJ))

приведе до такої відповіді системи функційного програмування:

(G HH JJ)

7. Середовище розробки Dr.Racket

Чому Scheme, чому Dr. Racket? Ось лише декілька тез в підтримку цих потужніх мов та середовища розробки:

1. **Все є виразами**. Більшість мов програмування є комбінацією двох синтаскично різних інгрідієнтів: вирази (об’єкти, які обчислюються для отримання значення) і вислови (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.
2. **Кожен вираз є або одиночним значенням або списком.** Одиночними значеннями в мові Scheme є атоми, числа, стрічки.
3. **Функціональний стиль програмування**. Так, існує багато функціональних мов програмування, а Lisp і Scheme не є «чисто»-функціональними мовами, однак багато розробників недооцінюють переваги цієї мови. 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

мовах: мутація даних (зміни даних на місці, а не повернення значення) і 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.

4. **Бібліотеки і документація.** Це може виглядати не як конкурентоздатна характеристика, але відзначимо, що 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 властива 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. Синтактичні 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. 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. На рис. 9 наведено вигляд головного вікна середовища.

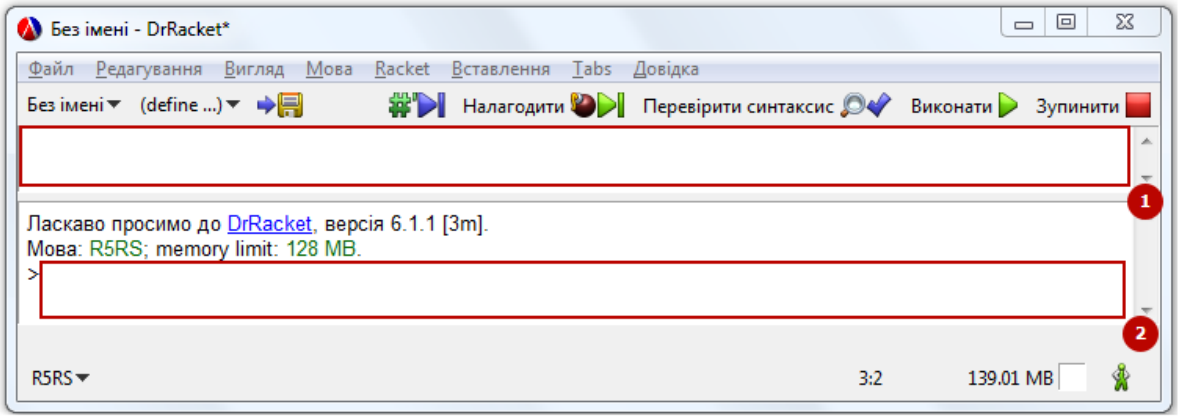

Рис. 9. Головне вікно середовища.

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.

На панелі інструментів винесено кнопки для від лагодження, перевірки синтаксису, запуску на виконання та зупинки програми (рис. 10).

Рис. 10. Панель інструментів в середовищі Dr.Racket

Приклад 1. Обчислимо суму 1 + 2. На мові Scheme потрібно написати (+ 1 2). Після цього запустимо програму на виконання клавішою F5 або кнопкою «Запустити». Результат виконання програми відобразиться в другій половині екрану (рис. 11).

