Київський національний університет імені Тараса Шевченка
Факультет комп'ютерних наук та кібернетики

М.М. Верес, О.В. Галкін,
Є.О. Демківський, Л.О. Катеринич, О.С. Шкільняк

Функціональне програмування

Навчальний посібник

Київ – 2022

УДК [004.432.42+510.648](075.8)

Ф94

Рецензенти:

д-р фіз.-мат. наук, проф. О.О. Марченко,

к-т фіз.-мат. наук, доц. Л.Л. Омельчук.

Рекомендовано до друку вченою радою
факультету комп'ютерних наук та кібернетики
(протокол № 2 від 27 вересня 2022 року)

Верес М.М., Галкін О.В., Демківський Є.О. та ін.
Ф94 Функціональне програмування: навчальний посібник / Верес М.М., Галкін О.В., Демківський Є.О. та ін. – Київ, 2022. – 178 с.

В посібнику викладено основні принципи функціонального програмування, базову термінологію і навички програмування мовами Lisp та Haskell. Окрема частина присвячена огляду теоретичного підґрунтя функціональної парадигми – лямбда-числення. Текст містить велику кількість прикладів та завдання для самостійної роботи, що має сприяти кращому засвоєнню матеріалу.

Для студентів, які вивчають дисципліну "Функціональне програмування", а також навчаються за освітніми програмами, пов'язаними з інформаційними та комп'ютерними технологіями.

УДК [004.432.42+510.648](075.8)

Зміст

Частина I. Мова Lisp

1.1 Основні функції Lisp

Вступ до Lisp

Lisp (LISt Processing) – одна з найдавніших (друга після Fortran) мов програмування високого рівня, що досі використовується. Характерними її ознаками є особлива дужкова префіксна нотація та робота зі списками як основними структурами даних. Така форма запису виникла під впливом лямбда-числення, яке і лягло в основу парадигми функціонального програмування. До найвідоміших діалектів мови Lisp загального призначення належать Common Lisp та Scheme. Надалі будемо розглядати саме діалект Common Lisp (CL).

Базовими елементами в Lisp є s-вирази (s-expressions), які складаються зі списків та атомів. Списки обмежуються дужками і можуть містити будь-яке число розділених пробілами елементів. Все, що не список, є атомом. Коментарі починаються з крапки з комою, поширюються до кінця рядка і трактуються як пробіл. Будь-яка послідовність цифр трактується як число. Рядкові літерали поміщуються в подвійні лапки. Всередині рядка зворотний слеш (\) екранує наступний знак, що викликає включення його в рядок «як є». Тільки два знаки мають бути екрановані в рядку: лапки і сам зворотний слеш:

`"foo"`: рядок, що містить знаки 'f', 'o' і 'o'.

`"fo\\o"`: рядок, що містить знаки 'f', 'o', '\' і 'o'.

Імена в Lisp можуть бути будь-якими, що починаються з літери. Ім'я не повинно збігатися з ключовим словом. Пробіл не може входити в ім'я.

Вирази Lisp нечутливі до регістру, тому наступні записи означатимуть те саме: **`(SETQ NAME 10)`**, **`(setq name 10)`**, **`(sEtq Name 10)`**. Насправді йдеться про те, що у внутрішньому представленні імена записані у верхньому регістрі, а при зчитуванні символів нижній регістр переводиться у верхній, якщо явно не вказується заборона на таку конвертацію (екрануванням окремого символа через \ або блоку символів, розміщеного між вертикальними рисками | |).

Атом

Найпростіші форми Lisp, – атоми, – можуть бути розділені на дві категорії: символи й усе інше. Символ, який обчислюється як форма, трактується як ім'я змінної і обчислюється в її поточне значення. Всі інші атоми (числа і рядки) є типом об'єктів – це самообчислювані об'єкти (self-evaluating objects). Це означає, що коли вираз передається в уявну функцію обчислення, він просто повертається.

Самообчислювані символи – такі, чиї імена починаються з :. Коли процедура читання обробляє подібне ім'я, вона автоматично визначає константну змінну з таким іменем і таким символом як значення.

Основні функції обробки списків

Базовими функціями роботи зі списками є: CAR, CDR, ATOM, CONS, EQ. Розглянемо їх докладніше.

Функції CAR і CDR

Функція CAR має вигляд:

```
(car список) => (s–вираз)
```

Функція CAR повертає голову списку. Приклад:

```
(car '(1 2 3))
```

На екрані отримаємо 1.

Функція CDR має вигляд:

```
(cdr список) => список
```

Функція CDR повертає хвіст списку. Приклад:

```
(cdr '(1 2 3))
```

На екрані отримаємо (2 3).

Функція АТОМ

Функція ATOM має вигляд:

(atom s-вираз) => T або список

Ця функція повертає True, якщо s-вираз виявиться атомом, в інакше функція поверне NIL. Розглянемо приклад:

(atom '1)

На екрані отримаємо True.

(atom '(1 2 3))

На екрані отримаємо NIL.

Функція CONS

Функція CONS включає новий елемент в початок списку. Вона будує (construct) новий список з переданих їй як аргументи голови і хвоста:

(cons голова хвіст)

Розглянемо приклад:

(cons 'a' (b c))

На екрані (A B C).

(cons '(a b)' (c d))

На екрані ((A B) C D).

Предикат EQ

EQ перевіряє тотожність двох символів. Предикат EQ порівнює два символи і повертає значення T, якщо вони ідентичні, в іншому випадку NIL.

>(eq 'X' cat)

На екрані NIL.

>(eq '1' 1)

На екрані T.

Предикат EQ в Lisp-системах зазвичай такий, що його можна застосовувати до спискових та числових аргументів, не отримуючи повідомлення про помилку; він не перевіряє логічної рівності чисел, рядків або інших об'єктів, а лише дивиться, чи представлені об'єкти в пам'яті обчислювальної машини фізично однієї і тієї ж самою структурою.

Предикат EQL

EQL порівнює числа однакового типу. Розглянемо приклад:

```
>(eql 3 3)
```

На екрані T.

```
>(eql 3 3.0)
```

На екрані NIL. Предикат EQL не можна використовувати для порівняння списків.

Предикат EQUAL

Узагальненням EQL є предикат EQUAL. Він працює як EQL, але, крім того, перевіряє однаковість двох списків:

```
>(equal 'X' X)
```

На екрані T.

Функція NULL

NULL перевіряє список на порожність.

```
>(null '())
```

На екрані T.

```
>(null (cdr '(a b c)))
```

На екрані NIL.

Функція NTH

Функція NTH виділяє n-й елемент списку.

```
>(nth 2 '(1 2 3))
```

На екрані 2.

Відповідно last – виділяє останній елемент списку: **(last x)**.

Функція LIST

Функція LIST створює список.

```
>(list x1 x2 x3)
```

Наприклад:

```
>(list 1 2 3)
```

На екрані (1 2 3).

Константи

Значенням константи є сама константа. Наприклад:

```
>3.14
```

3.14

Функція SET

Функція SET обчислює ім'я і зв'язує його. Наприклад:

```
>(set x 1)
```

На екрані 1.

```
>(set 'func x)
```

На екрані 1.

```
>func
```

На екрані 1.

Функція SETQ

Функція SETQ пов'язує ім'я не обчислюючи його. Наприклад:

```
>(setq x 1)
```

На екрані 1.

```
>(setq x (+ x 1))
```

На екрані 2.

Функція SETF

Це – узагальнена функція присвоювання.

```
>(setf mylist '(1 2 3))
```

На екрані (1 2 3).

```
>mylist
```

На екрані (1 2 3).

Введення-виведення в Lisp

Виведення здійснюється за допомогою функції PRINT:

```
>(print '(Hello world))
```

На екрані (Hello world).

Введення здійснюється за допомогою функції READ:

```
>(setq v (read))
```

На екрані 3.

```
>(print v)
```

На екрані 3.

Форматоване виведення

Функція FORMAT виводить рядок відповідно до зразка. Вона має наступний вигляд:

```
(format потік &rest аргументи)
```

Перший аргумент форми (потік) задає файл або пристрій виводу. Значення T означає виведення на екран. Розглянемо приклад:

```
>(format t "Hello, world")
```

На екрані отримаємо:

Hello, world

Розглянемо деякі коди:

1) **~%** – переводить рядок;

```
>(format t "~% Hello, world")
```

2) ~ **nT** – починає виведення з колонки n;

3) ~~ – виводить знак тильди.

Предикат BOUNDP

Предикат BOUNDP перевіряє наявність у символу значення. Наприклад:

```
>(setq x 10)
```

На екрані 10.

```
>(set 'func x)
```

На екрані 10.

```
>(boundp 'func)
```

На екрані T.

Предикат FBOUNDP

Предикат FBOUNDP перевіряє, чи пов'язано з символом визначення функції. Наприклад:

```
>(defun func (x y) (+ x y))
```

На екрані func.

```
>(fboundp 'func)
```

На екрані T.

```
>(fboundp 'sss)
```

На екрані NIL.

Послідовні обчислення

Пропозиції PROG1, PROG2 і PROGN дозволяють працювати з декількома формами, що обчислюються:

```
(prog1 форма1 форма2 ... формаN)
(prog2 форма1 форма2 ... формаN)
(progn форма1 форма2 ... формаN)
```

У цих спеціальних форм змінне число аргументів, які вони послідовно обчислюють і повертають в якості значення – значення першого (PROG1), другого (PROG2) або останнього (PROGN) аргументу. Розглянемо приклад:

```
>(progn (setq x 2) (setq y (* 3 x)))
```

На екрані 6.

Крапкова нотація для подання списків

Списки можна представляти у крапковій нотації, а саме:

```
(a1 a2 ... aN)
(a1.(a2. ... (aN.nil)...))
```

Розглянемо приклад:

```
(a b (c d) e)
(a.(b.((c.(d.nil)).(e.nil))))
```

Функції RPLACA і RPLACD

Функція RPLACA знищує попереднє значення голови списку і записує туди нове. Функція RPLACD знищує попереднє значення хвоста списку і записує туди нове. Розглянемо приклад:

```
>(rplaca '(a b c d) '(r))
```

На екрані ((R) A B C D).

```
>(rplacd '(a b c d) '(r))
```

На екрані (a r).

Властивості символу

Властивості символу записуються в список властивостей, що зберігається разом з ним. У Lisp з символом пов'язано значення, визначене функцією SETQ, а також порядок обчислень, визначений функцією DEFUN. Значення властивості, пов'язаної з символом, можна визначити за допомогою функції GET:

```
(get символ властивість)
```

Розглянемо приклад:

```
>(get 'l' a)
```

На екрані Nil.

```
>(setf (get 'l' a) 'hello)
```

На екрані Hello.

```
>(get 'l' a)
```

На екрані Hello.

Функція REMPROP

Видалення властивості і її значення здійснюється за допомогою функції REMPROP. Розглянемо приклад:

```
>(remprop 'l' a)
```

На екрані (A Hello).

```
>(get 'l' a)
```

На екрані NIL.

Список властивостей можна зчитати цілком, використовуючи SYMBOL-PLIST. Розглянемо приклад:

```
>(setf (get 'l' a) 'hello1)
```

На екрані Hello1.

```
>(symbol-plist 'l)
```

На екрані (A HELLO1).

1.2 Робота з файлами

Використання файлів

У Lisp введення і виведення здійснюються незалежно від конфігурації зовнішніх пристроїв через потоки (stream). У системи є декілька стандартних потоків, які є значеннями глобальних змінних. Найбільш важливі з них:

```
*STANDARD-INPUT*
```

```
*STANDARD-OUTPUT*
```

Ці системні змінні визначають для функцій введення (READ і ін.) і відповідно для функцій виведення файли за замовчуванням, якими на початку сеансу є дисплей або термінал користувача.

Файл спочатку потрібно відкрити (open), в залежності від його використання, для читання або для запису. У директиви OPEN може бути багато ключових параметрів. Найважливішим з них є DIRECTION:

```
(open файл &key: direction)
```

Значенням параметра можуть бути наступні ключі:

ПАРАМЕТР	ЗНАЧЕННЯ
INPUT	відкрити потік для введення
OUTPUT	відкрити потік для виведення
IO	відкрити двосторонній потік

Ключі у виклику функції не вимагають апострофа.

Потік можна за допомогою присвоювання зробити системним потоком за замовчуванням:

```
(setq *standard-input *thread)
```

Після цього дія функцій READ і PRINT була би перенаправлена на файл, а не на призначений для користувача пристрій введення. Розглянемо приклад:

```
(setq thread (open "D: \ templisp": direction :output))
(prin1 '(hello) thread)
(close thread)
```

Поряд з файлами за замовчуванням, іншим способом задати джерело для введення або одержувача для виведення є передача бажаного потоку в якості параметра виклику. Для функцій READ, PRINx це здійснюється через необов'язковий параметр. Більш докладним описом параметрів цих функцій буде наступний:

```
(read x & optional потік)
(print x & optional потік)
```

```
(prin1 x & optional потік)
(princ x & optional потік)
```

Робота з файлами зручніша через форму WITH-OPEN-FILE:

```
(with-open-file
                  (thread file режим1 ... режимN) форма1 ... формаN)
```

Виклик WITH-OPEN-FILE дозволяє відкрити певний потік в межах форми, який автоматично створюється на самому початку обчислень і закривається після їх завершення. Такі характеристики потоку, як його напрям, задаються через DIRECTION й інші ключі так само, як і в директиві OPEN.

Форми виклику WITH-OPEN-FILE обчислюються послідовно, і значення останньої з них повертається в якості результату виклику (неявний PROGN). Розглянемо приклад:

```
(with-open-file (thread1 "myfile": direction (read thread1))
```

Директива LOAD

На практиці написання програм здійснюється записом в файл визначень функцій, даних та інших об'єктів за допомогою наявного в програмному оточенні редактора. Після цього для перевірки визначень викликають інтерпретатор Lisp, який може прочитати записані в файл вирази директивою LOAD:

```
(load файл)
```

Вирази, що зчитуються, обчислюються так, ніби вони були введені користувачем. Після завантаження можна використовувати функції, значення змінних, значення властивостей й інші визначення.

1.3 Робота з функціями

Визначення нових функцій

Зазвичай функції визначаються за допомогою макросу DEFUN. Типове використання DEFUN має вигляд:

```
(defun name (parameter)
    "Optional documentation string."
    тіло-функції)
```

Як ім'я може використовуватися будь-який символ. Список параметрів функції визначає змінні, які будуть використовуватися для зберігання аргументів, переданих при виклику функції. Якщо функція не приймає аргументів, то список порожній і записується як ().

Розрізняють обов'язкові, необов'язкові, множинні та іменовані параметри. За списком параметрів може знаходитись рядок, який описує призначення функції. Після того, як функція визначена, цей рядок (рядок документації) буде асоційований з ім'ям функції і може бути пізніше отриманий за допомогою функції DOCUMENTATION. Наприклад, наступний код

```
(documentation 'foo' function)
```

поверне рядок документації для функції foo. Однак, варто зауважити, що документація призначається для людей, а не для роботи програм. Реалізації Lisp не зобов'язані зберігати її і можуть видалити її в будь-який час, тому програми, що переносяться, не повинні залежати від наявності документації до функції.

Тіло DEFUN складається з будь-якого числа виразів Lisp. При виконанні функції вони обчислюються по порядку, і результат обчислення останнього виразу повертається як значення функції. Для повернення з будь-якої точки функції може використовуватися спеціальний оператор RETURN-FROM. Розглянемо приклад 1:

```
(defun hello1() (format t "hello, world"))
```

Виклик цієї функції буде мати вигляд:

```
(hello1)
```

Приклад 2:

```
(defun verbose1 (x y)
```

```
(format t "Summing ~d and ~d. ~%" x y) (+ x y))
```

Ця функція називається verbose1, отримує два аргументи, які зв'язуються з параметрами x та y, має рядок документації, і її тіло складається з двох виразів. Значення, повернене викликом функції +, стає значенням функції verbose1.

Списки параметрів функцій

Функція може мати як обов'язкові, так і необов'язкові параметри. Коли список параметрів є простим списком імен змінних, як в verbose1, то параметри називаються обов'язковими. Коли функція викликається, вона повинна отримати рівно по одному аргументу для кожного з обов'язкових параметрів.

Необов'язкові параметри

Для визначення функції з необов'язковими параметрами, після списку обов'язкових параметрів слід помістити символ &OPTIONAL, за яким перераховуються імена необов'язкових параметрів. Розглянемо приклад:

```
(defun foo (a b &optional c d) (list a b c d))
```

Дану функцію можна викликати

```
(foo 1 2)
```

на екрані (1 2 NIL NIL), або

```
(foo 1 2 3 4)
```

на екрані (1 2 3 4). Якщо необхідно вказати інше значення за замовчуванням, відмінне від NIL, то тоді маємо:

```
(defun foo (a &optional (b 10)) (list a b))
```

Тоді при виклику

```
(foo 1)
```

на екрані отримаємо (1 10).

Іноді корисно буде знати, чи було значення необов'язкового параметра задано користувачем, або використовувалося значення за замовчуванням. Замість того, щоб писати код, який перевіряє, чи є дорівнює передане значення значенню за замовчуванням (це все одно не буде працювати, оскільки користувач може явно задати значення, рівне значенню за замовчуванням), можна додати ще одне ім'я змінної до списку параметрів після виразу для значення за замовчуванням. Зазначена змінна буде мати справжнє значення, якщо користувач задав значення для аргументу, і NIL в іншому випадку. Розглянемо приклад:

```
(defun foo (a b & optional (c 3 supplied))
    (list a b c supplied))
```

Тоді виклик даної функції може мати вигляд:

```
(foo 1 2)
```

на екрані (1 2 3 NIL),

```
(foo 1 2 3)
```

на екрані (1 2 3 T),

```
(foo 1 2 4)
```

на екрані (1 2 4 T).

Залишкові (REST) параметри

У Lisp можна написати функцію зі змінною кількістю аргументів, просто описавши множину необов'язкових параметрів. Для цього необхідно використовувати параметр &REST (залишковий параметр), тоді будь-які аргументи, що залишилися після зв'язування обов'язкових і необов'язкових параметрів, будуть зібрані в список, який стане значенням залишкового параметра &REST. Розглянемо приклад:

```
(defun myfunc (n & rest numbers)
    (list n (car numbers)))
```

Тоді виклик функції

```
(myfunc 2 3 4)
```

поверне (2 3).

Іменовані параметри

Припустимо, що необхідно встановити значення лише для деяких параметрів, розташованих не послідовно.

Для цього зручно використовувати іменовані параметри. Назви налаштовуються за допомогою символу &KEY.

Розглянемо приклад:

```
(defun foo (&key abc) (list abc))
```

Коли функція викликається, кожен іменований параметр пов'язується зі значенням, яке вказано після ключового слова, що має те саме ім'я, що й параметр.

Якщо ключове слово не вказано у списку аргументів, відповідний параметр отримує значення за замовчуванням, тобто принцип той самий, як і для необов'язкових параметрів.

Оскільки іменовані аргументи мають мітку, то вони можуть бути вказані у будь-якому порядку, якщо вони йдуть після обов'язкових аргументів.