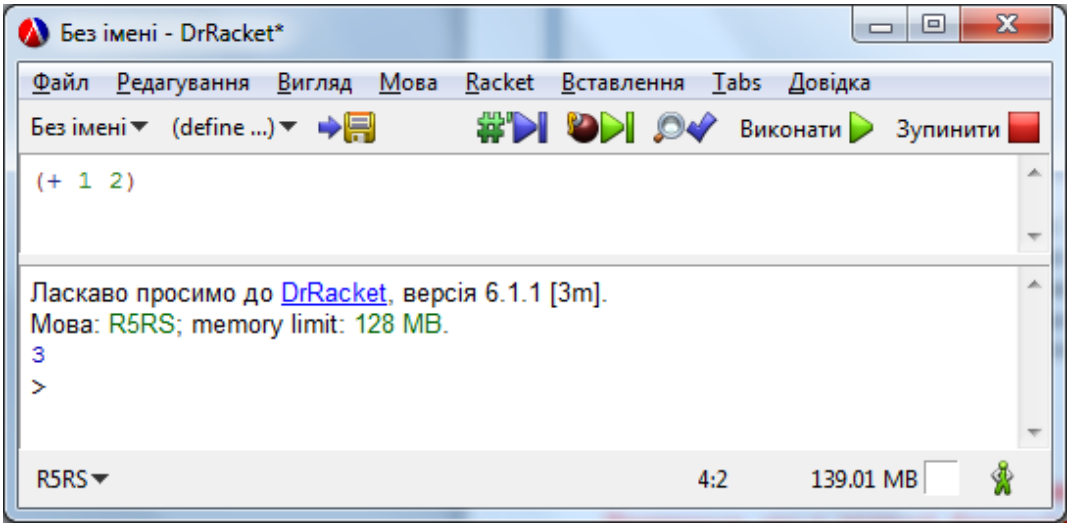

Рис. 11. Виконання програми в середовищі Dr. Racket

Крім цього, є можливість виконувати команди безпосередньо в командному рядку: після введеного тексту програми потрібно натиснути

клавішу ентер. Перемножимо числа 2, 3, 4: для цього введемо в командний рядок текст (* 2 3 4) і натиснемо клавішу ентер. Наступна стрічка міститиме шуканий результат (рис. 12).

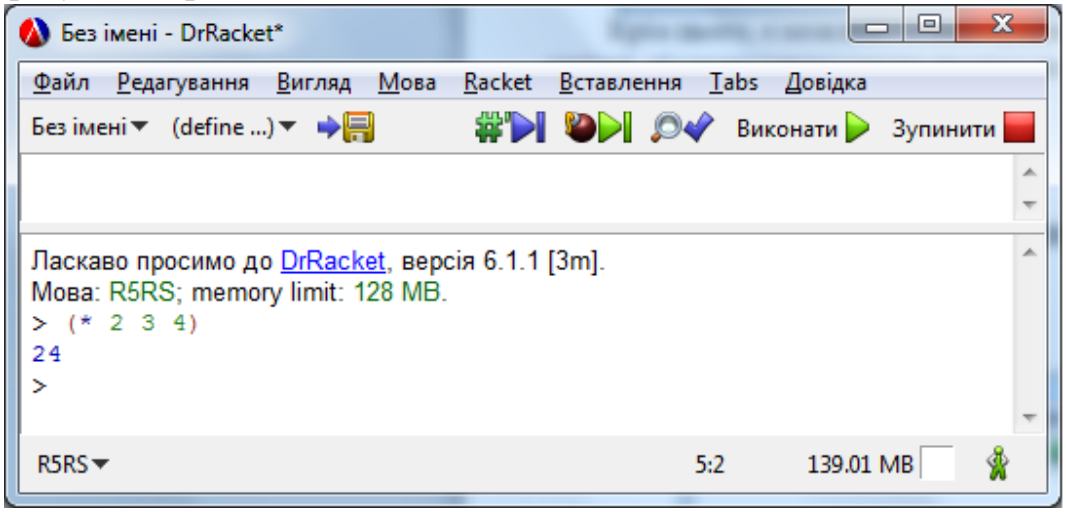

Рис. 12. Виконання програми в командному рядку

Щоб поставити коментар потрібно перед текстом поставити символ «;». Scheme не є чутливим до регістру і до кількості пробілів між об'єктами. Вирази (a) та (a) є тотожними.

На рис. 13 наведено приклад використання змінних та коментарів.

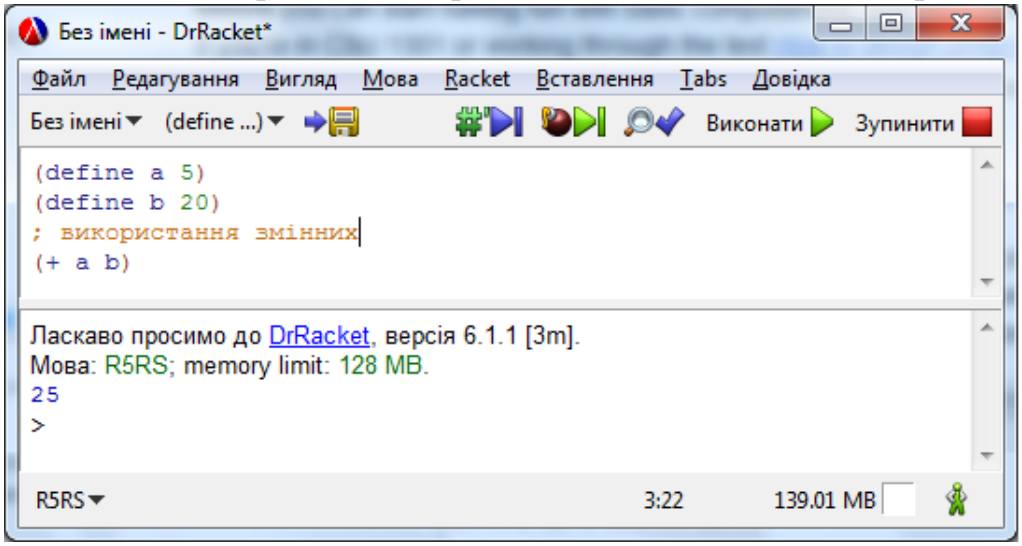

Рис. 13. Використання змінних та коментарів у середовищі Dr. Racket

Процес відлагодження коду в середовищі Dr.Racket.

Приклад 2. Обчислимо такий вираз: **(+ 2 (* 3 4))**. Замість кнопки «Виконати» натиснемо кнопку «Налагодити». Після цього головне вікно буде мати вигляд як на рис. 14.

Вікно має такі основні частини: 1 – текст програми, 2 – поточний результат виконання програми, 3 – меню для відлагодження, 4 – вікно із відображенням вмісту стеку, 5 – вікно для відображення поточних значень змінних.

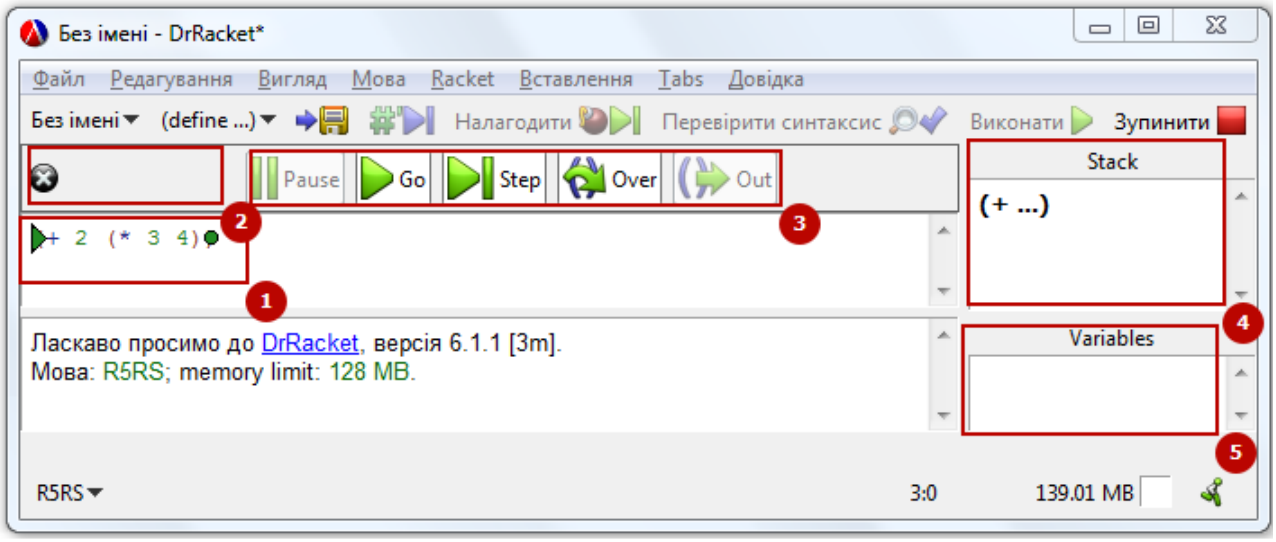

Рис. 14. Інтерфейс середовища Dr.Racket у режимі відлагодження

Спочатку в стеку є наявна функція додавання (рис. 14). Курсор відлагодження вказує на цю функцію. Натиснемо кнопку «Step». В стеку з'явилась функція множення (рис. 15). Курсор відлагодження вказує на цю функцію у вікні тексту програми.