Приклад:

```
(foo) ==> (NIL NIL NIL)
(foo: a 1) ==> (1 NIL NIL)
(foo: b 1) ==> (NIL 1 NIL)
(foo :c 1) ==> (NIL NIL 1)
(foo :a 1 :c 3) ==> (1 NIL 3)
(foo :a 1 :b 2 :c 3) ==> (1 2 3)
(foo :a 1 :c 3 :b 2) ==> (1 2 3)
```

Також, як і для необов'язкових параметрів, для іменованих параметрів можна задавати вираз для обчислення значення за замовчуванням та ім'я SUPPLIED-P-змінної.

Розглянемо приклад:

```
(defun foo (&key (a 0) (b 0 b-supplied-p) (c (+ ab)))
                                (list abc b-supplied-p))
```

Тоді виклик даної функції матиме вигляд:

```
(foo: a 1) ==> (1 0 1 NIL)
(foo: b 1) ==> (0 1 1 T)
(foo :b 1 :c 4) ==> (0 1 4 T)
(foo :a 2 :b 1 :c 4) ==> (2 1 4 T)
```

Якщо з певних причин необхідно, щоб користувач використовував імена аргументів, які відрізняються від імен параметрів, можна замінити ім'я параметра на список, що містить ім'я, яке буде використовуватися користувачем під час виклику, та ім'я параметра.

Приклад:

```
(defun foo (&key ((:apple a))((:box b) 0)
      ((:charlie c) 0 c-supplied-p)) (list abc c-supplied-p))
```

Виклик такої функції може мати вигляд:

```
(foo :apple 10 :box 20 :charlie 30) ==> (10 20 30 T)
```

Спільне використання різних типів параметрів

Використання всіх чотирьох типів параметрів в одній функції хоч і є цілком можливим, але застосовується рідко.

Коли використовується більше одного типу параметрів, вони мають бути оголошені у такому порядку:

- спочатку вказуються імена необхідних параметрів,
- потім – необов'язкових,
- потім – залишкових (&REST),
- в кінці – іменованих параметрів.

Але зазвичай у функціях, які використовують кілька типів параметрів, комбінують необхідні з одним з інших видів параметрів, або, можливо, комбінують необов'язкові та залишкові параметри.

Комбінація необов'язкових та іменованих параметрів досить ризикована. Розглянемо приклад:

```
(defun foo (x &optional y &key z) (list xyz))
```

Цей виклик до проблем не призводить:

```
(foo 1 2 :z 3)==> (1 2 3)
```

Однак виклик:

```
(foo 1 :z 3) ==> ERROR
```

призводить до помилки.

Це тому, що ім'я параметра :z береться як значення для необов'язкового параметра y, залишаючи для обробки лише аргумент 3.

При цьому Lisp очікує, що в цьому місці зустрінеться або пара ім'я/значення, або не буде нічого, і одиничне значення призведе до появи помилки.

Також небезпечне поєднання &REST та &KEY. Розглянемо приклад:

```
(defun foo (&rest rest &key abc) (list rest abc))
```

Тоді виклик

```
(foo :a 1 :b 2 :c 3)
```

Дасть результат

```
((:A 1 :B 2 :C 3) 1 2 3)
```

Якщо в списку параметрів використовуються і залишкові й іменовані параметри, то виконуються обидві дії – всі значення, що залишилися, збираються в список, який присвоюється параметру &REST, а також відповідні значення присвоюються іменованим параметрам.

Повернення значень із функції

Всі функції так чи інакше використовують звичайну поведінку, що полягає у поверненні значення останнього обчисленого виразу як власного значення, що повертається.

Однак, іноді буває потрібно повернути значення із середини функції, вириваючись таким чином із вкладених керуючих конструкцій.

У такому разі можна використовувати спеціальний оператор RETURN-FROM який призначений для негайного повернення будь-якого значення з функції. (Насправді RETURN-FROM не прив'язаний до функцій; він використовується для повернення з блоку коду, визначеного за допомогою оператора BLOCK. Однак, DEFUN автоматично поміщає тіло функції в блок коду з тим же ім'ям, що і функція).

Розглянемо приклад:

```
(defun foo (n)
 (dotimes (i 10)
 (dotimes (j 10)
 (when (> (* ij) n)
 (return-from foo (list ij))))))
```

1.4 Функції вищого порядку та анонімні функції

Функції вищого порядку

У Lisp можна передавати одну функцію як аргумент в іншу функцію.

Спеціальний оператор FUNCTION забезпечує механізм отримання об'єкта-функції. Він приймає єдиний аргумент та повертає функцію з цим ім'ям. Ім'я не екранується.

Розглянемо приклад. Нехай визначено функцію foo:

```
(defun foo (x) (* 2 x))
```

Тоді отримати об'єкт функцію можна так:

```
(function foo)
```

Після того, як отримано об'єкт-функцію, її можна виконати.

Common Lisp надає дві функції для виконання функції через об'єкт-функцію: FUNCALL і APPLY. Вони відрізняються тим, як отримують аргументи, які будуть передані функції, що викликається.

FUNCALL – це функція, яка використовується тоді, коли під час написання коду відома кількість аргументів, які будуть передані функції.

Першим аргументом FUNCALL є об'єкт-функція, що запускається, а решта аргументів передається цій функції.

Приклад 1:

```
(funcall #'foo 1 2 3)
```

рівносильно

```
(foo 1 2 3)
```

Приклад 2:

```
(defun plot (fn min max step))
 (loop for i from min to max by step do
 (loop repeat (funcall fn i) do (format t "*"))
 (format t "~%")))
```

Вираз FUNCALL обчислює значення функції для кожного значення i.

Внутрішній цикл використовує це значення для визначення того, скільки разів надрукувати знак «зірочка».

Функцію plot можна викликати з будь-якою функцією, яка має один числовий аргумент. Наприклад:

```
 (plot #'exp 0 4 1/2)
*
*
**
****
*******
************
********************
*********************************
******************************************************
NIL
```

Функція APPLY як перший аргумент приймає об'єкт-функцію. Але після першого аргументу замість перелічення окремих аргументів вона приймає список. Потім APPLY застосовує функцію до значень у списку.

Попередній код можна переписати за допомогою функції APPLY наступним чином:

```
(defun plot (fn min max step))
 (loop for i from min to max by step do
 (loop repeat (apply fn (list i)) do (format t "*"))
 (format t "~%")))
```

Анонімні функції

У Lisp можна використовувати лямбда-функції. Лямбда-вираз у Lisp має вигляд:

```
(lambda (parameters) body)
```

Розглянемо приклад:

```
((lambda (xy) (+ xy)) 2 3)==> 5
```

Лямбда-вирази можна передавати як аргументи у функції вищого порядку:

```
 (plot # '(lambda (x) (* 2 x)) 0 10 1)
```

1.5 Робота зі змінними та константами

Змінні

Common Lisp є динамічно типизованою мовою: помилки типів виявляються динамічно.

Усі значення Common Lisp, принаймні концептуально, є посиланнями на об'єкти.

Тому присвоєння змінній нового значення змінює те, на який об'єкт посилається змінна (тобто, куди вона посилається), але не чинить ніякого впливу на об'єкт, на який змінна посилалася раніше.

Однак, якщо змінна містить посилання на об'єкт, що може змінюватися, то можна використовувати дане посилання для зміни цього об'єкта, і ця зміна буде видимою будь-якому коду, який має посилання на цей же об'єкт.

Нові змінні можна вводити за допомогою спеціального оператора LET. Шаблон форми LET має такий вигляд:

```
(let (variable) body-form)
```

Тут кожна variable є формою ініціалізації змінної.

Кожна форма ініціалізації є або списком, що містить ім'я змінної і форму початкового значення, або, як скорочення для ініціалізації змінної значенням NIL, просто ім'ям змінної.

Наступна форма LET, наприклад, пов'язує три змінні x, y та z з початковими значеннями 10, 20 і NIL:

```
(let ((x 10) (y 20) z)
................)
```

При обчисленні форми LET спочатку обчислюються всі форми початкових значень.

Потім, перед виконанням форм тіла, створюються та ініціалізуються у відповідні початкові значення нові прив'язки.

Всередині тіла LET імена змінних посилаються на новостворені прив'язки.

Після LET імена продовжують посилатися туди ж, на що вони посилалися перед LET.

Значення останнього виразу тіла повертається як значення виразу LET.

Як і параметри функцій, змінні, що вводяться LET, зв'язуються знову щоразу, коли потік управління заходить в LET

Якщо використовуються вкладені зв’язуючі форми, які вводять змінні з однаковими іменами, то прив'язки внутрішніх змінних приховують зовнішні прив'язки. Розглянемо приклад:

```
(defun foo (x)
 (format t "Параметр: ~a~%" x); |<- x - аргумент
 (let ((x 2))
 (format t "Зовнішній LET: ~a~%" x); |<---- x = 2
```

```
(let ((x 3))
(format t "Внутрішній LET: ~a~%" x)); | <- x = 3
(format t "Зовнішній LET: ~a~%" x))
(format t "Параметр: ~a~%" x))
```

Кожне звернення до x посилатиметься на прив'язку з найменшою навколишньою областю видимості.

Як тільки потік управління залишає область видимості якоїсь зв'язуючої форми, прив'язка з безпосередньої навколишньої області видимості перестає приховуватися й x посилається вже на неї.

Таким чином, результатом виклику foo буде наступне виведення:

```
(foo 1)
```

Параметр: 1

Зовнішній LET: 2

Внутрішній LET: 3

Зовнішній LET: 2

Параметр: 1

NIL

Глобальні змінні

Common Lisp надає два способи створення глобальних змінних: DEFVAR і DEFPARAMETER.

Обидві форми приймають ім'я змінної, початкове значення та опціональний рядок документації.

Після створення змінної за допомогою DEFVAR або DEFPARAMETER ім'я може бути використане будь-де для посилання на поточну прив'язку цієї глобальної змінної.

Використання DEFVAR та DEFPARAMETER має вигляд:

```
(defvar count 0)
(defparameter gap-tolerance 0.001)
```

Загальний вигляд DEFVAR і DEFPARAMETER має вигляд:

defparameter *name initial-value [documentation] => name*

defvar *name [initial-value [documentation]] => name*

Відмінність між цими двома формами полягає в тому, що DEFPARAMETER завжди присвоює початкове значення названій змінній, а DEFVAR робить це тільки якщо змінна була не визначена.

Форма DEFVAR також може бути використана без початкового значення для визначення глобальної змінної без надання їй значення. Така змінна називається незв'язаною.

Розглянемо приклад:

```
(defvar x 10)
  (defun foo () (format t "x: ~d~%" x))
```

DEFVAR створює глобальну прив'язку змінної x зі значенням 10.

Звернення до x в foo буде шукати поточну прив'язку динамічно.

Можна використовувати LET для створення нової прив'язки, яка тимчасово приховає глобальну прив'язку і foo надрукує інше значення.

```
(let ((x 20)) (foo))
```

Присвоювання нового значення впливає лише на поточну прив'язку.

Щоб побачити це, можна перевизначити foo, додавши присвоєння значення змінній x:

```
(defun foo ()
 (format t "Перед присвоєнням ~ 18tX: ~ d ~ %" x)
 (setf x (+ 1 x))
 (format t "Після присвоєння ~ 18tX: ~ d ~ %" x))
```

Тоді виклик foo дає

```
(foo)
```

Перед присвоєнням X: 10

Після присвоєння X: 11

NIL

Константи

Всі константи є глобальними та визначаються за допомогою директиви DEFCONSTANT.

Базова форма DEFCONSTANT має вигляд:

```
(defconstant name initial-value-form [ documentation-string ])
```

DEFCONSTANT спричиняє глобальний ефект на ім'я, що використовується: після цього, ім'я може бути використане тільки для звернення до константи; воно не може бути використане як параметр функції або бути перезв'язане за допомогою будь-якої іншої зв’язуючої форми.

1.6 Керуючі конструкції

Стандартні керуючі конструкції

Найзагальнішу форму умовного виразу представляє спеціальний оператор IF, який має наступну базову форму:

```
(if condition then-form [else-form])
```

Умова CONDITION обчислюється, і якщо її значення не NIL, тоді виконується THEN-FORM і отримане значення повертається.

Інакше виконується ELSE-FORM, якщо вона є, та повертається її значення.

Якщо condition дає NIL і немає ELSE-FORM, тоді IF повертає NIL.

Розглянемо приклади:

```
(if (> 2 3) "Yup" "Nope") ==> "Nope"
(if (> 2 3) "Yup") ==> NIL
(if (> 3 2) "Yup" "Nope") ==> "Yup"
```

Однак, THEN-FORM та ELSE-FORM обмежені однією Lisp-формою.

Це означає, якщо тут необхідно виконати послідовність дій, потрібно обернути їх у якийсь інший синтаксис.

Наприклад:

```
(if (condition) (progn (expression1) (expression2)))
```

У Lisp є макрос WHEN, який дозволяє попередній вираз записати наступним чином:

```
(when (condition)
 (expression1)
 (expression2))
```

Але якби він не був вбудований у стандартну бібліотеку, можна було б визначити WHEN як макрос:

```
(defmacro when (condition &rest body)
 `(if ,condition (progn ,@body)))
```

Супутнім до макросу WHEN є UNLESS, який «обертає» умову, виконуючи форми з тіла, лише якщо умова хибна.

Іншими словами:

```
(defmacro unless (condition &rest body)
 `(if (not, condition) (progn, @ body)))
```

Макрос COND

Макрос COND – аналог switch-case в C++.

Розглянемо приклад. Нехай є вираз

```
(if a
 (expression1)
 (if b
 (expression2)
 (expression3)))
```

Цей вираз можна переписати у вигляді:

```
(cond
 (a (expression1))
 (b (expression2))
 (t (expression3)))
```

Якщо гілка немає форм після умови, то повертається саме значення умови.

За домовленістю, гілка, що представляє останній випадок else в ланцюжку if/else-if, записується з умовою T.

Оператори AND, OR та NOT

AND, OR та NOT – булеві логічні оператори.

Розглянемо приклад використання цих операторів:

(not nil) `==> T`

(not (= 1 1)) `==> NIL`

(and (= 1 2) (= 3 3)) `==> NIL`

(or (= 1 2) (= 3 3)) `==> T`

Цикли

1) DOLIST та DOTIMES

DOLIST проходить по всіх елементах списку, виконуючи тіло циклу зі змінною, що послідовно містить елементи списку:

dolist (var list-form) body-form)

Розглянемо приклад:

(dolist (x' (1 2 3)) (print x))

На екрані отримаємо

1

2

3

NIL

Якщо необхідно перервати цикл DOLIST до закінчення списку, можна використовувати RETURN.

(dolist (x '(1 2 3)) (print x) (if (evenp x) (return)))

На екрані

1

2

NIL

DOTIMES – це конструкція циклу верхнього рівня для циклів з підрахунком.

```
(dotimes (var count-form) body-form)
```

COUNT-FORM має видати ціле число.

VAR містить послідовні цілі числа, починаючи з 0

Розглянемо приклад:

```
(dotimes (i 4) (print i))
```

0

1

2

3

NIL

Так само, як і з DOLIST, можна використовувати RETURN, щоб зупинити цикл раніше.

2) Цикл DO

Цикл DO має вигляд:

```
(do (variable-definition)
    (end-test-form result-form)
  statement)
```

Кожне VARIABLE-DEFINITION (визначення змінної) вводить змінну, яка буде в полі видимості тіла циклу.

Повна форма одного визначення змінної – це список, що містить три елементи:

```
(var init-form step-form)
```

Форма INIT-FORM буде виконана на початку циклу та отримане значення присвоєне змінній VAR.

Перед кожною наступною ітерацією циклу, STEP-FORM буде виконано та її значення присвоєне VAR.

Форма STEP-FORM необов'язкова; якщо її не буде, змінна залишиться з тим самим значенням від ітерації до ітерації, поки їй не буде призначено нове значення в тілі циклу.

Приклад 1:

```
(do ( (n 0 (1+ n))
    (cur 0 next)
    (next 1 (+ cur next))
  )
    ((= 10 n) cur)
)
```

Крокові форми (1+ n), next та (+ cur next) всі обчислюються, використовуючи старі значення n, cur та next. Тільки після обчислення всіх крокових форм, змінні набувають своїх нових значень.

Приклад 2:

```
(do ((i 0 (1+ i)))
  )
    ((>= i 4))
  (print i)
)
```

3) Цикл LOOP

Макрос LOOP буває двох видів: простий та розширений. Проста версія: нескінченний цикл без пов'язаних із ним змінних. Скелет виглядає так:

```
(loop
 body-form)
```

Форми всередині тіла виконуються щоразу в процесі циклу, який триватиме вічно, доки не буде використано RETURN, щоб перервати його.

Наприклад:

```
(loop
 (when (> ab)
 (return)
 )
 .......................
)
```

Розширений LOOP відрізняється використанням спеціальних ключових слів.

Розглянемо наступний приклад. Використовуємо цикл DO:

```
(do ((nums nil) (i 1 (1+ i)))
 )
  ((> i 10) (nreverse nums)
 )
 (push i nums)
)
```

На екрані

(1 2 3 4 5 6 7 8 9 10)

З використанням LOOP цей же код можна записати компактно:

```
(loop for i from 1 to 10 collecting i)
```

1.7 Робота з макросами

Створення макросів

Макроси визначаються за допомогою форми DEFMACRO**.** Базовий шаблон DEFMACRO має вигляд:

```
(defmacro name (parameter)
```

```
 "Optional documentation string."
 body-form)
```

Подібно до функцій, макрос складається з імені, списку параметрів, необов'язкового рядка документації і тіла, що складається з виразів Lisp. Однак роботою макросу не є здійснення якоїсь дії безпосередньо, – він генерує код, який потім щось робитиме.

Розглянемо приклад макросу. Для цього введемо дві допоміжні функції:

```
(defun primep (number)
  (when (> number 1)
    (loop for fac from 2 to (sqrt number) never
                                                    (zerop (mod number fac))
    )
  )
)
```

Дана функція перевіряє, чи є дане число простим.

```
(defun next-prime (number)
 (loop for n from number when (primep n)
 return n))
```

Дана функція повертає наступне просте число, що більше або дорівнює її аргументу.

Припустимо, необхідний макрос, який можна викликати так:

```
(do-primes (p 0 19) (format t "~d" p))
```

і який еквівалентний наступному коду:

```
(do (
 (p (next-prime 0)
 (next-prime (1+ p))
 )
 )
 ((> p 19)) (format t "~d" p)
)
```

Таким чином, макрос має вигляд:

```
(defmacro do-primes ((var start end) &body body)
   `(do (
 (,var (next-prime,start) (next-prime (1+,var)))
 )
  ((>, var, end)), @ body
 )
)
```

Усередині списку параметрів, що деструктуруються, просте ім'я параметра може бути замінене вкладеним списком параметрів. Параметри в такому списку будуть отримувати свої значення з елементів виразу, які були б пов'язані з параметром, заміненим цим списком.

Іншою особливістю списку параметрів макросів є те, що можна використовувати &BODY як синонім &REST.

Список параметрів, що деструктуруються, також надає автоматичну перевірку помилок: при визначенні таким чином do-primes, Lisp буде здатний визначати виклики, в яких перший аргумент не є триелементним списком, і видавати адекватні повідомлення про помилки

Списки параметрів, що деструктуруються, можуть містити параметри &OPTIONAL, &KEY та &REST.