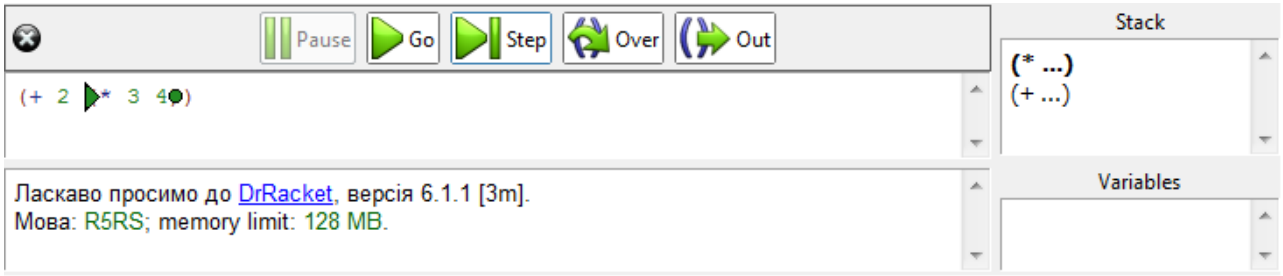

Рис. 15. Додання в стек функції множення

Натиснемо ще раз кнопку «Step». Оскільки далі за функцією множення немає інших функцій, відбудеться обчислення множення за вказаними параметрами 3 і 4. Проміжний результат (число 12) відображено у відповідній частині екрану (рис. 16).

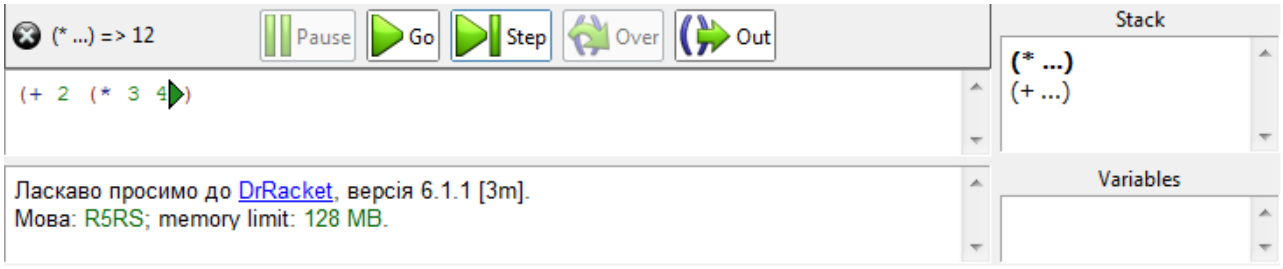

Рис. 16. Здійснення обрахунків для функції множення

Натиснемо ще раз кнопку «Step». Як видно із стеку, функція множення виконалась і вийшла з стеку. Курсор відлагодження вказує на кінець виклику функції додавання. Результат виконання функцій (число 14) відображено в області біля меню відлагодження (рис. 17).

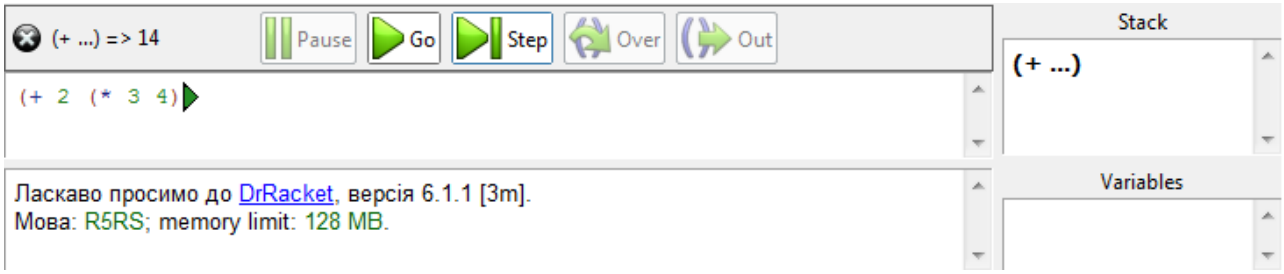

Рис. 17. Здійснення обрахунків для функції додавання

Чергове натискання кнопки «Step» завершує виконання програми – з стеку вивільняється функція додавання, а остаточний результат відображається у вікні командного рядка (рис. 18).

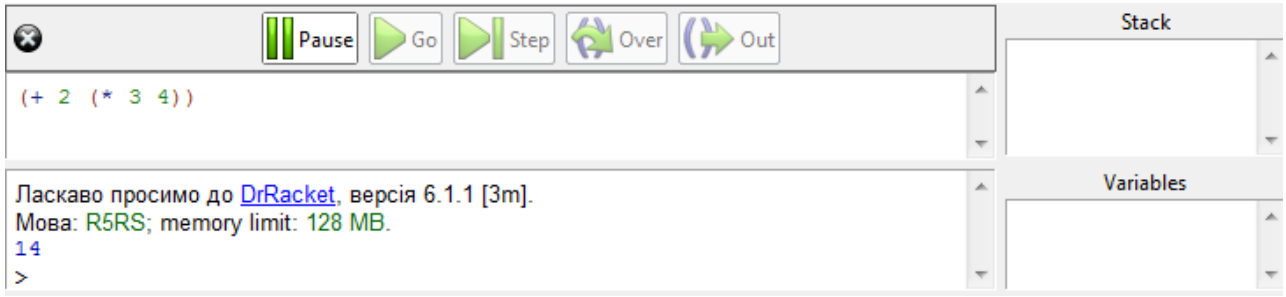

Рис. 18. Завершення виконання відлагодження програми

Приклад 3. Відлагодимо функцію з стор. 25:

(DEFINE FUN1 (LAMBDA (U V) (CONS (CAR U) (CDR V))))
(FUN1 '(G H J) '(GG HH JJ))

Процес відлагодження функції подано у табл. 17 – 18.

Таблиця 17.

Позиція курсору під час відлагодження

Крок	Позиція курсору відлагодження в тексті програми
1	(DEFINE FUN1 (LAMBDA (U V) (CONS (CAR U) (CDR V)))) FUN1 '(G H J) '(GG HH JJ)
2	(DEFINE FUN1 (LAMBDA (U V) CONS (CAR U) (CDR V))) (FUN1 '(G H J) '(GG HH JJ))
3	(DEFINE FUN1 (LAMBDA (U V) (CONS CAR U (CDR V)))) (FUN1 '(G H J) '(GG HH JJ))
4	(DEFINE FUN1 (LAMBDA (U V) (CONS (CAR U (CDR V)))) (FUN1 '(G H J) '(GG HH JJ))
5	(DEFINE FUN1 (LAMBDA (U V) (CONS (CAR U) CDR V))) (FUN1 '(G H J) '(GG HH JJ))
6	(DEFINE FUN1 (LAMBDA (U V) (CONS (CAR U) (CDR V)))) (FUN1 '(G H J) '(GG HH JJ))
7	(DEFINE FUN1 (LAMBDA (U V) (CONS (CAR U) (CDR V)))) (FUN1 '(G H J) '(GG HH JJ))
8	(DEFINE FUN1 (LAMBDA (U V) (CONS (CAR U) (CDR V)))) (FUN1 '(G H J) '(GG HH JJ)) (g hh jj)

Таблиця 18.

Вміст стеку, значення змінних та результат під час відлагодження функції

Крок	Стек	Змінні	Проміжний результат
1	(FUN1 …)		
2	(CONS …) (FUN1 …)	u => {g h j} v => {gg hh jj}	
3	(CAR …) (FUN1 …)	u => {g h j} v => {gg hh jj}	
4	(CAR …) (FUN1 …)	u => {g h j} v => {gg hh jj}	(CAR …) => g
5	(CDR …) (FUN1 …)	u => {g h j} v => {gg hh jj}	
6	(CDR …) (FUN1 …)	u => {g h j} v => {gg hh jj}	(CDR …) => (hh jj)
7	(FUN1 …)		(FUN1 …) =>(g hh jj)
8			(g hh jj)