Після деструктурування аргументів, все, що залишається зробити, – це підставити їх у шаблон для отримання розкриття.

Для простих макросів, на кшталт do-primes, найкращим варіантом є використання спеціального синтаксису квазіцитування. Вирази квазіцитування подібні до виразів цитування, за винятком того, що можна прибрати лапки з певних підвиразів, записавши попереду, за якою можливо слідує знак «at» (@). Без «at» кома викликає включення як є значення наступного за нею подвиразу. Зі знаком «at» значення, яке має бути списком, «вклеюється» до навколишнього списку. Розглянемо приклад синтаксису квазіцитування:

```
`(a (+ 1 2) c)              (a (+ 1 2) c)
`(a ,(+ 1 2) c)             (a 3 c)
`(a (list 1 2) c)           (a (list 1 2) c)
`(a ,(list 1 2) c)          (a (1 2) c)
`(a ,@(list 1 2) c)         (a 1 2 c)
```

Макрос do-primes без квазіцитування матиме такий вигляд:

```
(defmacro do-primes-a ((var start end) &body body)
 (append'(do)
 (list (list (list var
 (list 'next-prime start)
 (list 'next-prime (list '1+ var))
 )
  )
 )
 (list (list (list '> var end))) body
 )
)
```

Макроси, що створюють макроси

Можна написати макрос, що створює макрос: макрос, що генерує код, який генерує код. Розглянемо приклад:

```
(defmacro do-primes ((var start end) &body body)
 (with-gensyms (ending-value-name)
 `(do ((,var (next-prime ,start)) (next-prime (1+,var))
 )
 (, ending-value-name, end)
 )
```

```
 ((> ,var ,ending-value-name)) , @body
 )
 )
)
```

Тут with-gensyms є макросом, який має вигляд:

```
(defmacro with-gensyms ((&rest names) &body body)
 `(let ,(loop for n in names collect `(,n (gensym)))
 , @ Body
 )
)
```

1.8 Робота з рядками та масивами

Рядки

Рядки в Common Lisp є складеними типами даних – одномірними масивами букв.

Рядки записуються в подвійних лапках. Можна включити в рядок будь-який знак з підтримуваного набору, за винятком подвійних лапок (“) та зворотного слеша (\). Можна також увімкнути ці два символи, якщо їх екранувати за допомогою зворотного слеша. Знак зворотний слеш завжди маскує наступний знак, незалежно від цього. Розглянемо приклади:

`"foobar"` звичайний рядок.

`"foo\\bar"` перший зворотний слеш екранує другий.

`"foo\bar"` зворотний слеш екранує знак b

Порівняння рядків

Функції порівняння базуються на функціях порівняння символів, які мають вигляд:

= CHAR-EQUAL

/= CHAR-NOT-EQUAL

<   CHAR-LESSP

>   CHAR-GREATERP

<= CHAR-NOT-GREATERP

>= CHAR-NOT-LESSP

У Common Lisp використовується кодування Unicode.

Функції порівняння рядків мають вигляд:

=   STRING-EQUAL

/= STRING-NOT-EQUAL

<   STRING-LESSP

>   STRING-GREATERP

<= STRING-NOT-GREATERP

>= STRING-NOT-LESSP

Функції порівняння, за винятком STRING-EQUAL, повертають індекс позиції у першому рядку, де рядки починають відрізнятися. Наприклад:

```
(STRING-NOT-EQUAL "lisp" "lissome")==> 4
```

Масиви в Lisp

Для визначення масиву заданої розмірності використовується функція MAKE-ARRAY, причому підтримуються лише одновимірні масиви-вектори:

```
(make-array <розмірність>)
```

Приклад:

```
(setq data (make-array 10))
```

На екрані отримаємо

```
(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
```

Тут data – ім'я масиву, NIL – початкове наповнення.

Доступ здійснюється за допомогою функції AREF. Вона має два аргументи – ім'я масиву та індекс, і повертає значення комірки:

```
(aref <ім'я> <індекс> )
(aref data 8)
```

На екрані

```
NIL,
```

Особливості: перша комірка має номер 0.

Помістити дані в масив можна, використовуючи функцію SETF:

```
(setf (aref data 2) 'dog)
```

Тут AREF викликає значення комірки, а функція SETF поміщає значення.

1.9 ООП та Lisp. Методи

Об'єктно-орієнтоване програмування

Поза сімейством мов Lisp, майже всі об'єктно-орієнтовані мови наслідують базовий дизайн, заданий мовою Simula, коли поведінка, пов'язана з класом, реалізується у вигляді методів або функцій-членів, які відносяться до певного класу. У цих мовах метод, викликаний для певного об'єкта, і клас, до якого цей об'єкт належить, визначають, який код буде запущено. Така модель називається **передачею повідомлень**.

Концептуально, виклик методів розпочинається з відправки повідомлення, що містить ім'я методу, що запускається, і необхідних аргументів екземпляру об'єкта, метод якого викликається.

Об'єкт потім використовує свій клас для пошуку методу, пов'язаного з іменем, зазначеним у повідомленні, та викликає його. Оскільки кожен клас може мати власний метод для заданого імені, то однакове повідомлення, надіслане різним об'єктам, може викликати різні методи.

Ранні реалізації об'єктної системи Lisp працювали аналогічним чином, надаючи спеціальну функцію SEND, яка могла бути використана для надсилання повідомлення певному об'єкту. Синтаксично виклик методу записувався як:

```
(send object 'foo)
```

замість:

```
(foo object)
```

Більш важливим було те, що оскільки методи не були функціями, вони не могли бути передані як аргументи для функцій вищого порядку, наприклад, MAPCAR; якщо необхідно викликати метод для всіх елементів списку, використовуючи MAPCAR, необхідно було написати:

```
(mapcar #'(lambda (object) (send object 'foo)) objects)
```

замість

```
(mapcar #'foo objects)
```

У сучасному Lisp уніфікували методи та функції шляхом введення нового їх вигляду, названого узагальненим (generic).

Узагальнені функції та методи

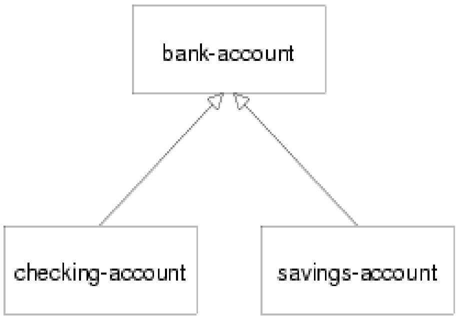
Узагальнена функція визначає абстрактну операцію, вказуючи ім'я операції та перелік параметрів, але не визначає реалізацію. Розглянемо приклад:

```
(defgeneric draw (shape)
 (:documentation "Натисніть Given shape on the
 screen.")
)
```

Функція є узагальненою в тому сенсі, що вона може приймати як аргументи будь-які об'єкти. Однак сама ця функція не робить нічого; при виклику з будь-якими аргументами вона видаватиме помилку.

Дійсна реалізація узагальненої функції забезпечується методами. Кожен метод надає реалізацію узагальненої функції окремих класів аргументів.

Розглянемо застосування узагальнених функцій. Нехай є клас bank-account, і він має два підкласи – checking-account та savings-account.
Ієрархія класів виглядає наступним чином:



Створимо узагальнену функцію withdraw, яка зменшує баланс на зазначену суму. Якщо баланс менший цієї суми, вона повинна видати помилку і залишити баланс у незмінному вигляді.

Основна форма DEFGENERIC схожа на DEFUN за винятком, що немає тіла функції. Список параметрів DEFGENERIC визначає параметри, які повинні прийматись усіма методами, визначеними для даної узагальненої функції.

Замість тіла DEFGENERIC може містити різноманітні опції. Однією з опцій, яку необхідно завжди вказувати, є :DOCUMENTATION, яка використовується для вказування рядка з описом призначення узагальненої функції.

```
(defgeneric withdraw (account amount)
 (:documentation "Знижка
 the account. Signal an error if the current balance is
less than
 amount.")
)
```

Список параметрів методу, що реалізує узагальнену функцію, має бути аналогічним його узагальненій функції. У даному випадку це означає, що всі методи, визначені для withdraw, повинні мати два обов'язкові параметри.

Іншими словами, методи повинні мати таку ж кількість обов'язкових і необов'язкових параметрів, і крім цього, повинні вміти приймати будь-які аргументи, що відносяться до залишкових (&REST) або іменованих (&KEY) параметрів, визначених в узагальненій функції.

Оскільки базові дії зі списання грошей з рахунку є однаковими для всіх рахунків, можна визначити метод, який спеціалізує параметр account для класу bank-account:

```
(defmethod withdraw ((account bank-account) amount)
 (when (< (balance account) amount)
 (error "Account overdrawn.")
 )
 (decf (balance account) amount)
)
```

Функція balance повертає поточне значення суми на рахунку.

Форма DEFMETHOD більше схожа на DEFUN у порівнянні з DEFGENERIC.

Основною відмінністю є те, що необхідні параметри можуть бути спеціалізовані шляхом заміни імені параметра на список двох елементів. Першим елементом є ім'я параметра, а другим – спеціалізатор, який, наприклад, може бути ім'ям класу. Ім'я параметра може бути будь-яким – воно не обов'язково має збігатися з іменем, вказаним в оголошенні узагальненої функції.

Цей метод буде використовуватися тоді, коли перший аргумент withdraw є екземпляром класу bank-account. Другий параметр amount спеціалізується неявно.

DEFMETHOD також дозволяє створювати методи, які спеціалізовані для конкретного об'єкта, використовуючи EQL-спеціалізатор. Розглянемо приклад:

```
(defmethod withdraw ((account (eql account-of-bank-president ))
 amount)
 (let ((overdraft (- amount (balance account))))
```

```
 (when (plusp overdraft)
 (incf (balance account) overdraft)
 )
 )
 (call-next-method)
)
```

Форма, вказана в EQL-спеціалізаторі, який використовується, щоб вказати об’єкт (у нашому випадку це account-of-bank-president), обчислюється один раз, при обчисленні DEFMETHOD.

Визначимо метод, що реалізує узагальнену функцію withdraw для класу checking-account:

```
(defmethod withdraw ((account checking-account) amount)
 (let ((overdraft (- amount (balance account))))
 (when (plusp overdraft)
 (withdraw (overdraft-account account) overdraft)
 (incf (balance account) overdraft)
 )
 )
 (call-next-method)
)
```

Функція CALL-NEXT-METHOD є частиною системи узагальнених функцій та використовується для комбінації відповідних методів. Вона повідомляє, що контроль має бути переданий від поточного методу до методу, спеціалізованого для bank-account.

Стандартний комбінатор методів

Відповідно до концепції, ефективний метод будується за три кроки:

- На першому кроці узагальнена функція будує список методів, для яких застосовні передані аргументи.

- На другому кроці отриманий список методів сортується відповідно до спеціалізованих параметрів.
- На третьому кроці методи по порядку беруться зі списку та їх код комбінується, утворюючи ефективний метод.

Для знаходження методів, для яких застосовні ці аргументи, узагальнена функція порівнює аргументи з відповідними спеціалізаторами параметрів усіх визначених методів.

Метод вважається допустимим, лише якщо всі спеціалізатори сумісні з відповідними параметрами.

Коли спеціалізатор є іменем класу, він вважається сумісним, якщо вказане ім'я збігається з іменем класу аргументу (або іменем одного із суперкласів аргументу). EQL-спеціалізатор вважається сумісним, якщо аргумент є тим же об'єктом, що вказаний у спеціалізаторі.

Після того, як знайдено всі відповідні методи, необхідно відсортувати їх, щоб потім скомбінувати в ефективний метод. При впорядкуванні двох методів узагальнена функція порівнює їх спеціалізатори параметрів зліва направо, і перший спеціалізатор, який відрізняється у списку параметрів методів, визначатиме їх порядок: першим ставиться метод з більш специфічним спеціалізатором.

За замовчуванням узагальнені функції використовують так званий стандартний комбінатор методів. Він поєднує методи таким чином, що CALL-NEXT-METHOD працює так: спочатку запускається найбільш специфічний метод, при цьому кожен метод може передати керування наступному методу, використовуючи CALL-NEXT-METHOD.

Стандартний комбінатор методів також підтримує три види допоміжних методів: :BEFORE, :AFTER та :AROUND. Визначення допоміжних методів записується за допомогою DEFMETHOD так само, як і для основних, але крім цього між іменем методу і списком параметрів вказується кваліфікатор методу, який іменує його тип.

Розглянемо приклад. Метод :BEFORE для функції withdraw, який спеціалізує параметр account для класу bank-account, буде починатися з наступного рядка:

```
(defmethod withdraw :before ((account bank-account) amount) ...)
```

Методи :BEFORE можуть бути використані для виконання будь-якої підготовчої роботи, яка може знадобитися, щоб переконатися, що основний метод може бути виконаний. Усі застосовні методи :BEFORE (і не тільки найбільш специфічні) запускаються як частина ефективного методу. Вони запускаються, починаючи з самого специфічного методу, та (цілком відповідаючи назві) до найбільш специфічного основного методу.

Наприклад, можна використовувати метод :BEFORE, спеціалізований для checking-account для реалізації захисту від перевитрати для чекових рахунків:

```
(defmethod withdraw :before ((account checking-account)
 amount)
  (let ((overdraft (- amount (balance account))))
  (when (plusp overdraft)
  (withdraw (overdraft-account account) overdraft)
  (incf (balance account) overdraft)
  )
  )
)
```

Цей метод :BEFORE має три переваги, порівняно з використанням основного методу.

По-перше, він явно показує, як метод змінює загальну поведінку функції withdraw, – він не перетинається з основною поведінкою і не змінює значення, що повертається.

Наступною перевагою є те, що головний метод, спеціалізований для класу специфічнішого, ніж checking-account, не буде суперечити методу

:BEFORE, що дозволяє автору підкласу checking-account легше розширити поведінку withdraw, зберігаючи при цьому стару функціональність.

Третя перевага: оскільки метод :BEFORE не повинен викликати CALL-NEXT-METHOD для передачі управління методам, що залишилися, немає можливості зробити помилку, забувши вказати цю функцію.

Інші допоміжні методи також організуються в ефективні методи засобами, які відображені в їх іменах.

Усі методи :AFTER запускаються після основного методу в такому порядку, що найбільш специфічний метод викликається останнім (тобто, порядок протилежний виклику методів :BEFORE).

Таким чином, методи :BEFORE і :AFTER комбінуються для створення вкладених обгорток навколо головної функціональності, що забезпечується основними методами: кожен більш специфічний метод :BEFORE може зробити щось для менш специфічного методу :BEFORE, основні методи можуть успішно виконуватися, а потім кожен більш специфічний метод :AFTER може прибрати щось за основним методом і менш специфічними методами :AFTER.

Методи :AROUND комбінуються майже так само, як основні, за винятком того, що вони виконуються «навколо» інших методів. Таким чином, код найбільш специфічного методу :AROUND запускається до будь-якого іншого коду.

Отже, усередині коду методу :AROUND, виклик CALL-NEXT-METHOD призведе до того, що виконуватиметься код наступного методу :AROUND, або, при виклику з найменш специфічного методу :AROUND, призведе до виконання ланцюжка методів :BEFORE, основного методу і потім методів :AFTER. Усі методи :AROUND матимуть у своєму коді виклик CALL-NEXT-METHOD. Розглянемо приклад:

```
(defmethod key-input :around (key-name)
 (format nil "~:(~a~) key pressed" (call-next-method key-name)
```

```
 )
)
defmethod key-input ((key-name (eql :escape))) "escape")
```

Під час виклику KEY-INPUT відбувається наступне:

- викликається метод із міткою :AROUND,
- він викликає наступний метод, тобто одну з спеціалізованих версій KEY-INPUT яка повертає рядок, і цей рядок форматує метод з :AROUND.

Комбінатори методів

На додаток до стандартного комбінатора методів, мова Lisp реалізує дев'ять інших вбудованих комбінаторів методів. Вони мають імена від операторів: +, AND, OR, LIST, APPEND, NCONC, MIN, MAX та PROGN.

Прості комбінатори підтримують лише два типи методів – основні методи та методи: AROUND.

Наприклад, узагальнена функція, яка використовує комбінатор методів +, поверне суму всіх результатів, повернутих викликаними основними методами.

Комбінатори AND і OR не обов'язково виконуватимуть усі основні методи, оскільки ці макроси можуть використовувати скорочену схему роботи – узагальнена функція, яка використовує комбінатор AND, поверне значення NIL, як тільки один із методів поверне його, або інакше поверне значення, повернене останнім викликаним методом.

Для визначення узагальненої функції, яка використовує конкретний комбінатор методів, необхідно вказати опцію :METHOD-COMBINATION при оголошенні DEFGENERIC. Значення, вказане цією опцією, визначає ім'я комбінатора методів, яке буде використовуватися. Розглянемо приклад:

```
(defgeneric priority (job)
  (:documentation "Return the priority at which the
```

```
job should be run.")
 (:method-combination +))
```

За замовчуванням, всі комбінатори методів комбінують методи в порядку, починаючи з найбільш специфічного. Однак можна змінити порядок, вказавши ключове слово :MOST-SPECIFIC-LAST після імені комбінатора в оголошенні функції за допомогою DEFGENERIC. Приклад:

```
(defgeneric priority (job)
  (:method-combination + :most-specific-last))
```

Основні методи узагальнених функцій, які використовують один із цих комбінаторів, мають бути кваліфіковані іменем комбінатора методів. Наприклад:

```
(defmethod priority + ((job express-job)) 10)
```

Всі прості вбудовані комбінатори методів підтримують методи :AROUND, які працюють так само, як і методи :AROUND у стандартному комбінаторі: найбільш специфічний метод :AROUND виконується до будь-якого методу, і він може використовувати CALL-NEXT-METHOD для передачі контролю менш специфічному методу :AROUND до тих пір, поки не буде досягнуто основного методу. Опція :MOST-SPECIFIC-LAST не впливає порядок виклику методів :AROUND.

Вбудовані комбінатори методів не підтримують методи :BEFORE і :AFTER.

Мультиметоди

Методи, які явно спеціалізують більше одного параметра узагальненої функції, називаються мультиметодами.

Мультиметоди корисні у ситуаціях, коли невідомо, якого класу має віноситися певна поведінка. Наприклад, звук, що видає барабан, коли по ньому стукають паличкою, це функція барабана чи палички? Очевидно, він належить обом предметам.

Розглянемо узагальнену функцію beat:

```
(defgeneric beat (drum stick))
```

Тепер визначимо різні мультиметоди:

```
(defmethod beat ((drum snare-drum) (stick wooden-drumstick)) ...)
(defmethod beat ((drum snare-drum) (stick brush)) ...)
(defmethod beat ((drum snare-drum) (stick soft-mallet)) ...)
(defmethod beat ((drum tom-tom) (stick wooden-drumstick)) ...)
(defmethod beat ((drum tom-tom) (stick brush)) ...)
(defmethod beat ((drum tom-tom) (stick soft-mallet)) ...)
```

Мультиметоди також рятують від вимоги, щоб один набір класів був тісно пов'язаний з іншим набором. У цьому прикладі барабанів/паличок, не потрібно, щоб клас, що представляє барабани, знав про різні класи, що представляють барабанні палички, і навпаки.