Зазначимо, що командний рядок в середовищі «пам’ятає» попередньо введені команди. Таким чином, можна задекларувати змінну, а потім її використовувати (рис. 19).

```
Ласкаво просимо до DrRacket, версія 6.1.1 [3m].
Мова: R5RS; memory limit: 128 MB.
> (define a 20)
> a
20
> (+ a 7)
27
```

Рис. 19. «Запам’ятовування» командним рядком введених даних

Крім цього, в подальшому можна замінити значення вказівника і використовувати його в новому контексті (рис. 20).

```
Ласкаво просимо до DrRacket, версія 6.1.1 [3m].
Мова: R5RS; memory limit: 128 MB.
> (define a 20)
> a
20
> (+ a 7)
27
> (define a (lambda(x)(+ x 1)))
> a
#<procedure:a>
> (a 7)
8
```

Рис. 20. Зміна значення вказівника і його подальше використання

Відповідь системи **#<procedure:a>** говорить про те, що ідентифікатор **a** вказує на функцію.

Контрольні питання

1. Які основні властивості функційних мов програмування?
2. Наведіть приклади застосування функційних мов програмування.
3. Які є типи даних у функційних мовах програмування?
4. Що таке список? Наведіть переваги списків як ефективної форми представлення різноманітних даних.
5. Які категорії атомів використовуються для побудови S-виразів?
6. Чи можуть списки входити до складу пари?
7. Опишіть принципи фон Неймана, покладені в основу архітектури сучасних ЕОМ.
8. Що означає, якщо мова програмування є інтерпретуючою?
9. Що таке динамічна типізація?
10. Доведіть, що у функційному програмуванні список є рекурсивною структурою.
11. Поясніть твердження, що списки є ієрархічними структурами.
12. Чи можна у S-виразах доставляти круглі дужки без зміни їх структури?
13. Дайте визначення, що таке область визначення і область значення функції.
14. Поясніть, чому списки називають гнучкими структурами даних.
15. Що таке композиція функцій?
16. Що таке функції-предикати?
17. Які базові функції-предикати використовуються для написання нових функцій?
18. Для чого використовують базові функції селектори?
19. Як базові функції селектори пов’язані з конструктором?
20. З допомогою якої функції можна реалізувати розгалужений процес?
21. З допомогою якої функції можна визначити чи об’єкт є списком?
22. У якому вигляді записуються формальні параметри у визначенні нової функції?
23. Як блокуються обчислення S-виразів?
24. Що є областю визначення функцій-селекторів?
25. Що є областю значень функцій-предикатів?
26. Які діють правила для побудови комбінації функцій-селекторів?
27. Чи функція вибору завжди набуває значення?
28. Як працює функція EQ? EQV? EQUAL? зі списками?

Завдання для самоперевірки

1. До якого типу даних належать об’єкти:
   а) (123.e);
   б) (1 (123) e);
   в) 123;
   г) (1 2 3);
   д) 1.23;
   е) (1 d . 23)?
2. Скільки елементів верхнього рівня містять структури даних:
   а) ((1 (3 4)));
   б) ((1 A 5) ((S(E(F) K)) 4));
   в) ((3 ((G) H J)) ((L) R (2)));
   г)(((6)(0 1) 7(F)));
   д) (G (J K (123) 8) (45 (456)))?
3. Назвіть рівень вкладеності для структур даних
   а) (1 2 3);
   б) (1(2) (3) (1) (1 2 3));
   в) ((1) (2.3) (23));
   г) (1 2 (3(4(4) 4 (3(5 6)))) 6);
   д) (1 (2) (3 (1 (2))) 321 ((321) 3))?
4. Схематично представте структуру даних
   а) (A (S) T (S (E W) (E.W)) (A (A)));
   б) ((W E R).(D (F) G K(Y)));
   в) ((1.2) (3 (4 5) (3.4)) T Y (2 (2 3 4 (3.4)));
   г) (2. ((2).(4 (2) 3)));
   д) (((s f h) . (f . (f g (s)))) fsd);
   е) ((x . (c a x)) . ((c a x).x)).
5. Побудувати список, який задовольняє наступним умовам:
   а) містить два підсписки, перший з яких має три атоми, а другий - чотири атоми;
   б) містить три атоми;
   в) містить три складені об'єкти, і лише його другий елемент є атомом;
   г) перший елемент списку містить три атоми, а кількість атомів в усьому списку дорівнює 3.
   д) містить тільки порожній список;.
   е) всі елементи є списками з підсписками.