1.10 ООП та Lisp. Класи

Класи

Створити власний клас можна за допомогою макросу DEFCLASS. Оскільки поведінка класу визначається узагальненими функціями і спеціалізованими методами для класу, то DEFCLASS відповідає лише за визначення класу як типу даних.

Клас як тип даних складається з трьох частин: імені, відношення до інших класів та імен слотів (в інших об'єктно-орієнтованих мовах, слоти можуть називатися полями, змінними членами класу або атрибутами).

Базова форма DEFCLASS має вигляд:

```
(defclass name (direct-superclass-name)
 (slot-specifier)
)
```

Імена класів знаходяться у власному просторі імен, окремо від імен функцій та змінних, тому можна задати для класу те саме ім'я, що й існуюча функція та змінна.

Опція DIRECT-SUPERCLASS-NAMES використовується, щоб вказати імена класів, від яких проводитиметься спадкування цього класу. Якщо жодного класу не вказано, то він буде успадкований від STANDARD-OBJECT. Всі класи, зазначені в даній опції, повинні бути класами, визначеними користувачем, щоб бути впевненим, що кожен новий клас походить від STANDARD-OBJECT.

Наприклад, запис DEFCLASS для деяких класів, може виглядати так:

```
(defclass bank-account () ...)
(defclass checking-account (bank-account)...)
(defclass savings-account (bank-account)...)
```

Специфікатори слотів

Специфікатор слота, як мінімум, вказує його ім'я, так що специфікатор може бути простим іменем. Наприклад:

```
(defclass bank-account ()
 (customer-name balance)
)
```

Кожен екземпляр цього класу містить два слоти: один для зберігання імені клієнта, а другий – для зберігання поточного балансу рахунку. Використовуючи це визначення, можна створити нові об'єкти bank-account за допомогою MAKE-INSTANCE:

```
(make-instance 'bank-account)
```

Аргументом MAKE-INSTANCE є ім'я класу, а значенням, що повертається - новий об'єкт.

Використовуючи це визначення bank-account, нові об'єкти будуть створюватися зі слотами, які не пов'язані зі значеннями. Будь-яка спроба отримати значення для незв'язаного слота призведе до видачі помилки, так що необхідно задати значення до того, як воно буде зчитуватися.

Розглянемо приклади:

```
(defparameter account (make-instance 'bank-account)) ==>
ACCOUNT
(setf (slot-value account 'customer-name) "John Doe") ==>
"John Doe"
(setf (slot-value account 'balance) 1000) ==> 1000
```

Тепер можна отримувати значення слотів:

```
(slot-value account 'customer-name) ==> "John Doe"
(slot-value account 'balance) ==> 1000
```

Ініціалізація об'єкту

Common Lisp надає ряд способів керування початковими значеннями слотів. Наприклад, можна додати опції в специфікатори слотів DEFCLASS: за допомогою опції :INITARG можна вказати ім'я, яке потім буде використовуватися як іменований параметр при виклику MAKE-INSTANCE, і передане значення буде збережено в слоті.

Інша опція – :INITFORM – дозволяє вказати вираз на Lisp, який використовуватиметься для обчислення значення, якщо при виклику MAKE-INSTANCE не було передано аргумент :INITARG.

Для повного контролю за ініціалізацією об'єкта можна також визначити метод для узагальненої функції INITIALIZE-INSTANCE, яку викликає MAKE-INSTANCE.

Специфікатор слота, який включає опції, такі як :INITARG або :INITFORM, записується як список, що починається з імені слота, за яким йдуть опції.

Розглянемо приклад:

```
(defclass bank-account ()
 (
 (customer-name: initarg: customer-name)
 (balance: initarg: balance: initform 0)
```

```
 )
)
```

Тепер можна одночасно створювати рахунок та вказувати значення слотів:

```
(defparameter account
 (make-instance 'bank-account :customer-name
 "John Doe" :balance 1000))
```

Також можна отримати значення параметрів:

```
(slot-value account 'customer-name) ==> "John Doe"
(slot-value account 'balance) ==> 1000
```

Якщо при виклику MAKE-INSTANCE не буде передано аргумент :BALANCE, то виклик SLOT-VALUE для слота balance буде отриманий обчисленням форми, вказаної опцією :initform:

```
(slot-value (make-instance 'bank-account) 'balance) ==> 0
(slot-value (make-instance 'bank-account) 'customer-name) ==>
error
```

Якщо необхідно переконатися, що ім'я клієнта було задано під час створення рахунку, то можна видати помилку в початковому виразі (INITFORM), оскільки воно буде обчислене, тільки якщо початкове значення (INITARG) не було задано:

```
(defvar account-numbers 0)
(defclass bank-account ()
 (
 (customer-name :initarg :customer-name
 :initform (error "Must supply a customer name."))
 (balance: initarg: balance: initform 0)
 (account-number :initform (incf account-numbers))
 )
)
```

Метод INITIALIZE-INSTANCE, спеціалізований для STANDARD-OBJECT, забезпечує ініціалізацію слотів, виходячи з даних, заданих опціями :INITARG й :INITFORM.

Розглянемо приклад. Додамо слот account-type до класу bank-account:

```
(defclass bank-account ()
 ((customer-name :initarg :customer-name
 :initform (error "Must supply a customer name."))
 (balance: initarg: balance: initform 0)
   (account-number :initform (incf account-numbers))
 account-type)
)
```

Після цього можна визначити метод :AFTER для INITIALIZE-INSTANCE, який встановить значення слота account-type на основі значення, яке було збережено в слоті balance:

```
(defmethod initialize-instance :after ((account
                                        bank-account) &key)
 (let ((balance (slot-value account 'balance)))
 (setf (slot-value account 'account-type)
 (cond
 ((>= balance 100000) :gold)
 ((>= balance 50000) :silver)
 (t: bronze)
 )
 )
 )
)
```

Інструкція &KEY у списку параметрів потрібна, щоб зберегти список параметрів подібним до відповідного списку параметрів узагальненої функції – список параметрів, вказаний для функції INITIALIZE-INSTANCE, включає

&KEY, щоб дозволити окремим методам передавати власні іменовані параметри, але при цьому він не вимагає вказування конкретних назв. Таким чином, кожен метод повинен вказувати &KEY, навіть якщо він не вказує жодного іменованого параметра.

Якщо метод INITIALIZE-INSTANCE, спеціалізований для конкретного класу, вказує іменований параметр, цей параметр стає допустимим параметром для функції MAKE-INSTANCE під час створення екземпляра даного класу. Розглянемо приклад:

```
(defmethod initialize-instance :after ((account bank-account)
                                   &key opening-bonus-percentage)
 (when opening-bonus-percentage
 (incf (slot-value account 'balance)
 (* (slot-value account 'balance) (/ opening-
bonus-percentage 100))
 )
 )
)
```

Тоді під час створення об'єкта класу bank-account отримаємо:

```
(defparameter acct (make-instance
 'bank-account
 :customer-name "Sally Sue″
 :balance 1000
 :opening-bonus-percentage 5))
(slot-value acct 'balance) ==> 1050
```

Функції доступу

Розглянемо реалізацію інкапсуляції в Lisp.

Наприклад, визначення функції, яка читає вміст слота balance, має вигляд:

```
(defun balance (account) (slot-value account 'balance))
```

Проте, якщо будуть визначатися підкласи для bank-account, тоді можна визначити balance як узагальнену функцію. Маємо:

```
(defgeneric balance (account))
(defmethod balance ((account bank-account))
     (slot-value account 'balance))
```

Якщо необхідно заборонити користувачу встановлювати баланс безпосередньо, необхідно визначити функцію SETF. Розглянемо визначення SETF для встановлення значення слота customer-name у класі bank-account:

```
(defun (setf customer-name) (name account)
 (setf (slot-value account 'customer-name) name))
```

Тоді, у коді

```
 (setf (customer-name my-account) "Sally Sue")
```

буде викликано функцію SETF, яка визначена вище.

Найкращим рішенням буде визначити SETF як узагальнену функцію:

```
(defgeneric (setf customer-name) (value account))
(defmethod (setf customer-name) (value (account bank-account))
(setf (slot-value account 'customer-name) value))
```

Можна також визначити функцію читання customer-name:

```
(defgeneric customer-name (account))
(defmethod customer-name ((account bank-account))
 (slot-value account 'customer-name)
)
```

Тоді можна написати

```
(setf (customer-name account) "Sally Sue") ==> "Sally Sue"
(customer-name account) ==> "Sally Sue"
```

DEFCLASS підтримує три опції для слотів, які дозволяють автоматично створювати функції читання та запису значень окремих слотів.

Опція :READER вказує ім'я, що буде використовуватися як ім'я узагальненої функції, яка приймає об'єкт як єдиний аргумент. Коли обчислюється DEFCLASS, створюється відповідна узагальнена функція. Після цього для даної узагальненої функції створюється метод, спеціалізований для нового класу, що повертає значення слота. Ім'я функції може бути будь-яким, але зазвичай використовують те саме ім'я, що має сам слот. Розглянемо приклад:

```
 (balance
 :initarg :balance
 :initform 0
 :reader balance)
```

Опція :WRITER використовується для створення узагальненої функції та методу для встановлення значення слота. Функція, що створюється, і метод дотримуються вимог для SETF-функції, отримуючи нове значення як перший аргумент і повертаючи його як результат, так що можна визначити SETF-функцію, задаючи ім'я, таке як (setf customer-name):

```
(customer-name
 :initarg :customer-name
 :initform (error "Must supply a customer name.")
 :reader customer-name
 :writer (setf customer-name)
)
```

Оскільки досить часто потрібне визначення обох функцій доступу, DEFCLASS також має опцію :ACCESSOR, яка створює і функцію читання, і відповідну SETF-функцію. Розглянемо приклад:

```
(customer-name
 :initarg :customer-name
 :initform (error "Must supply a customer name.")
 :accessor customer-name)
```

Опція :DOCUMENTATION дозволяє встановити рядок, який описує даний слот.

Макроси WITH-SLOTS та WITH-ACCESSORS

Макроси WITH-SLOT і WITH-ACCESSORS створюють блок коду, у якому можуть використовуватись прості імена змінних для звернення до слотів певного об'єкта. WITH-SLOTS надає прямий доступ до слота, так само як при використанні SLOT-VALUE, в той час як WITH-ACCESSORS надає скорочений спосіб виклику функцій доступу.

Базова форма WITH-SLOTS виглядає так:

```
(slots (slot) instance-form body-form)
```

Кожен елемент списку SLOT може бути або іменем слота, яке також буде іменем змінної, або списком двох елементів, де перший аргумент є іменем, яке буде використовуватися як змінна, а другий – іменем відповідного слота.

Вираз INSTANCE-FORM обчислюється один раз для отримання об'єкта, до слотів якого буде доступ. Наприклад:

```
(defmethod assess-low-balance-penalty ((account bank-account))
 (with-slots (balance) account
 (when (< balance minimum-balance)
 (decf balance (balance 0.01))
 )
 )
)
```

У варіанті списку маємо:

```
(defmethod assess-low-balance-penalty ((account bank- account))
 (with-slots ((bal balance)) account
   (when (<bal *minimum-balance*)
    (decf bal (*bal 0.01))
 )
 )
)
```

Форма WITH-ACCESSORS збігається з WITH-SLOTS за винятком, що кожен елемент списку слотів є списком з двох елементів, що містять ім'я змінної та ім'я функції доступу. Всередині тіла WITH-ACCESSORS, посилання на одну із змінних аналогічне виклику відповідної функції доступу. Розглянемо приклад:

```
(defmethod assess-low-balance-penalty ((account bank-account))
 (with-accessors ((balance balance)) account
  (when (< balance minimum-balance)
  (decf balance (* balance 0.01))
 )
 )
)
```

Тут перше входження balance є іменем змінної, а друге – іменем функції доступу; вони не обов'язково мають бути однаковими.

Якщо слот має значення опції, що дорівнює :CLASS, то слот має тільки одне значення, яке зберігається всередині класу, і використовується всіма екземплярами (статичний атрибут). За замовчуванням слоти мають значення :INSTANCE.

Доступ до слотів зі значенням :CLASS здійснюється так само, як і для слотів зі значенням :INSTANCE – за допомогою SLOT-VALUE або функції доступу, що означає, що доступ можна отримати лише через екземпляр класу, хоча це значення не зберігається в цьому екземплярі.

Слоти та спадкування

У Common Lisp конкретний об'єкт може мати лише один слот із певним ім'ям. Однак можливо, що в ієрархії спадкування класу декілька класів матимуть слоти з тим самим ім'ям. Це може статися або тому, що підклас включає специфікатор слота з тим самим іменем, що й слот, вказаний в суперкласі, або тому, що кілька суперкласів мають слоти з одним і тим самим іменем.

Common Lisp вирішує цю проблему шляхом злиття усіх специфікаторів з одним і тим самим іменем з нового класу та всіх його суперкласів для створення окремих специфікаторів для кожного унікального імені слота.

При злитті специфікаторів, різні опції специфікаторів слотів розглядаються по-різному. Наприклад, оскільки слот може мати лише одне значення за замовчуванням і при цьому кілька класів вказують опцію :initform, то новий клас буде використовувати цю опцію з найбільш спеціалізованого класу.

Опції :INITARGS не повинні бути взаємовиключними – кожна опція :INITARG створює іменований параметр, який можна використовувати для ініціалізації слота. Множинні параметри не призводять до конфлікту, тому новий специфікатор слоту міститиме всі опції :INITARGS.

Користувачі, що викликають MAKE-INSTANCE, можуть використовувати будь-яке з імен, зазначених у :INITARGS для ініціалізації слота.

Успадковані опції :READER, :WRITER і :ACCESSOR не входять у новий специфікатор слота, оскільки методи, створені під час оголошення суперкласу, будуть автоматично застосовуватися до нового класу. Однак новий клас може створити свої функції доступу шляхом оголошення власних опцій :READER, :WRITER або :ACCESSOR.

Розглянемо приклад:

```
(defclass foo ()
 (
 (a :initarg :a :initform "A" :accessor a)
 (b :initarg :b :initform "B" :accessor b)
 )
)
(defclass bar (foo)
 (
 (a :initform (error "Must supply a value for a"))
 (b :initarg :the-b :accessor the-b :allocation :class)
 )
```

)

При створенні екземпляра класу bar, можна використовувати успадкований початковий аргумент :A, щоб вказати значення для слоту a. І справді, це необхідно зробити для того, щоб уникнути помилок, оскільки опція :INITFORM визначеня, bar замінює опцію, успадковану від foo.

Множинне спадкування

Common Lisp також підтримує множинне спадкування – клас може мати кілька прямих суперкласів, успадковуючи відповідні методи та специфікатори слотів із усіх цих класів.

Проблема впорядкування класів за специфічністю при множинному спадкуванні в Lisp вирішується наступним чином: Lisp сортує суперкласи, що не відносяться один до одного, по порядку, в якому вони перелічені у визначенні безпосередніх суперкласів в DEFCLASS. При цьому класи, зазначені в списку першими, вважаються більш специфічними за класи, зазначені в списку останніми.

Завдання з Lisp

1. Дано прямокутне поле розміром n×m клітинок. Можна робити кроки на одну комірку вправо, вниз чи по діагоналі вправо-вниз. У кожній клітинці записано деяке натуральне число. Мета – потрапити з верхньої лівої комірки до правої нижньої. Вага маршруту обчислюється як сума чисел із усіх пройдених клітинок. Необхідно знайти маршрут із мінімальною вагою.
2. Створіть програму з графічним інтерфейсом (можна використовувати GCL), яка виконує запити до бази даних за заданою тематикою: домашня відеотеку – фільми, актори, режисери. В якості БД використайте MySQL.

   Для фільмів необхідно зберігати:

   - назву;
   - акторів;
   - дату виходу;
   - країну, в якій випущено фільм.

   Для акторів та режисерів необхідно зберігати:

   - ім'я;
   - дату народження.

   Приклади запитів:

   - знайти всі фільми, що вийшли на екран, починаючи з минулого року,
   - вивести інформацію про акторів, які знімалися у заданому фільмі,
   - вивести інформацію про акторів, які знімалися щонайменше у 2 фільмах,
   - вивести інформацію про акторів, які були режисерами хоча б одного з фільмів.
3. Кожен елемент квадратної матриці розмірності n×n дорівнює нулю або одиниці. Для заданої матриці знайдіть кількість «острівців», утворених

одиницями. «Острівцем» вважається група одиниць (або одна одиниця), з усіх боків оточена нулями (чи краями матриці). Одиниці відносяться до одного «острівця», якщо з однієї з них можна перейти до іншої по одиницях, розташованих в сусідніх комірках. Сусідніми є клітинки, що межують по горизонталі або вертикалі.

4. Напишіть Lisp-програму для ініціалізації масиву об'єктів інформацією з XML-файлу. Для роботи з XML використайте XMLisp.

   Потрібно створити файл XML та відповідне йому DTD-визначення наступної тематики: лікарські засоби. Вкажіть схему XSD. Визначіть клас LISP, що відповідає даному опису.

   Лікарські препарати мають такі властивості.

   - Name – назва препарату.
   - Price – вартість за упаковку (0 – n гривень).
   - Dosage – дозування препарату (мг/день).
   - Visual (має бути декілька) – візуальні характеристики препарату: колір (білий, жовтий, зелений, червоний), консистенція (рідкий, порошкоподібний, твердий), показання до застосування (респіраторні захворювання, захворювання шлунково-кишкового тракту, захворювання опорно-рухового апарату, протизапальні, загальнозміцнюючі тощо).

   Кореневий елемент називається Medicine.

5. Розв’яжіть систему лінійних рівнянь за допомогою нейронної мережі: використайте мережу Хопфілда.

6. Створіть графічний додаток для виведення конуса, що обертається навколо своєї осі, при цьому його основа має рухається по синусоїді «туди-назад» за допомогою клавіатури. Увімкніть освітлення. Використайте бібліотеку cl-opengl.

7. Задача на графи. На підприємстві N різних верстатів послідовно один за одним об'єднані в конвеєр і працює N робітників. Задається матриця

C[N, N], де C[i, j] – продуктивність i-ого робітника на j-му верстаті. Напишіть програму, що визначатиме:

а) на якому верстаті повинен працювати кожен із робітників, щоб продуктивність була максимальною.

б) те саме, але за умови, що верстати розташовані паралельно і виконують однорідні операції.

8. Використовуючи сокети, створіть клієнт-серверний додаток. Сервер має надсилати повідомлення клієнтам, вибраним зі списку. Список зберігається у файлі.
9. Реалізуйте сортування масиву в різних потоках методами Шелла, QuickSort, бульбашки та на основі бінарного дерева (tree sort).
10. Web-програмування на Lisp (використайте вебсервер Hunchentoot). Відсортуйте введений користувачем масив цілих чисел. Числа вводяться через кому.

Частина II. Мова Haskell

2.1 Типи в Haskell

Мова програмування Haskell

Мова програмування Haskell названа на честь американського математика Гаскелла Каррі (Haskell Curry), роботи якого в галузі математичної логіки є базовими для функціонального програмування. Haskell є строго типізованою винятково функціональною мовою, в основі якої безпосередньо лежить лямбда-числення.

Однак система типів мови Haskell відрізняється від систем типів більшості мов, в певній мірі вона є багатшою.

Типи та значення

Оскільки Haskell є чисто функціональною мовою, всі обчислення здійснюються за допомогою обчислення виразів (синтаксичних термів), що повертають значення (абстрактні сутності, які розглядаються як відповіді).

Основні типи в мові Haskell:

- Char – символьний тип,
- Bool – True або False,
- Int – цілочисельний тип; його розмір може бути 32 або 64 біти залежно від архітектури,
- Integer – «нескінченне» ціле,
- Double – дробове число (з плаваючою крапкою).

Кожне значення має з пов’язаний з ним тип. На інтуїтивному рівні можна розглядати типи як множини значень.

Приклади виразів включають як атомарні значення (атоми), такі як ціле число **5**, символ **'a'** і функція **\x -> x+1**, так і структуровані значення, такі як список **[1, 2, 3]** або пара **('b', 4)**.

Так само як вирази позначають значення, вирази типу є синтаксичними термами, які позначають значення типу (або просто типи). Прикладами виразів типу можуть бути атомарні типи **Int** (цілі необмеженого діапазону), **Char** (символи), **Int->Int** (функції, що відображають **Int** в **Int**), а також структуровані типи **[Int]** (однорідний список цілих) та **(Char, Int)** (пара із символу та цілого).

Всі значення в Haskell є сутністю «першого класу»: вони можуть передаватися в функції як аргументи, повертатися з них як результати, розміщуватися в структурах даних тощо.

З іншого боку, типи у Haskell – не «першокласні» сутності. Насправді, типи описують значення; зв'язування деякого значення з його типом називається **типізацією**. Розглянемо приклади:

```
5 :: Int
'a' :: Char
inc :: Int -> Int
[1,2,3] :: [Int]
('b',4) :: (Char,Int)
```

Функції Haskell зазвичай визначаються як набір рівнянь. Наприклад, функція **inc** може бути визначена одним рівнянням:

```
inc n = n + 1
```

Статична система типів мови Haskell визначає формальний зв'язок між типами та значеннями. Статична система типів гарантує, що програми на Haskell «типобезпечні», тобто програміст не допустив будь-якої плутанини в типах. Наприклад, не можна скласти два символи, тому вираз **'a' + 'b'** є неправильно типізованим.

Основна перевага статично типізованих мов загальновідома: всі помилки типізації виявляються під час компіляції. Однак система типів не перехоплює помилки в повному обсязі: вираз **1/0** є вірним за типом, але його обчислення призведе до помилки під час виконання.

Поліморфні типи

До складу Haskell включено також поліморфні типи – типи, які певним чином містять квантор загальності над усіма типами.

Вирази з поліморфними типами за своєю сутністю описують сімейства типів. Наприклад, **(forall a) [a]** – це сімейство типів, що складається з типів списку з елементами із **a**, для будь-якого типу **a**.

Список цілих (наприклад, **[1, 2, 3]**), список символів (**['a', 'b', 'c']**), список цілих списків тощо будуть членами цього сімейства, тоді як **[2,'b']** туди не належатиме, оскільки не існує одного типу, який включав би і **2**, і **'b'**.

Список **[1, 2, 3]** у Haskell насправді є скороченою формою запису для списку **1:(2:(3:[ ]))**, де **[ ]** – це порожній список, а «**:**» – інфіксний оператор, який додає свій перший аргумент на початок свого другого аргументу (деякого списку). (Зауважимо, що оператори «**:**» та «**[ ]**» відповідають **cons** і **nil** у мові Lisp.) Оскільки оператор «:» правоасоціативний, можна також записати цей список як **1:2:3:[ ]**.

Розглянемо приклад функції яка підраховує кількість елементів у списку:

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

Цей приклад підкреслює важливий аспект мови Haskell: **зіставлення зі зразком** (pattern matching).

Ліві частини рівнянь містять зразки, такі як **[ ]** і **x: xs**.

При застосуванні функції ці зразки зіставляються з фактичними параметрами інтуїтивно зрозумілим способом (**[ ]** відповідає тільки порожньому списку, а **x: xs** буде успішно зіставлений з будь-яким списком, що містить хоча б один елемент, при цьому **x** зв'язується з першим елементом, а **xs** – з рештою списку).

Якщо співставлення пройшло успішно, права частина рівності обчислюється і повертається як результат застосування функції. Якщо зіставлення не вдале, пробується наступне рівняння, а якщо всі рівняння призведуть до невдалого співставлення, то результатом буде помилка.

Функція **length** також є прикладом поліморфної функції. Вона може бути застосована до списку, що містить елементи довільного типу, наприклад

```
length [1,2,3] => 3
length ['a','b','c'] => 3
length [[1],[2],[3]] => 3
```

Для роботи зі списками в Haskell існують вбудовані функції. Наприклад, функція **head** повертає перший елемент списку, а функція **tail** повертає все, крім першого елемента.

Можна помітити, що деякі типи, строго кажучи, є більш загальними за інші, у тому сенсі, що множина заданих ними значень ширша.

Наприклад, тип **[a]** є більш загальним, ніж **[Char]**. Іншими словами, останній може бути виведений з першого за допомогою відповідної підстановки для **a.**

Відносно подібного порядку узагальнення, система типів Haskell має дві важливі властивості:

- для будь-якого правильно типізованого виразу гарантується існування єдиного **основного типу**,
- основний тип може бути виведений автоматично.

Основний тип виразу або функції – це найзагальніший тип, який «містить всі екземпляри виразу». Наприклад, основний тип функції **head** – це **[a] -> a**; тип **a** є вірним, але надлишково загальним, у той час як типи на кшталт **[Int] -> Int** занадто часткові.

Існування та єдиність основного типу є відмінною ознакою системи типів **Хіндлі-Мілнера**, яка лежить в основі системи типів Haskell.

Типи, що визначаються користувачем

Можна визначати власні типи в Haskell за допомогою оголошення **data**. Розглянемо приклади.

```
data Nat = Zero | Succ Nat
```

Маємо один конструктор-примітив Zero і один конструктор Succ, за допомогою якого можна створювати складені значення.

Визначивши тип Nat таким чином, ми говоримо, що значення типу **Nat** можуть бути тільки такими:

```
Zero, Succ Zero, Succ (Succ Zero),
                 Succ (Succ (Succ Zero)), ...
```

Тобто всі значення є ланцюжками Succ із Zero на кінці.

Якщо спробувати десь побудувати значення, яке не відповідає визначеному вище типу, буде помилка компіляції, тобто програма не пройде перевірку типів.

Таким чином типи описують множину допустимих значень. Так, наступні значення неприпустимі для **Nat**:

```
Succ Zero Zero, Succ Succ, True, Zero (Zero Succ), ...
```

Структура типів алгебри даних

Для конструювання нових типів існує лише дві операції: сума та добуток. Розглянемо для початку два крайні випадки.

Тільки добуток типів:

```
data T = Name T1 T2 ... TN
```

Тут значення нового типу T складається із значень типів T_1, T_2, …, T_N, й існує лише один спосіб скласти значення цього типу.

Єдине, що можна зробити, це застосувати до значень типів T_i конструктор **Name**. Приклад:

```
data Time = Time Hour Second Minute
```

Тільки сума типів:

```
Data T = Name1 | Name2 | ... | NameN
```

Тут у нового типу T може лише декілька значень, і вони перераховуються в альтернативах через знак «|». Приклад:

```
data Bool = True | False
```

За такою ж схемою можна визначити наступний колірний тип:

```
data Color = Red | Green | Blue | Indigo | Violet
```

І **Bool**, і **Color** складаються зі скінченного числа конструкторів даних.

Розглянемо приклад:

```
data Color = Red | Green deriving (Eq)
ffunc :: Color -> Int
ffunc (x) = if x == Red then 1 else 2
```

Слід зазначити: кожен добуток типів визначає новий конструктор. Кількість конструкторів у типі дорівнює числу альтернатив. Так, у першому випадку була одна альтернатива, і отже, був лише один конструктор Name.

Імена конструкторів мають бути унікальними в межах модуля. Це говорить про те, що за іменем конструктора компілятор «знає», значення якого типу він може побудувати.

Добуток типів складається з конструктора, за яким через проміжок йдуть підтипи. Така структура невипадкова, вона копіює структуру функції. В якості імені функції виступає конструктор, а в якості аргументу – значення заданих у добутку підтипів. Функція-конструктор після застосування «обгортає» значення аргументів і створює нове значення.

За рахунок цього можна визначати типи по-іншому: визначити їх у стилі класів типів:

```
data Bool where
    True :: Bool
    False :: Bool
```

Тут «класом» є **Bool**, у якого два методи.

У такому ж стилі можна визначити **Nat**:

```
data Nat where
    Zero :: Nat
    Succ :: Nat -> Nat
```

Аналогічно можна визначати і списки:

```
data [a] where
```

```
[] :: [a]
(:) :: a -> [a] -> [a]
```

Конструктор порожнього списку «**[ ]**» є константою, а конструктор об'єднання елемента зі списком «(:)» є функцією.

Однак варто зауважити, що **[ ]** у цьому сенсі також є конструктором типу. Для будь-якого типу **t** ми можемо «застосувати» **[ ]**, щоб створити новий тип **[t]**.

Синтаксис мови Haskell дозволяє записати **[ ] t** як **[t]**. Аналогічно, **->** теж є конструктором типу: для будь-яких двох типів **t** і **u**, **t -> u** є типом функції, що відображає елементи типу **t** у елементи типу **u**.

Наступний приклад вводить тип, у якого лише один конструктор даних:

```
data Point a = Pt a a
```

(Тут тип бінарного конструктора даних **Pt** визначений як **a -> a -> Point a**).

Через єдиність конструктора такі типи, як **Point**, часто називають **типами кортежів** (tuple), оскільки вони по суті представляють декартів добуток (в нашому випадку бінарний) інших типів (кортежі певним чином відповідають поняттю запису (record) в інших мовах).

Слід зауважити, що **Point** – це приклад поліморфного типу: для будь-якого типу **t** він визначає тип декартової точки, яка використовує **t** як тип координат. З цих міркувань зрозуміло, що тип **Point** є унарним конструктором типу, оскільки для даного типу **t** він конструює новий тип **Point t.**

Важливо розрізняти застосування **конструктора даних**, що повертає значення, і застосування **конструктора типу**, що повертає тип: перше відбувається під час виконання і є способом, яким здійснюються обчислення в Haskell, тоді як останнє відбувається під час компіляції і є частиною процесу, в ході якого система типів забезпечує «типобезпеку».

Рекурсивні типи

Типи можуть бути рекурсивними, як у випадку типу двійкових дерев:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

Тут визначається поліморфний тип бінарного дерева, елементами якого є або вузли-листи, що містять значення типу a, або внутрішні вузли («гілки»), що містять (рекурсивно) два піддерева.

При читанні оголошень **data**, подібних цьому, слід пам'ятати, що **Tree** – це конструктор типу, тоді як **Branch** і **Leaf** – конструктори даних.

Крім встановлення зв'язку між цими конструкторами, це оголошення по суті є визначенням типів для **Branch** та **Leaf**:

```
Branch :: Tree a -> Tree a -> Tree a
Leaf :: a -> Tree a
```

Розглянемо рекурсивну функцію, пов'язану із типом **Tree**. Припустимо, необхідно визначити функцію **fringe**, яка повертає список усіх елементів у листі дерева зліва направо.

Зазвичай буває корисно спочатку записати тип нової функції: наразі видно, що її тип може бути **Tree a -> [a].**

Таким чином, **fringe** – поліморфна функція, що відображає дерево елементів типу **a** у список елементів типу **a**. Відповідне визначення має вигляд:

```
fringe :: Tree a -> [a]
fringe (Leaf x) = [x]
fringe (Branch left right) = fringe left ++ fringe right
```

Тут «++» є інфіксним оператором, що конкатенує два списки.

Розглянемо інший приклад.

```
data Nat = Zero | Succ Nat
```

У другій альтернативі бере участь сам тип **Nat**.

Як побачили у попередньому прикладі, рекурсивні визначення типів призводять до рекурсивних функцій.

```
(+) a Zero = a
(+) a (Succ b) = Succ (a + b)
(*) a Zero = Zero
(*) a (Succ b) = a + (a * b)
```

Списки як рекурсивний тип:

```
data [a] = [] | a : [a] deriving (Eq, Ord)
```

Функції списків також будуть рекурсивними. Розглянемо три основні функції роботи зі списками.

1. Функція **map**:

```
map :: (a -> b) -> [a] -> [b]
map f[] = []
map f (a:as) = f a : map f as
```

Ця функція застосовує функцію f для кожного елемента списку.

2. Функція **filter**:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p[] = []
filter p (x:xs) = if p x then x : filter p xs else filter p xs
```

Функція застосовує предикат p до кожного елемента списку та повертає новий список, що складається з елементів, для яких предикат true. Тому після введення

```
Prelude >filter isUpper "Hello World"
```

на екрані: "HW".

3. Функція згортки **foldr**:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (a:as) = f a (foldr f z as)
```

Введемо:

```
Prelude> let xs = [1,2,3,4,5]
Prelude> foldr (+) 0 xs
```

На екрані: 15.

Синоніми типів

З міркувань зручності в Haskell забезпечується можливість визначати синоніми типів, тобто імена для типів, що часто використовуються. Синоніми типів створюються за допомогою оголошення **type**. Приклад:

```
type String = [Char]
type Person = (Name,Address)
type Name = String
data Address = None | String
```

Синоніми типів не визначають нових типів, а просто задають нові імена існуючим. Наприклад, тип **Person -> Name** точно відповідає типу **(String, Address) -> String.**

Можна давати нові імена поліморфним типам:

```
type AssocList a b = [(a, b)]
```

Це тип асоціативного списку, який асоціює значення типу **a** зі значеннями типу **b**.

Вбудовані типи

Вбудовані типи не відрізняються від користувацьких, за винятком спеціального синтаксису.

1) Булевий тип

```
data  Bool  = False | True
   deriving (Read, Show, Eq, Ord, Enum, Bounded)
```

Булевий тип **Bool** є переліком.

Основні булеві функції – це «&&» (і), «||» (або) та **not** (заперечення). Ім'я **otherwise** (інакше) визначено як True.

2) Символи та рядки

Символьний тип **Char** є переліченням, значенням якого є символи Unicode. Тип **Char** є екземпляром класів **Read**, **Show**, **Eq**, **Ord**, **Enum** та **Bounded**.

Функції **toEnum** і **fromEnum**, які є стандартними функціями класу Enum, відповідно відображають символи в тип **Int** і навпаки.

Наступні літерали рівнозначні: \a і \BEL, \b і \BS, \f і \FF, \r і \CR, \t і \HT, \v і \VT \n і \LF.

Рядок – це список символів:

```
type String = [Char]
```

Рядки можна скоротити, використовуючи лексичний синтаксис.

Наприклад, "A string" є скороченням (абревіатурою) для ['A', ' ', 's', 't', 'r', 'i', 'n', 'g'].

3) Кортежі

Кортежі – це алгебраїчний тип даних зі спеціальним синтаксисом.

Тип кожного кортежу має один конструктор.

Всі кортежі є екземплярами класів **Eq**, **Ord**, **Bounded**, **Read**, та **Show** (за умови, що всі їх складові типи є екземплярами цих класів).

Конструктор для кортежу має вигляд: **(x, y)** або **(,) x y**. Обидва записи позначають той самий кортеж.

Аналогічно для конструкторів типів кортежів: той самий тип будуть позначати **(Int, Bool, Int)** та **(,,) Int Bool Int**.

4) Одиничний тип даних

```
data  () = () deriving (Eq, Ord, Bounded, Enum, Read,  Show)
```

Одиничний тип даних ( ) має єдиний, відмінний від _|_, член – конструктор без аргументів ( ).

5) Типи функцій

Функції – це абстрактний тип.

6) Типи IO та IOError

Тип **IO** вказує на операції, які взаємодіють із зовнішнім світом.

Тип **IO** є абстрактним: жодних конструкторів не видно для користувача. **IO** є екземпляром класів **Monad** та **Functor**.

Тип **IOError** є абстрактним типом, який представляє помилки, спричинені операціями введення-виведення. Він є екземпляром класів **Show** та **Eq**. Значення цього типу створюються різними функціями введення-виведення.

7) Числа

Haskell пропонує кілька видів чисел; на числові типи та операції над ними сильно вплинули Common Lisp та Scheme.

Integer – цілі числа довільної точності,

Int – цілі числа фіксованої точності,

(Integral a) => Ratio a – раціональні числа,

Float – дійсні числа з плаваючою крапкою одинарної точності,

Double – дійсні числа з плаваючою крапкою подвійної точності,

(RealFloat a) => Complex a – комплексні числа з плаваючою крапкою.

2.2 Функції

Арифметичні та теоретико-числові операції

Інфіксні методи класу (+), (*), (–) та унарна функція **negate** (яка також може бути записана як знак мінус, що стоїть перед аргументом), можуть бути застосовані до всіх чисел.

Методи quot, rem задовольняють наступним умовам, якщо **y** відмінний від нуля:

(x \`quot\` y) * y + (x \`rem\` y) = = x

(тут \`quot\` – ділення націло із округленням у бік нуля).

Метод класу **quotRem** приймає в якості аргументів ділене та дільник і повертає пару (частка, остача):

```
quotRem x y  =  (x `quot` y, x `rem` y)
```

Також для цілих чисел визначені предикати even (парний) і odd (непарний):

```
even x = x `rem` = 2 == 0
odd = not. even
```

Піднесення до степеня та логарифми

Показникова функція exp і логарифмічна функція log приймають як аргумент число з плаваючою крапкою і використовують при обчисленні основу *e*.

logBase a x повертає логарифм x за основою a.

sqrt повертає арифметичне значення квадратного кореня числа з плаваючою крапкою.

Є три операції піднесення до степеня, кожна з яких приймає по два аргументи:

- (^) підносить будь-яке число до невід'ємного цілого степеня,
- (^^) підносить дробове число до будь-якого цілого степеня,
- (**) приймає два аргументи з плаваючою крапкою.

Значення x^0 або x^^0 дорівнює 1 для будь-якого x, включаючи нуль; значення 0**y не визначено.

Абсолютна величина та знак

Функції abs і signum можуть бути застосовані до будь-якого числа і задовольняють умові:

abs x * signum x = = x.

Функції у Haskell

Оскільки Haskell є функціональною мовою, то функції відіграють у ньому головну роль.

Розглянемо визначення функції, яка додає два своїх аргументи:

```
add :: Int -> Int -> Int
add x y = x + y
e1 :: Int
e1 = add 1 2
```

На екрані 3.

Це приклад каррованої функції. (Термін «каррований» походить від прізвища Хаскелла Каррі, який популяризував цю ідею).

Щоб досягти ефекту некаррованої функції, необхідно використовувати кортеж:

```
add (x, y) = x + y
```

Однак у цьому випадку add насправді є функцією від одного аргументу.

Застосування першого варіанта функції add має вигляд **add e1 e2** та еквівалентно **((add e1) e2)**, оскільки застосування функції лівоасоціативне.