6. Визначити, що є головою, а що хвостом таких списків:
   а) ((A.B) (A D F) D A);
   б) (F G (A.B) ((H J K) K L));
   в) (A B (A.B);
   г) ((A B) G (S D F) J K);
   д) (A);
   е) (A B).
7. Визначте значення наступних S-виразів:
   а) (CAR (CDR(CAR '((a b c (d e)) 3 w 2 t 1 (1 2 3)))))
   б) (CAR '(CAR '((a b c (d e)) 3 w 2 t 1 (1 2 3))))
   в) (COND ((EQ? (CAR ‘(12 23 31)) 2) 123)((LIST? '(123)) 321) (#T 213))
   г) (EQ? '(ABC) '(ABC))
   д) (EQ? 'ABC 'ACB)
   е) (EQ? (LIST? 'ABC) (EQ? (CAR '((1) 2 3)) 1))
   ж) (CDR (CDR(CAR '(a b c (d e) 3 w 2 t 1 (1 2 3)))))
   з) (EQ? (LIST? '(ABC)) (EQ? (CAR '((1) 2 3)) 1))
   и) (EQ? (LIST? 'ABC) (EQ? (CAR '((1) 2 3)) '(1)))
8. Запишіть комбінацію селекторних функцій для вибору атома META із заданих списків:
   а) (a s d a s d META (a s d))
   б) (a s d (a s d) META (a s d))
   в) (a s d (a s d) (META) (a s d))
   г) (a s d (a s d) (META (a s d)))
   д) (a s d ((a s d) META) (a s d))
   е) ((a s d (a s d) (META)) (a s d))
   ж) ((a s d (a s d META)) (a s d))
   з) ((a s d (a s d (META))) (a s d))
9. Записати алгебраїчні вирази у формі S-виразів:
   а) $\dfrac{a \cdot x + b \cdot y}{c - z}$;
   б) $(x + y)^2 - \dfrac{a}{b - y}$.
10. Побудувати функцію предикатного типу, яка визначає чи введений список складається з одного елемента.
11. Побудувати функцію предикатного типу, яка визначає чи введений список містить першим елементом список.
12. Побудувати функцію логічного множення для двох S-виразів логічних значень.

Завдання до лабораторної роботи №1.

Використовуючи лише потрібні із 8 базових функцій-примітивів мови Scheme реалізувати функцію за варіантом. Передбачити коректну поведінку функції із всіма типами S-виразів: атом, порожній список, непорожній список, неправильно-сформований список, пара – виконання функції немає завершуватися із помилкою для цих типів S-виразів. Зобразити щонайменше 5 різних S-виразів схематичним представленням комірками в пам'яті.

Варіанти:

1. Довизначити функцію CAR(X) для випадку, коли X є атомом.
2. Довизначити функцію CDR(X) для випадку, коли X є атомом.
3. Визначити функцію QUADRAT(A, B, C, D), яка за вхідними параметрами формує спискову структуру ((A B) (C D)).
4. Визначити функцію, яка за 2 вхідними параметрами списками виду (A, B, C), (X, Y, Z) будує список (A, Y, Z). Передбачити випадок, якщо на вхід дано 2 порожніх списку, то на вихід – порожній список.
5. Визначити логічну функцію кон'юнкція двох аргументів.
6. Визначити функцію, яка за 2 вхідними параметрами списками виду (A, B, C), (X, Y, Z) будує список (B, C, Z). Передбачити випадок, якщо на вхід дано 2 порожніх списку, то на вихід – порожній список.
7. Визначити функцію TYPE(X), яка визначає тип виразу X. Можливі лише 3 типи виразів і відповідні значення функції TYPE: атом, пустий список, непустий список.
8. Визначити логічну функцію імплікація двох аргументів.
9. Визначити логічну функцію додавання за модулем 2 двох аргументів.
10. Визначити логічну функцію заперечення одного аргумента.
11. Визначити логічну функцію диз'юнкція двох аргументів.
12. Визначити логічну функцію стрілка Пірса (антидиз'юнкція) двох аргументів.
13. Визначити логічну функцію інверсія імплікації двох аргументів.
14. Визначити логічну функцію штрих Шеффера (антикон'юнкція) двох аргументів.
15. Визначити логічну функцію еквіваленція двох аргументів.
16. Визначити предикат PAIR? (X), який перевіряє чи аргумент – пара.
17. Визначити функцію M (A, B, C, D), яка за вхідними параметрами формує спискову структуру (A (B . C) (A D)).
18. Визначити функцію TYPE2(X), яка визначає тип виразу X. Можливі лише 3 типи виразів і відповідні значення функції TYPE2: атом, пустий список, пара.