Іншими словами, застосування add до першого аргументу породжує нову функцію, яка потім застосовується до другого аргументу. Це називається каррінгом. Це відповідає типу функції add, **Int -> Int -> Int**.

Використовуючи add, можна визначити inc наступним способом:

```
inc :: Int -> Int
inc = add 1
```

Розглянемо випадок, що ілюструє користь передачі функції як аргумента:

```
add :: Int -> Int -> Int
add x y = x+y
map1 :: (a->b) -> [a] -> [b]
map1 f [] = []
map1 f (x: xs) = f x : map1 f xs
```

Виклик матиме вигляд

```
map1 (add 1) [1,2,3]
```

На екрані [2, 3, 4].

Символ долара

Складні вирази вигляду a + b зустрічаються досить часто, тому дужки можна замінити символом долара, щоб їх уникати: **f $ a + b** еквівалентно **f (a + b)** і транслюється у f(a + b). Зокрема, знак долара можна вкладати:

`f $ g x $ h y $ a + b` – еквівалентно

`f (g x (h y (a + b)))` і транслюється як f(g(x, h(y, a + b))).

Іноді можна побачити варіант з кутовими дужками: <$>. Можна вважати це тим самим, що і $. Також зустрічається <*> – вважається, що це кома, і **f <$> a <*> b** транслюється у f(a, b).

Зворотні апострофи

x `f` y транслюється у f(x, y). Те, що знаходиться між апострофами, – функція, зазвичай бінарна, а праворуч і ліворуч – аргументи.

Лямбда-абстракції

Замість використання рівнянь, функції можна визначати «анонімно», через лямбда-абстракцію.

Наприклад, функція, еквівалентна inc, може бути записана як **\x -> x + 1**. Аналогічно, функція add еквівалентна **\x -> \y -> x + y** (або **\x y -> x + y**). Тоді функцію map можна викликати:

```
map1 (x->x+1) [1,2,3]
```

Інфіксні оператори

Інфіксні оператори є звичайними функціями і можуть бути визначені у вигляді рівнянь. Розглянемо приклад визначення оператора конкатенації списків:

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs++ys)
```

Лексично інфіксні оператори мають повністю складатися зі знаків (тобто не букв, цифр чи прогалин), на противагу звичайним ідентифікаторам, які є буквено-цифровими. Haskell не має префіксних операторів, окрім мінуса (–), який є й інфіксним, і префіксним.

В якості іншого прикладу розглянемо інфіксний оператор, що оперує функціями та здійснює композицію функцій:

```
( . ) :: (b->c) -> (a->b) -> (a->c)
f. g = \ x -> f (g x)
```

Використання цього оператора матиме вигляд

```
(inc. inc) 2
```

Розтини

Оскільки інфіксні оператори насправді – просто функції, їм має сенс також надати можливість часткового застосування. У Haskell часткове застосування інфіксного оператора називають **розтином**. Розглянемо приклад:

```
(x+) = \y -> x+y
(+y) = \x -> x+y
(+) = \x y -> x+y
```

Використання таких операторів має вигляд

```
add1 :: Int -> Int
add1 = (+ 1)
e3 = add1 4
```

Тепер функції add та inc можна визначити так:

```
inc = (+1)
add = (+)
```

Можна змусити інфіксний оператор виступати як функціональне значення, для цього необхідно записати ідентифікатор в одиночні лапки. Наприклад, **x `add` y** – це те саме, що і **add x y**.

Оголошення пріоритетів операторів

Пріоритети операторів можуть бути задані для будь-якого інфіксного оператора чи конструктора. Ці оголошення вказують рівень пріоритету від 0 до 9 (пріоритет 9 – найвищий; звичайне застосування функції трактується як рівень пріоритету, що дорівнює 10).

Наприклад, оголошення пріоритетів операторів для «++» та «.» такі:

```
infixr 5 ++
infixr 9 .
```

Обидва вони оголошені як правоасоціативні, перший – із рівнем пріоритету 5, другий – 9. Лівоасоціативність задається за допомогою infixl, відсутність асоціативності – за допомогою infix.

У тому самому оголошенні може бути заданий пріоритет кількох операторів.

Якщо для даного оператора немає оголошення пріоритету, за замовчуванням приймається **infixl 9**.

Строгі та нестрогі функції

Функція називається **строгою**, якщо її застосування до обчислення, що незавершується, веде до того, що вона теж ніколи не завершиться.

У більшості мов програмування всі функції є строгими. Однак у Haskell це не так. Розглянемо приклад.

Розглянемо функцію

```
const1 :: a -> Int
const1 x = 1
```

Нехай також є функція bot:

```
bot = bot
```

Тоді у Haskell виклик

```
const1 bot
```

поверне 1.

Оскільки помилки та обчислення, що не завершуються, є семантично еквівалентними в Haskell, наведене справджується також і для помилок: наприклад, **const1 (1 / 0)** також повертає 1.

Головною перевагою нестрогих функцій є те, що вони звільняють програміста від багатьох турбот про порядок обчислень. Значення, що вимагають істотних обчислювальних витрат, можуть передаватися як аргументи функції без остраху того, що вони можуть бути обчислені без необхідності.

Важливим прикладом цього є можливість роботи з нескінченними структурами даних.

2.3 Нескінченні структури даних, ліниві зразки та вкладені форми

«Нескінченні» структури даних

Структури даних у Haskell також є нестрогими. Іншими словами, нестрогими є конструктори даних.

Нестрогі конструктори допускають створення нескінченних (концептуально) структур даних. Наприклад, нескінченний список одиниць:

```
ones = 1 : ones
```

Інший приклад, функція numsFrom:

```
numsFrom n = n : numsFrom (n+1)
```

Тобто, **numsFrom n** – це нескінченний список послідовних цілих чисел, починаючи з n. З її допомогою ми можемо отримати нескінченний список квадратів:

```
squares = map (^2) (numsFrom 0)
```

Вочевидь передбачається виділити деяку кінцеву частину списку для фактичних обчислень.

Розглянемо приклад: функція take, яка виділяє перші n елементів списку:

```
take1 :: [a]->Int->[a]
take1 (s: sr) k = if k> 0 then s: take1 sr (k-1) else []
```

Тоді виклик

```
take1 (numsFrom 0) 5
```

повертає **[0, 1, 2, 3, 4]**.

Інший приклад використання циклічності – послідовність Фібоначчі, яка може ефективно обчислюватися як наступна нескінченна послідовність:

```
fib = 1:1: [a+b | (a,b) <- zip fib (tail fib)]
```

де zip – це функція зі Standard Prelude, яка повертає список пар елементів із двох своїх аргументів-списків:

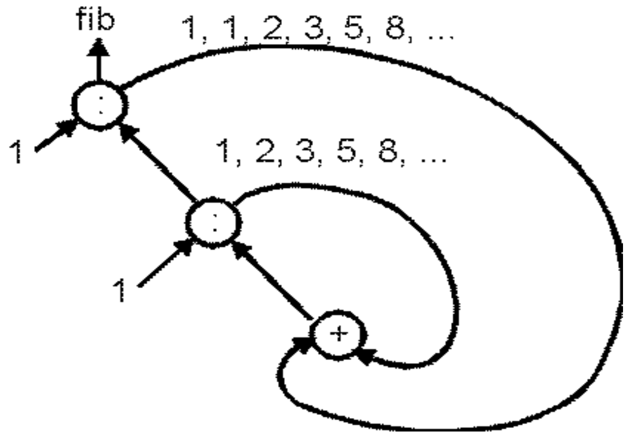
```
zip (x: xs) (y: ys) = (x, y) : zip xs ys
zip xs ys = []
```

Варто зазначити, що нескінченний список fib визначений у термінах самого себе. Тоді виклик

```
take 5 fib
```

дасть **[1, 1, 2, 3, 5]**.

Ці обчислення можна проілюструвати на малюнку:



Функція error

У Haskell є вбудована функція error, тип якої **String -> a**. Ця функція повертає значення поліморфного типу (про який їй, на перший погляд, нічого не відомо).

Однак, існує одна величина, спільна для всіх типів: _|_. Семантично це точно те значення, яке завжди повертається функцією error (всі помилки мають значення «_|_»).

Ця функція корисна, коли необхідно перервати виконання програми, якщо щось пішло не так. Наприклад, визначення head із Standard Prelude має вигляд:

```
head (x: xs) = x
head [] = error "head{PreludeList}: head []"
```

Умовні вирази

Для визначення функції у мові Haskell можна використовувати умовні вирази. Наприклад, функція signum, що обчислює знак переданого їй аргументу, має вигляд:

```
signum :: Int -> Int
signum x = if x > 0 then 1
```

```
                else if x < 0 then -1 else 0
```

Умовний вираз записується у вигляді:

```
if умова then вираз else вираз.
```

Умовою у визначенні умовного оператора є будь-який вираз типу Bool.

При порівнянні можна використовувати наступні оператори:

<, >, <=, >= – оператори мають звичний зміст (наприклад, як в мові С),

= = – оператор перевірки на рівність,

/ = – оператор перевірки на нерівність.

Вирази типу **Bool** можна комбінувати за допомогою загальноприйнятих логічних операторів && та || (І та АБО), і функції заперечення **not**. Приклади допустимих умов:

```
x >= 0 && x <= 10
x > 3 && x /= 10
(x > 10 || x < -10) && not (x == y)
```

Зрозуміло, можна визначати свої функції, що повертають значення типу Bool, і використовувати їх як умови. Наприклад, можна визначити функцію isPositive, що повертає True, якщо її аргумент невід'ємний, і False в іншому випадку:

```
isPositive :: Int -> Bool
isPositive x = if x > 0 then True else False
```

Варто зазначити, що функцію isPositive можна визначити і простіше:

```
isPositive x = x > 0
```

Зіставлення зі зразком

As-зразки (as-patterns). Іноді зручно присвоїти зразку ім'я для використання у правій частині рівності. Наприклад, функція, що дублює перший елемент списку, може бути записана так:

```
f (x: xs) = x: x: xs
```

Зауважимо, що **x: xs** з'являється і як зразок у лівій частині рівності, і як вираз – у правій. Для поліпшення читабельності можна віддати перевагу запису **x: xs** тільки один раз; цього можна досягти, використовуючи as-зразок, як тут:

```
f s @ (x: xs) = x: s
```

З технічної точки зору, as-зразок завжди призводить до успішного зіставлення, хоча підзразки (в нашому випадку **x: xs**) можуть, звичайно, привести до невдачі.

Символи підстановки (wildcards). Часто зустрічається інша ситуація, коли зіставлення відбувається зі значенням, яке насправді не цікавить.

```
head (x:_) = x
tail (_: xs) = xs
```

Кожен символ підстановки (незалежно від інших) зіставляється з будь-чим, але на відміну від формальних параметрів, зв'язування не відбувається; з цієї причини допускається наявність більш ніж одного «_» у даному рівнянні.

Семантика зіставлення зі зразком

Зіставлення зі зразком може завершитися невдачею (fail), успіхом (succeed), або бути відхиленим (diverge).

Успішне зіставлення із зразком зв'язує формальні параметри зразка.

Відхилення відбувається, коли значення, необхідне для зразка, містить помилку (_|_).

Сам процес зіставлення відбувається «згори донизу, зліва направо». Невдача будь-де у цій функції призводить до невдачі всієї функції, після чого спроба зіставлення робиться для наступної функції.

Якщо всі функції приведуть до невдачі, значення застосування функції – _|_, що призводить до помилки виконання.

Шаблони верхнього рівня можуть також мати булеві запобіжники як функції, що дозволяють з'ясувати знак числа:

```
sign x | x > 0 = 1
       | x == 0 = 0
       | x < 0 = -1
```

Варто зазначити, що для того самого зразка може бути передбачена ціла послідовність запобіжників; як і зразки, вони обчислюються зверху вниз, і перший з них, що повертає True, приведе до успішного зіставлення.

Операція вибору

У Haskell, як і в C++, крім оператора розгалуження є також оператор вибору.

Припустимо, необхідно визначити деяку функцію, яка повертає 1, якщо їй передано аргумент 0; повертає 5, якщо аргумент дорівнював 1; повертає 2, якщо аргумент дорівнює 2 і повертає –1 в решті випадків. У цьому випадку зручним виявляється оператор вибору:

```
f x = case x of
   0 -> 1
   1 -> 5
   2 -> 2
   _ -> -1
```

Синтаксис оператора case очевидний; символ «_» аналогічний конструкції default у мові C++.

Наведену вище функцію можна записати і таким чином:

```
f x = case x of
 {0 -> 1; 1 -> 5; 2 -> 2; _ -> -1 }
```

Такий спосіб явно задає групування та розділення конструкцій мови. Однак можна обійтися без нього.

Загальне правило таке. Після ключових слів where, let, do і of інтерпретатор вставляє відкриваючу дужку { та запам'ятовує колонку, в якій записана наступна команда.

Надалі перед кожним новим рядком, вирівняним на запам'ятовану величину, вставляється символ-роздільник ';'.

Якщо наступний рядок вирівняно менше (тобто його перший символ знаходиться ліворуч від запам'ятованої позиції), вставляється закриваюча дужка.

Ліниві зразки

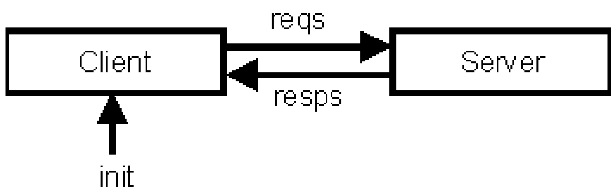
Лінивий зразок має вигляд **~pat**. Ліниві зразки неспростовні, зіставлення значення «v» із зразком ~pat завжди успішне, незалежно від pat.

Іншими словами, якщо ідентифікатор з pat надалі буде використаний у правій частині, він буде пов'язаний або з відповідною частиною отриманого значення, якщо зіставлення «v» з pat завершиться успішно, або з _|_ в іншому випадку.

Ліниві зразки корисні в тих випадках, коли нескінченні структури даних визначаються рекурсивно.

Розглянемо приклади.

Приклад 1. Клієнт client надсилає послідовність запитів (requests) до сервера server, а сервер відповідає на кожен запит відповіддю (response) певного типу.



Припустимо, сервер та клієнт мають наступну структуру:

```
client init (resp:resps)=init:client (next resp) resps
server (req:reqs) = process req : server reqs
```

Тут передбачається, що функція next визначає наступний запит на задану відповідь сервера, а функція process обробляє запит клієнта і повертає потрібну відповідь сервера.

Однак, ця програма нічого не робитиме.

Проблема полягає в тому, що client, у тому вигляді, в якому він використовується в рекурсивних викликах reqs і resps, намагається виконати зіставлення зі зразком списку відповідей, перш ніж він отримає першу відповідь від сервера.

Найкращим виходом є використання лінивого зразка:

```
client init ~(resp:resps) =init:client (next resp) resps
```

Оскільки ліниві зразки неспростовні, зіставлення відразу виявляється успішним, дозволяючи «передати» початковий запит, що, в свою чергу, дозволяє згенерувати першу відповідь.

Приклад 2:

```
lazy :: [Int] -> Bool -> [Int]
lazy ~(x:xs) isNull = if isNull then [] else [x]
e4 = lazy [1,2] False
e5 = lazy [] True
e6 = lazy [] False
e4 -> [1]
e5 -> []
e6 -> Exception.
```

Вкладені форми

Часом необхідно створити вкладену область видимості усередині виразу, щоб породити локальні зв'язування, не видимі ззовні, – своєрідні «блочно-структурні» форми. Це можна зробити двома способами.

1. **Let-вирази**. В Haskell let-вирази зручно використовувати щоразу, коли потрібен вкладений набір зв'язувань.

Розглянемо приклад:

```
e9 = let y = 2 :: Float
         f x = (x+y)/y
     in f 1 + f 2
```

Набір зв'язувань, що породжуються let-виразами, є взаємно рекурсивним, а зразки, що зв'язуються, трактуються як ліниві (тобто такі, що несуть неявну ~).

Єдиними допустимими видами оголошень є сигнатури типів, зв'язування функцій і зв'язування зразків.

2. **Конструкція where**. Іноді зручно поширити зв'язування на декілька функцій із запобіжниками, для цього використовується конструкція where.

Розглянемо приклад:

```
f :: Int -> Int -> String
f x y | y > z = "y > x^2"
      | y == z = "y = x^2"
      | y < z = "y < x^2"
  where
     z = x * x
```

Тоді виклик

```
f 3 5
```

дасть **y < x^2**

Такого ефекту не можна досягти використанням let-виразів, чия область видимості обмежена тими виразами, до яких вони входять.

Конструкція where має такі самі властивості й обмеження на зв'язування, як і вираз let.

Зауважимо, що при створенні конструкцій let та where можна використовувати явну нотацію. Наприклад:

```
e9 = let { y = 2 :: Float
         ; f x = (x+y)/y}
      in f 1 + f 2
```

Явна нотація корисна у випадку, коли бажано розмістити в одному рядку більше одного оголошення. Розглянемо приклад:

```
let y = a * b; z = a/b
    f x = (x+y)/z
in f c + f d
```

Стрілка вліво

Працює як оператор присвоєння:

```
a <- createEntry x
```

Це еквівалентно

```
a = createEntry(x)
```

2.4 Класи типів

Класи типів та перевантаження

В мові Haskell існує спеціальний поліморфізм. Прикладами такого поліморфізму можуть бути:

- числові оператори (наприклад, +) часто визначають як здатні працювати з різними видами чисел;
- оператор рівності (= =) зазвичай працює з числами та багатьма іншими (але не всіма) типами.

Варто зазначити, що така перевантажена поведінка буде специфічною для кожного типу (фактично поведінка іноді не визначена або помилкова), у той час як для параметричного поліморфізму тип насправді не важливий (приміром, для функції length не важлива природа елементів списку).

У Haskell механізмом, що забезпечує керування спеціальним поліморфізмом (перевантаженням) є класи типів. Розглянемо приклад – оператор рівності.

Існує багато типів, для яких необхідно визначити рівність, проте є типи для яких цього робити не варто. Наприклад, порівняння на рівність функцій, взагалі кажучи, є обчислювально нерозв'язною задачею, тоді як задача порівняння двох списків виникає часто. (Зауважимо, що поняття рівності можна трактувати по-різному, тут йдеться саме про «рівність значень» на противагу «рівності вказівників», що має місце, наприклад, для оператора == Java. Рівність вказівників не є прозорою за посиланнями і тому погано підходить для функціональних мов).

Розглянемо наступне визначення функції **elem**, яка перевіряє входження до списку:

```
x `elem` [] = False
x `elem` (y: ys) = x == y | (x `elem` ys)
```

На перший погляд, тип elem повинен бути **a -> [a] -> Bool**. Однак із цього випливає, що = = має тип **a -> a -> Bool**, хоч як вже згадувалося, не варто визначати = = для всіх типів.

Більше того, якщо оператор = = був визначений для всіх типів, порівняння на рівність двох списків досить сильно відрізняється від порівняння двох чисел.

Класи типів вирішують перелічені проблеми. Вони дозволяють оголосити, які типи є втіленнями (instances) даного класу та дати визначення перевантажених операцій, пов'язаних із цим класом.

Розглянемо приклад – клас типу, що містить оператор рівності:

```
class Eq a where
    (==) :: a -> a -> Bool
```

Тут **Eq** є собою іменем класу, що визначається, з єдиною операцією = =. Це оголошення може читатися так: «тип **a** є втіленням класу **Eq**, якщо для нього визначена (перевантажена) операція == підходящого типу».

Обмеження, що тип **a** може бути втіленням класу **Eq**, записується так: **Eq a**. Таким чином, **Eq a** не є виразом типу, але описує обмеження на тип і називається **контекстом**.

Тепер можна визначити функцію **elem**, використовуючи клас **Eq**. Маємо:

```
elem :: (Eq a) => a -> [a] -> Bool
x `elem` [] = False
x `elem` (y: ys) = x `eq` y | x `elem` ys
```

Тепер необхідно вказати, які типи є втіленням класу **Eq**, і задати специфічну поведінку = = для кожного з цих класів. Це роблять за допомогою оголошень втілень.

Розглянемо приклад:

```
instance Eq Integer where
 x == y = x `integerEq` y
```

Таке визначення == називається **методом**. Функція **integerEq** є «примітивною» функцією, що порівнює цілі значення на рівність, але в загальному випадку в правій частині рівності допускається довільний правильний вираз, так само як і у визначенні будь-якої іншої функції.

Розглянуте оголошення по суті означає: «тип **Integer** є втіленням класу **Eq**, і визначення методу, що відповідає операції ==».

Аналогічно,

```
instance Eq Float where
  x == y = x `floatEq` y
```

дозволяє порівнювати числа з плаваючою крапкою, використовуючи ==.

Розглянемо приклад визначення функції, яка перевіряє наявність елемента в списку:

```
class Eq1 a where
  eq2 :: a -> a -> Bool

elem1 :: (Eq1 a) => a -> [a] -> Bool
x `elem1` [] = False
x `elem1` (y: ys) = x `eq2` y || x `elem1` ys

instance Eq1 Int where
   x `eq2` y = abs (x-y) < 3

list1 :: [Int]
list1 = [1,5,9,23]

e1 = 2 `elem1` list1
```

На екрані отримаємо True.

Аналогічно можна чинити і з рекурсивними типами:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
                                           deriving Show
```

```
instance (Eq a) => Eq (Tree a) where
 (Leaf a) `eq` (Leaf b) = a `eq` b
 (Branch l1 r1) `eq` (Branch l2 r2) = (l1 `eq` l2)
                                       && (r1 `eq` r2)
  _ `eq` _ = False
```

Запис **instance (Eq a) => Eq (Tree a) where** означає, що **Eq (Tree a)** – нащадок **(Eq a)**

Це означає, що можна перевіряти на рівність дерева, що зберігають об'єкти типу **a**, тоді і тільки тоді, коли відомо, як перевіряти на рівність об'єкти типу **a**.

Отже, маємо:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Show

class Eq1 a where
   eq2 :: a -> a -> Bool

instance Eq1 Int where
   x `eq2` y = abs (x - y) < 3

instance (Eq1 a) => Eq1 (Tree a) where
   (Leaf a) `eq2` (Leaf b) = a `eq2` b
   (Branch l1 r1) `eq2` (Branch l2 r2) = (l1 `eq2` l2)
                                          && (r1 `eq2` r2)
    _ `eq2`_ = False

tree1,tree2 :: Tree Int
tree1 = Branch (Leaf 1) (Leaf 2)
tree2 = Branch (Leaf 2) (Leaf 1)
e4 = tree1 `eq2` tree2
```

На екрані отримаємо True.

У бібліотеці Prelude визначено клас Eq, який має вигляд

```
 class Eq a where
(==), (/=) :: a -> a -> Bool
```

```
 x /= y = not (x == y)
```

Це приклад класу із двома операціями: одна для рівності, інша – для нерівності. Він також демонструє використання **методу за замовчуванням**, в нашому випадку для операції нерівності /=. Якщо в оголошенні втілення опустити метод для певної операції, тоді замість нього буде використовуватися метод за замовчуванням, визначений в оголошенні класу, якщо він існує.

Як вже було показано вище, у Haskell також підтримується поняття розширення класів (деякою мірою аналогічно спадкуванню в імперативних мовах).

Наприклад, можна визначити клас **Ord**, який успадковує всі операції з **Eq**, і додатково має набір операцій порівняння та функції визначення мінімуму та максимуму:

```
class (Eq a) => Ord a where
   (<), (<=), (>=), (>) :: a -> a -> Bool
   max, min :: a -> a -> a
```

В такому випадку кажуть, що **Eq** є **суперкласом** відносно **Ord** (і навпаки, **Ord** є **підкласом Eq**), і будь-який тип, що є втіленням **Ord**, має бути також втіленням **Eq**. Розглянемо приклад:

```
class Eq1 a where
   eq2 :: a -> a -> Bool

class (Eq1 a) => Ord a where
  lt,le :: a -> a -> Bool
  x `le` y = x `eq2` y || x `lt` y

instance Eq1 Int where
   x `eq2` y = abs (x - y) < 3

instance Ord Int where
```

```
  x `lt` y = x < y + 1

i :: Int
i = 3
e5 :: Bool
e5 = i `lt` i
```

На екрані отримаємо True.

Мова Haskell також допускає **множинне спадкування**, оскільки класи можуть мати більше одного суперкласу. Наприклад:

```
class (Eq a, Show a) => C a where ...
```

створює клас **C,** який успадковує операції і з **Eq**, і з **Show**.

У Haskell методи класів відносяться до оголошень верхнього рівня. Вони поділяють той самий простір імен, що і звичайні змінні; те саме ім'я не може одночасно використовуватися для позначення і методу класу, і змінної чи методу іншого класу.

Методи класу можуть містити додаткові обмеження класу для будь-якої змінної типу, за винятком тієї, що визначає поточний клас. Наприклад:

```
class C a where
  m :: Show b => a -> b
```

Тут метод **m** вимагає, щоб тип **b** був із класу **Show**. Однак метод **m** не може накладати додаткових обмежень класу на тип **a**. Такі обмеження натомість мають бути частиною контексту в оголошенні класу.

У оголошенні класу можна використовувати типи вищого порядку, тобто типи функцій. Розглянемо приклад:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Зауважимо, що тип змінної **f** застосовується до інших типів в **f a** та **f b**. Таким чином, можна очікувати, що він буде зв'язуватися з деяким типом, так само як **Tree**, який може бути застосований до аргументу.

Реалізація класу **Functor** для типу **Tree** може мати вигляд:

```
instance Functor Tree where
  fmap f (Leaf x) = Leaf (f x)
  fmap f (Branch t1 t2) = Branch (fmap f t1) (fmap f t2)
```

Таким чином, маємо:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Show
```

Конструкція **deriving** неявно робить оголошення втілення **Show**.

```
class Functor1 f where
  fmap1 :: (a -> b) -> f a -> f b

instance Functor1 Tree where
 fmap1 f (Leaf x) = Leaf (f x)
 fmap1 f (Branch t1 t2) = Branch (fmap1 f t1) (fmap1 f t2)

tree1 = Branch (Leaf 1) (Leaf 2)
e6 = fmap1 (+1) tree1
```

На екрані отримаємо

```
Branch (Leaf 2) (Leaf 3)
```

Порівняння з ООП

Розглянемо аналогію класів типів у Haskell з об’єктно-орієнтованим підходом. Цю відповідність можна зобразити наступним чином:

Haskell	**ООП**
Клас типу	Клас
Тип	Об'єкт

Слід зазначити, що на відміну ООП, типи – це об'єкти, і, зокрема, відсутнє поняття про внутрішній стан об'єкта чи типу, що міг би змінюватися.

Перевага мови Haskell над деякими іншими ОО-мовами полягає в тому, що її методи цілком «типобезпечні»: будь-яка спроба застосувати метод до значення, чий тип не належить до необхідного класу, буде помічена при компіляції, а не під час виконання.

Розглянемо порівняння Haskell з іншими мовами програмування (на прикладі C++ та Java).

1. Haskell відокремлює визначення типу від визначення методів, пов'язаних із цим типом. Клас C++ або Java зазвичай визначає і структуру даних (змінні – члени класу), і функції, асоційовані з цією структурою (методи). У Haskell ці визначення даються окремо.

2. Метод класу, визначений у класі Haskell, відповідає віртуальній функції у класі мови C++. Кожне втілення класу забезпечує своє власне визначення для кожного методу, замовчування у класі відповідають реалізації за замовчуванням віртуальної функції у базовому класі C++.

3. Класи Haskell у грубому наближенні схожі на інтерфейси Java. Як і оголошення інтерфейсів, оголошення класів Haskell задають протокол використання об'єктів, а не визначення власне об'єкта.

4. Haskell не підтримує перевантаження в стилі C++, коли функції з різними типами поділяють загальне ім'я.

5. Для типів об'єктів у Haskell немає неявного приведення; відсутній універсальний базовий клас, на зразок Object, до якого можна приводити значення.

6. C++ та Java доєднують ідентифікуючу інформацію (таку, як таблиця віртуальних функцій) до представлення об'єкта під час виконання. У Haskell подібна інформація приєднується до значень логічно, а не фізично, через систему типів.

7. У систему класів Haskell не вбудовано контроль доступу (такої, як мітки public або private у визначенні класу). Натомість для приховування або відкриття компонентів слід користуватися системою модулів.

2.5 Задання типів через newtype та data

Оголошення newtype

Ключове слово **newtype** вводить новий **тип-обгортку**. У нього може бути лише один конструктор з єдиним аргументом:

```
Newtype Sum a = Sum a
```

Це те саме, що і

```
data Sum a = Sum a
```

Єдина відмінність полягає в тому, що у разі нового типу обчислювач не бачить різниці між **Sum a** і **a**. Її бачить лише компілятор. Це означає, що на розгортання і загортання такого значення в тип-обгортку не витрачається жодних зусиль.

Такі типи підходять для вирішення двох завдань:

1. Більш точна перевірка типів. Наприклад, існують типи, які описують фізичні величини: вони є числами, але вони мають також і розмірності. Тоді можна написати:

```
type Velocity = Double
type Time = Double
type Length = Double
velocity :: Length -> Time -> Velocity
velocity leng time = leng / time
```

За таких умов, можна підставити на місце часу шлях і навпаки. Але за допомогою типів-обгорток ми можемо виключити такі випадки:

```
Newtype Velocity = Velocity Double
Newtype Time = Time Double
Newtype Length = Length Double
velocity :: Length -> Time -> Velocity
velocity (Length leng) (Time time) = Velocity $ leng / time
```

2. Визначення кількох екземплярів одного класу для одного типу. Наприклад, потрібно зробити два екземпляри для того самого типу **Num a** => **a**:

```
Newtype Sum a = Sum a
Newtype Prod a = Prod a
```

Тоді можемо визначити два екземпляри для двох різних типів. Один для Sum:

```
instance Num a => Monoid (Sum a) where
  mempty = Sum 0
  mappend (Sum a) (Sum b) = Sum (a + b)
```

А інший для Prod:

```
instance Num a => Monoid (Prod a) where
    mempty = Prod 1
    mappend (Prod a) (Prod b) = Prod (a * b)
```

Розглянемо приклад.

```
newtype Natural = MakeNatural Integer deriving (Show, Eq)

toNatural :: Integer -> Natural
toNatural x | x < 0 = error "Can't create
                                          negative naturals!"
            | otherwise = MakeNatural x

fromNatural :: Natural -> Integer
fromNatural (MakeNatural i) = i

instance Num Natural where
     fromInteger = toNatural
     x + y = toNatural (fromNatural x + fromNatural y)
     x - y = let r = fromNatural x - fromNatural y in
               if r < 0 then error "Unnatural subtraction"
                         else toNatural r
     x * y = toNatural (fromNatural x * fromNatural y)
```

Без цього оголошення тип **Natural** не належатиме до класу **Num**. Втілення класу, оголошені для старого типу, не переносяться на новий.

Повна мета типу **Natural** полягає у створенні окремого втілення класу **Num**. Це було б неможливим, якщо **Natural** був визначений як синонім типу для **Integer**.

```
e1 :: Natural
e1 = toNatural 1
```

На екрані **MakeNatural 1**

```
e4 :: Natural
e4 = (3 - 4) + 3
```

На екрані **Exception Unnatural subtraction**

Поведінка не зміниться, якщо замість оголошення newtype використовувати оголошення data.

Мітки полів (записи)

До полів усередині типу даних Haskell можна звертатися або за їхньою позицією, або по імені, за умови використання **міток полів**.

Розглянемо тип даних, що описує точку на площині:

```
data Point = Pt Float Float
```

Дві координати точки типу **Point** – це перший та другий аргументи конструктора даних **Pt**. Функції, подібні до наступної:

```
pointx :: Point -> Float
pointx (Pt x _) = x
```

можуть використовуватися як посилання на першу координату точки в більш описовому ключі, однак для великих структур створювати такі функції вручну накладно.

Конструктори в оголошенні **data** можуть бути оголошені з асоційованими іменами полів, вказаними у фігурних дужках. Ці імена ідентифікують компоненти конструктора за іменем, а не за позицією.

Тоді тип Point можна визначити так:

```
data Point = Pt { pointx, pointy :: Float }
```

Це визначення визначає два імені поля: pointx і pointy.

Такі імена полів можуть бути використані як функції-селектори, що вибирають компоненти структури. В нашому прикладі ці селектори мають вигляд:

```
pointx :: Point -> Float
pointy :: Point -> Float
```

Розглянемо приклад:

```
data Point = Pt {pointx, pointy :: Float} deriving Show
absPoint :: Point -> Float
absPoint p = sqrt (pointx p * pointx p + pointy p * pointy p)
```

```
e5 :: Point
e5 = Pt {pointx = 1, pointy = 2}
e6 :: Float
e6 = absPoint e5
```

На екрані **2.236**

Мітки полів поділяють простір імен верхнього рівня зі звичайними змінними та методами класів. Ім'я поля не може використовуватися для більш ніж одного типу даних у заданій області видимості.

Однак для певного типу даних те саме ім'я поля може використовуватися більш ніж в одному конструкторі, доки це поле має той самий тип для всіх випадків такого використання. Розглянемо приклад:

```
data T = C1 {f::Int, g::Float}
         | C2 {f::Int, h::Bool} deriving Show

e9 :: T
e9 = C1 {f = 1, g = 2}
e10 :: T
e10 = C2 {f = 3, h = False}
e11 :: Int
e11 = f e9
```

На екрані **1**

```
e12 :: Int
e12 = f e10
```

На екрані **3**.

Строгі конструктори даних

Структури даних в Haskell, взагалі кажучи, є лінивими: компоненти не обчислюються, поки в них не виникає необхідність. На внутрішньому рівні кожне поле лінивого об'єкта даних обгорнуте в структуру, на яку зазвичай посилаються як на «перехідник», який інкапсулює обчислення, що визначають значення поля. Вхід в цей «перехідник» не здійснюється, поки значення не буде потрібне; «перехідник», який містить помилку (_|_), не впливає на інші елементи структури даних. Наприклад, в Haskell кортеж **('a', _|_)** являє собою допустиме значення. Елемент **'a'** може використовуватися без впливу на інші компоненти кортежу.

Більшість імперативних мов програмування є строгими, а не лінивими: всі компоненти структури даних перетворюються до їх значень, перш ніж вони будуть поміщені в структуру. З «перехідниками» пов'язаний ряд накладних витрат: на їх конструювання та обчислення витрачається час, вони займають місце в динамічній пам'яті, і також змушують збирач сміття зберігати інші структури, які потребують обчислення «перехідників». Щоб уникнути накладних витрат, в оголошенні data використовуються **прапорці строгості**. Вони дозволяють обчислювати поля конструктора негайно, вибірково подавлюючи лінивість. Поле, позначене «!» в оголошенні data, обчислюється при створенні структури, замість відкладання обчислення в «перехідник». Розглянемо приклад:

```
data RealFloat a => Complex a =!a :+ !a
```

Тут + є інфіксним конструктором. Це визначення позначає дві компоненти, дійсну і уявну частину комплексного числа, як строгі. Це більш компактне представлення комплексних чисел, але воно призводить до того, що комплексні числа з невизначеним компонентом, наприклад 1 :+ _|_, стають повністю невизначеними (_|_). Прапорець строгості ! може бути присутнім тільки в оголошенні data. Він не може використовуватися ні в інших сигнатурах типу, ні в жодних інших визначеннях типу. Ефекту строгості для аргументів функції можна досягти застосуванням функції seq або $!.

Функція seq має вигляд:

```
seq :: a -> b -> b
```

і визначається формулами

```
seq _|_b = _|_
seq a b = b, if a /= _|_
```

Оператор $! є строгим (викликається за значенням) застосуванням, він визначений в термінах seq. Prelude також містить визначення оператора $ для виконання нестрогих застосувань. Оператори $! і $ визначаються наступним чином:

```
infixr 0 $, $!
    ($), ($!) :: (a -> b) -> a -> b
    f $ x = f x
    f $! x = x `seq` f x
```

Наявність оператора нестрогого застосування $ може здаватися надмірним, оскільки звичайне застосування (f x) означає те ж саме, що (f $ x). Проте $ має низький пріоритет і праву асоціативність, тому іноді круглі дужки можна опустити, наприклад:

```
f $ g $ h x = f (g (h x))
```

Розглянемо приклад застосування прапорця строгості:

```
data HList a = Cons !a (HList a) | Nil deriving Show

hd (Cons x y) = x
tl (Cons x y) = y

e10 :: HList Bool
e10 = Cons True (Cons (error "e10") Nil)
e11, e12 :: Bool
e11 = hd e10
e12 = hd (tl e10)
e13 :: HList Bool
e13 = tl (tl (e10))
```

Результат має вигляд:

e10 => Cons True *** *Exception e10*

e11 => True

e12 => Exception e10

e13 => Exception e10 *** *якщо прибрати прапорець строгості, то отримаємо Nil*

2.6 Робота з введенням-виведенням та файлами

Введення-виведення

Система введення-виведення в Haskell є чисто функціональною. Кожна дія введення-виведення повертає значення. В системі типів значення, що повертається, «помічено» типом IO, щоб відрізнятися від інших значень. Наприклад, тип функції **getChar** такий:

```
getChar :: IO Char
```

Тип IO Char вказує на те, що getChar, будучи викликаний, виконує деякі дії, які повертають символ. Розглянемо приклад:

```
e1 = getChar
```

Після введення символу, наприклад d, на екрані отримаємо 'd'

Дії, які нічого не повертають, використовують тип unit – (). Наприклад, функція **putChar**:

```
putChar :: Char -> IO ()
```

Вона приймає символ як аргумент, але не повертає нічого змістовного. Тип unit схожий на тип void в інших мовах.