Складові звіту

1. Тутильний аркуш.
2. Тема.
3. Мета.
4. Теоретичні відомості.
   4.1. Описати використані в програмі функції.
   4.2. Навести схематичне представлення 5 різних S-виразів комірками пам’яті.
5. Формулювання завдання
   5.1. Текст програми шрифтом Courier New
   5.2. Скріншот програми із результатами виконання
6. Висновки

Список літератури

1. A small place to discover languages in GitHub [Electronic resource]. – Web access: http://githut.info/ (2015).
2. Clojure [Electronic resource]. – Web access: http://clojure.org/ (2015).
3. GitHut. A small place to discover languages in GitHub [Electronic resource]. – Web access: http://githut.info/ (2015).
4. Introduction: Why Lisp? [Electronic resource]. – Web access: http://www.gigamonkeys.com/book/introduction-why-Lisp.html (2015).
5. Lisp (programming language) [Electronic resource]. – Web access: http://en.wikipedia.org/wiki/Lisp_%28programming_language%29 (2015).
6. LISP [Electronic resource]. – Web access: http://www.nist.gov/Lispix/doc/Lispix/Lisp-new.htm (2015).
7. LISP [Електронний ресурс]. – Веб-доступ до сторінки: http://lurkmore.to/LISP (2015).
8. List of programming languages by type [Electronic resource]. – Web access: http://en.wikipedia.org/wiki/List_of_programming_languages_by_type (2015).
9. PLT Scheme is a Racket [Electronic resource]. – Web access: http://racket-lang.org/new-name.html (2015).
10. Programming Language Popularity [Electronic resource]. – Web access: http://langpop.com/ (2015).
11. Quick: An Introduction to Racket with Pictures [Electronic resource]. – Web access: http://docs.racket-lang.org/quick/ (2015).
12. Scheme (programming language) [Electronic resource]. – Web access: http://en.wikipedia.org/wiki/Scheme_%28programming_language%29 (2015).
13. Scheme Tutorial [Electronic resource]. – Web access: https://classes.soe.ucsc.edu/cmps112/Spring03/languages/Scheme/SchemeTutorialA.html (2015).
14. What is Lisp used for today and where do you think it's going? [Electronic resource]. – Web access: http://stackoverflow.com/questions/794450/what-is-Lisp-used-for-today-and-where-do-you-think-its-going (2015).
15. Why is Lisp used for AI? [Electronic resource]. – Web access: http://stackoverflow.com/questions/130475/why-is-Lisp-used-for-ai (2015).
16. Why Racket? Why Lisp? [Electronic resource]. – Web access: http://practicaltypography.com/why-racket-why-Lisp.html (2015).
17. Зачем нужен ЛИСП С программисту? [Електронний ресурс]. – Веб-доступ до сторінки: http://Lisplearn.blogspot.com/2009/05/c.html (2015).
18. Рейтинг языков программирования [Електронний ресурс]. – Веб-доступ до сторінки: http://dou.ua/lenta/articles/language-rating-jan-2015/ (2015).

19. Хювёнен Э, Сеппянен И. Мир Лиспа. Т.1: Введение в язык Лисп и функциональное программирование / Э.Хювёнен, И.Сеппянен. – М.: Мир, 1990. – 458 с.
20. Хювёнен Э, Сеппянен И. Мир Лиспа. Т.2: Методы и системы программирования / Э.Хювёнен, И.Сеппянен. – М.: Мир, 1990. – 332 с.

Зміст