Дії збираються в послідовність за допомогою оператора: >>= (читається «зв'язати»). Цей оператор визначений в класі Monad. Приклад використання:

```
e1 = putStr "x: " >> getLine >> = \l -> return (words l)
```

Тут функція getLine пов'язана з лямбда функцією

```
\l -> return (words l)
```

Після введення

```
*Main>e1
```

на екрані отримаємо

x: asdsafdsaf – рядок, що вводиться

тоді на екрані маємо

```
["asdsafdsaf"]
```

Оператори зв'язування можна також трактувати як спосіб позбутися від проміжних змінних:

```
doSomething >>= doSomethingElse >>= finishItUp
```

Це еквівалентно запису

```
x = doSomething()
y = doSomethingElse(x)
finishItUp(y)
```

Іноді пишуть

```
z <- finishItUp =<< doSomethingElse =<< doSomething
```

Це буде еквівалентно

```
x = doSomething()
y = doSomethingElse(x)
z = finishItUp(y)
```

Оператор >> може розглядатися наступним чином:

```
doSomething >> doSomethingElse
```

що еквівалентно наступному імперативного коду:

```
doSomething()
doSomethingElse()
```

Замість прямого використання оператора >>= зазвичай використовується **do**-нотація, яка дозволяє приховати оператори, які монтують послідовність, за синтаксисом, схожим на загальноприйняті мови. Наприклад, буде коректним наступний код:

```
e1=do putStr "x:"
      l <- getLine
      return (words l)
```

Для do-виразів виконуються тотожності:

```
do {e} = e
do {e; stmts} = e >> do {stmts}
do {p <- e; stmts} = let ok p = do {stmts}
      ok _ = fail "..."
```

```
        in e >> = ok
    do {let decls; stmts} = let decls in do {stmts}
```

де stmts – деякі вирази. Пропуски "..." позначають повідомлення про помилку, що генерується компілятором, яке передається функції fail. Таким чином, ключове слово do вводить послідовність інструкцій, які виконуються в певному порядку.

Інструкція – це або дія (зразок зв'язується з результатом дії за допомогою <-), або множина локальних визначень, що вводяться з використанням **let**. Розглянемо програму, яка читає і друкує символ:

```
main :: IO ()
main = do c <- getChar
    putChar c
```

Інший приклад, виведення на екран рядка, що вводиться з клавіатури:

```
main = do l <- getLine
    putStr l
```

Використання імені main є важливим: main визначається як точка входу в програму на Haskell (аналогічно функції main в C) і повинна мати тип IO, зазвичай IO () (винятком є модуль Main).

Перша програма виконує дві дії послідовно: по-перше, вона читає символ, пов'язуючи результат зі змінною «**c**», по-друге, друкує символ. На відміну від виразу **let**, при використанні якого областю видимості змінної стає все визначення, змінні, визначені через **<-**, доступні тільки в наступних інструкціях. Розглянемо функцію ready, яка читає символ і повертає True, якщо символ – це «y»:

```
ready :: IO Bool
ready = do c <- getChar
    c == 'y'
```

Наведена програма не працює, тому що друга інструкція в do – це просто логічне значення, а не дія. Необхідно взяти це булеве значення і створити дію, яка його повертає в якості свого результату. Функція return робить саме це:

```
return :: a -> IO a
```

Тоді, в попередньому прикладі, останній рядок повинен мати наступний вигляд:

```
turn (c == 'y')
```

Розглянемо функцію **getLine**:

```
getLine :: IO String
getLine = do c <- getChar
    if c == '\n'
        then return ""
        else do l <- getLine
            return (c:l)
```

Кожне do вводить окремий ланцюжок інструкцій. Будь-яка конструкція, що «втручається», на зразок if, повинна використовувати нове do для ініціювання подальшої послідовності дій. Варто зауважити, що викликати деяку дію введення-виведення в звичайній функції не можна. Наприклад, функція:

```
f :: Int -> Int -> Int,
```

не здатна виконувати жодне введення-виведення, оскільки IO відсутнє в значенні, що вона повертає.

Програмування з діями

Дії введення-виведення є звичайними значеннями Haskell: їх можна передавати у функції, поміщати в структури і використовувати як будь-яке інше значення Haskell. Розглянемо приклад:

```
todoList :: [IO ()]
todoList = [putChar 'a',
    do putChar 'b' putChar 'c',
        do c <- getChar
            putChar c]
```

Цей список насправді не викликає ніяких дій – він просто зберігає їх. Щоб з'єднати ці дії в єдину дію, можна використати функцію **sequence_**:

```
sequence_ :: [IO ()] -> IO ()
sequence_ [] = return ()
sequence_ (a: as) = do a
   sequence_ as
```

Наприклад, функція sequence_ може бути використана для конструювання **putStr** з **putChar:**

```
putStr :: String -> IO ()
putStr s = sequence_ (map putChar s)
```

Обробка виключень

Помилки кодуються з використанням спеціального типу даних **IOError**. Цей тип представляє всі можливі виключення, які можуть статися всередині монади введення-виведення. Це абстрактний тип: користувачеві не доступний жоден конструктор для IOError. Наприклад, функція:

```
isEOFError :: IOError -> Bool
```

визначає, чи відбулася помилка через досягнення кінця файлу. Те, що IOError зроблений абстрактним, дозволяє додавати нові види помилок в систему без помітних змін типу даних. Тип обробника виключень – **IOError -> IO a**.

Функція **catch** пов'язує обробник виключень з дією або множиною дій:

```
catch :: IO a -> (IOError -> IO a) -> IO a
```

Аргументами catch є дія і обробник. Якщо дія успішна, її результат повертається без виклику обробника. Якщо виникає помилка, вона передається в обробник як значення типу IOError, а потім викликається дія, пов'язана з обробником. Наприклад, getChar повертає символ нового рядка, коли виникає помилка:

```
getChar1 :: IO Char
getChar1 = getChar `catch` eofHandler where
   eofHandler e = if isEOFError e
      then return '\n'
      else ioError e
```

Використана тут функція ioError генерує виняток для наступного обробника виключень. Тип ioError:

```
ioError :: IOError -> IO a
```

Це схоже на return, за винятком того, що управління передається наступному обробнику переривань замість обробки наступної дії введення-виведення. Допустимі вкладені виклики catch, це призводить до вкладених обробникам винятків. Розглянемо приклад:

```
getLine1 :: IO String
getLine1 = catch getLine2
    (\err -> return ("Error: " ++ show err))
    where getLine2 = do c <- getChar1
        if c == '\ n'
            then return ""
            else do l <- getLine1
                return (c: l)
```

Вкладені обробники виключень дозволяють getChar1 відслідкувати кінець файлу, тоді як будь-які інші помилки повернуть з getLine1 рядок, що починається з "Error:".

Таким чином, виключення в Haskell можна розуміти наступним чином:

```
trySomething x `catch` \(e :: IOException) -> handleError e
```

або

```
catch (trySomething x) (\(e :: IOException) -> handleError e)
```

Аналог в C++ має вигляд

```
try {
    trySomething (x)}
catch (IOError e) {handleError (e)}
```

Файли

Відкриття файлу створює дескриптор (типу Handle), який використовується в транзакціях введення-виведення. Закриття дескриптора закриває пов'язаний з ним файл:

`type FilePath = String` – *шлях до файлу в файлової системі*

```
openFile :: FilePath -> IOMode -> IO Handle
hClose :: Handle -> IO ()
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

Дескриптори також можуть бути пов'язані з каналами: комунікаційні порти не прив'язуються до файлів безпосередньо. Спочатку визначено кілька каналів, включаючи **stdin** (стандартний канал введення), **stdout** (стандартний канал виведення) і **stderr** (стандартний канал виведення повідомлень про помилки).

Операції введення-виведення символьного рівня включають **hGetChar** і **hPutChar**, які приймають дескриптор як аргумент. Тоді функція getChar може бути визначена наступним чином:

```
getChar = hGetChar stdin
```

Haskell також дозволяє повернути в єдиний рядок весь вміст файлу або каналу за допомогою функції getContents:

```
getContents :: Handle -> IO String
```

Функція getContents буде зчитувати з файлу по одному символу за раз в той момент, коли вони потрібні для обчислень.

Розглянемо приклад, копіювання одного файлу в інший:

```
main = do fromHandle <- getAndOpenFile "Copy from:" ReadMode
    toHandle <- getAndOpenFile "Copy to:" WriteMode
    contents <- hGetContents fromHandle
    hPutStr toHandle contents
    hClose toHandle
    putStr "Done."
getAndOpenFile :: String -> IOMode -> IO Handle
getAndOpenFile prompt mode =
    do putStr prompt
```

```
        name <- getLine
        catch (openFile name mode)
             (\ _ -> do
                putStrLn ("Can not open" ++ name ++ "\ n")
                getAndOpenFile prompt mode)
```

Приклад роботи з введенням-виведенням

Розглянемо задачу сортування масиву методом "бульбашки" на Haskell.

```
import Array
import Ix
import IO
--Наступна функція повертає елемент якщо відомий його номер в масиві
indexArr :: (Ix i) => Array i e -> i -> e
indexArr b i = b!i
--Наступна функція змінює значення елемента з індексом і на значення е
indexArrset::(Ix i) => Array i e -> i -> e -> Array i e
indexArrset b i e = b//[(i, e)]
--Наступна функція змінює елементи з номерами і і j місцями
change :: (Ix i) => Array i e -> i -> i -> Array i e
change b i j = indexArrset (indexArrset b i (indexArr b j)) j
(IndexArr b i)
--Наступна функція сортує масив методом бульбашки
sort :: (Num e, Ix i, Ord e, Num i) => Array i e -> i -> i ->
Array i e
sort a m n = if n>= 1
    then if m>= 1
        then if a! m>a!n
            then sort (change a m n) (m-1) n
            else sort a (m-1) n
        else sort a (n-1) (n-1)
    else a
--Наступна функція вводить символи типу Integer з клавіатури
readNum :: IO Integer
readNum = readLn
```

--Наступна функція запитує елементи масиву, сортує їх, а потім виводить на екран

```
input :: Array Integer Integer -> Int -> IO ()
input a x = if x<= 5
    then do
        putStrLn "Guess a number (9 to end):"
        index <- readNum
        value <- readNum
        let c= indexArrset a index value
        print c
        if x == 5 then print (sort c 5 5) else input c (x + 1)
    else return ()
a1 = array (1,5) ([(1,0),(2,0),(3,0),(4,0),(5,0)])
e3 = input a1 1
```

2.7 Стандартні класи та похідні втілення

Стандартні класи Haskell

1) Класи рівності і впорядкованості.

До класів цієї категорії відносяться класи **Eq** і **Ord**. Клас Eq має наступний вигляд:

```
class Eq a where
    (==), (/ =) :: a -> a -> Bool
    x / = y = not (x == y)
    x == y = not (x / = y)
```

Клас **Eq** надає методи для порівняння на рівність (==) і нерівність (/=). Всі основні типи даних, за винятком функцій і **IO**, є екземплярами цього класу. Клас **Ord** має наступний вигляд:

```
data Ordering = EQ | LT | GT
class (Eq a) => Ord a where
    compare          :: a -> a -> Ordering
    (<), (<=), (> =), (>)      :: a -> a -> Bool
    max, min         :: a -> a -> a
```

```
    compare x y | x == y = EQ
                | x <= y = LT
                | otherwise = GT

    x <= y = compare x y /= GT
    x <y = compare x y == LT
    x >= y = compare x y / = LT
    x > y = compare x y == GT

    max x y     | x <= y = y
           | otherwise = x
    min x y     | x <= y = x
           | otherwise = y
```

Клас **Ord** використовується для повністю впорядкованих типів даних. Всі основні типи даних, за винятком функцій, **IO** і **IOError**, є екземплярами цього класу.

2) Клас Enum:

```
class Enum a where
    succ, pred          :: a -> a
    toEnum              :: Int -> a
    fromEnum            :: a -> Int
    enumFrom            :: a -> [a]           -- [n ..]
    enumFromThen        :: a -> a -> [a]      -- [n, n'..]
    enumFromTo          :: a -> a -> [a]      -- [n..m]
    enumFromThenTo      :: a -> a -> a -> [a] --[n, n'.. m]
```

Клас **Enum** визначає операції над послідовно впорядкованими типами. Функції succ і pred повертають відповідно наступний і попередній елемент заданого значення. Функції fromEnum і toEnum перетворять відповідно значення типу **Enum** в тип **Int** і значення типу **Int** в тип **Enum**.

Методи, що починаються з enumFrom..., використовуються при перетворенні арифметичних послідовностей. Іншими словами, функції класу **Enum** мають вигляд:

```
succ = toEnum. (+1). fromEnum
pred = toEnum. (Subtract 1). fromEnum
enumFrom x = map toEnum [fromEnum x ..]
enumFromTo x y = map toEnum [fromEnum x .. fromEnum y]
enumFromThenTo x y z = map toEnum [FromEnum x,
                                   fromEnum y .. fromEnum z]
```

3) Класи Read і Show:

```
type ReadS a = String -> [(a,String)]
type ShowS = String -> String

class Read a where
    readsPrec :: Int -> ReadS a
    readList :: ReadS [a]
```

-- *оголошення readList за замовчуванням дано в Prelude*

```
class Show a where
    showsPrec :: Int -> a -> ShowS
    show :: a -> String
    showList :: [a] -> ShowS
    showsPrec _ x s = show x ++ s
    show x = showsPrec 0 x ""
```

-- *оголошення для showList за замовчуванням дано в Prelude*

Класи **Read** і **Show** використовуються для перетворення значень до типу рядок або перетворення рядків до інших значень. Всі типи Prelude, за винятком функціональних типів і типів **IO**, є екземплярами класів **Show** і **Read**.

Найпростішою функцією класу **Show** є **show**. Вона приймає довільну величину відповідного типу і повертає її представлення у вигляді рядка символів (списку символів). Наприклад, результатом **show (2 + 2)** буде 4.

Функція **readList** – парсер повертає одноелементний список, що містить значення типу **a**, яке було зчитано з вхідного рядка, і залишок рядка, який слідує за тим, який був синтаксично розібраний. Якщо синтаксичний розбір виявився неможливим, результатом буде порожній список, а якщо рядок можна розібрати декількома способами (неоднозначність), результуючий список буде містити більше однієї пари.

Для прикладу розглянемо функцію, яка представляє двійкові дерева у вигляді рядків, яка правильним чином позначає вкладені піддерева і розділяє ліві і праві гілки.

```
showTree :: (Show a) => Tree a -> String
showTree (Leaf x) = show x
showTree (Branch l r) = "<" ++ showTree l ++ "|"
                                        ++ showTree r ++ ">"
```

Розглянемо функцію-парсер для рядкового подання бінарних дерев, що породжується showsTree:

```
readsTree :: (Read a) => ReadS (Tree a)
readsTree ('<': s) = [(Branch l r, u) | (l, '|':t) <- readsTree s,
                                        (r, '>':u) <- readsTree t]
readsTree s = [(Leaf x, t) | (X, t) <- readsTree s]
```

Якщо перший символ рядку, що аналізується, – це **'<'**, то маємо справу з поданням гілки (branch), в іншому випадку ми маємо справу з листом (leaf). У першому випадку для решти вхідного рядка **s**, що слідує за кутовою дужкою, яка відкривається, будь-який можливий синтаксичний розбір повинен бути tree Branch l r з рядком-залишком **u**.

Похідні втілення

Раніше було розглянуто втілення **Eq** для дерев:

```
Instance (Eq a) => Eq (Tree a) where
    (Leaf x) == (Leaf y) = x == y
    (Branch l r) == (Branch l 'r') = l == l '&& r == r'
    _ == _ = False
```

Варто зазначити, що такі втілення можуть бути зроблені автоматично при оголошенні data:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Eq
```

Конструкція **deriving** неявно робить оголошення втілення Eq.

Похідне втілення **Ord** для Tree є складнішим за втілення **Eq**:

```
instance (Ord a) => Ord (Tree a) where
  (Leaf _) <= (Branch _) = True
  (Leaf x) <= (Leaf y) = x <= y
  (Branch _) <= (Leaf _) = False
  (Branch l r) <= (Branch l 'r') = l == l' || r <= r ' || l <= l'
```

2.8 Монади і функтори

Монадні класи

Монада конструюється поверх поліморфного типу, такого як **IO**. Сама монада визначається оголошенням втілення, асоціюючи такий тип з одним або з усіма монадними класами: **Functor**, **Monad** і **MonadPlus**. Жоден з монадних класів не допускає утворення похідних втілень.

Крім **IO**, два інших типи з бібліотеки Prelude є членами монадних класів: списки (**[]**) і **Maybe**. З математичної точки зору монади управляються набором законів, які повинні виконуватися для монадних операцій. Ідея законів не специфічна для монад: Haskell включає інші операції, які, принаймні неформально, управляються законами. Наприклад, x /= y та not (x = = y) повинні повертати одне і те ж значення для будь-якого типу порівнюваних значень. Однак гарантії цього відсутні: і = =, і /= є окремими методами класу Eq, і немає способу переконатися в тому, що = = та /= пов'язані таким чином.

Клас **Functor** визначає одну операцію **fmap**. Функція відображення (map) застосовує операцію до об'єктів всередині контейнера (поліморфний тип може розглядатися як контейнер для значень іншого типу), повертаючи контейнер, структурно ідентичний вихідному.

Монадні закони застосовуються до операції **fmap** в класі **Functor**:

```
fmap id = id
fmap (f. g) = fmap f. fmap g
```

Ці закони гарантують, що структура контейнера не буде змінена **fmap**, і що вміст контейнера не буде переупорядкований в результаті відображаючої операції.

Клас **Monad** визначає два основних оператора: >>= (зв'язування) і **return**.

```
infixl 1 >>, >>=
class Monad m where
    (>> =) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    return :: a -> m a
    fail :: String -> m a
    m >> k = m >> = \_ -> k
```

Операції зв'язування (>> та >>=) комбінують два монадних значення, в той час як операція return вставляє значення в монаду (контейнер). Операція >>= комбінує монадне значення **m a**, що містить значення типу **a**, і функцію, яка приймає значення типу **a**, повертаючи монадне значення **m b**. Результатом є комбінування **m a** та **m b** в монадне значення, що містить **b**. Точний зміст зв'язування залежить від конкретної монади. Наприклад, в монаді IO зв'язування x >>= y послідовно виконує дві дії, передаючи результат першої дії в другу. Для інших вбудованих монад, списків і типу **Maybe**, ці монадні операції можуть розумітися в сенсі передачі нуля або більше значень з одного обчислення в наступне.

Синтаксична конструкція **do** забезпечує скорочену форму запису для ланцюжка монадних операцій. Сутність трансляції **do** може бути виражена двома наступними правилами:

```
do e1; e2 = e1 >> e2
do p <- e1; e2 = e1 >> = (\ v -> case v of p -> e2; _ -> fail "s")
```

де **"s"** – це рядок, який вказує місце розташування інструкції **do** для можливого використання в повідомленні про помилку.

Закони, що керують >>= і **return**, мають вигляд:

```
return a >>= k = k a
m >>= return  = m
(M >>= f) >>= g = m >>= (\x -> f x >>= g)
```

Клас **MonadPlus** використовується для монад, які мають елемент нуль і операцію плюс:

```
class (Monad m) => MonadPlus m where
mzero :: m a
mplus :: m a -> m a -> m a
```

Елемент нуль підпорядковується наступним законам:

```
m >>= \x -> mzero = mzero
mzero >>= m = mzero
```

Для списків значення нуль – це **[ ]**, порожній список. Монада введення-виведення не має нульового елемента і не належить цьому класу. Закони, що керують оператором **mplus**, наведені нижче:

```
m `mplus` mzero = m
mzero `mplus` m = m
```

У монаді списку оператор **mplus** – це звичайний оператор конкатенації списків.

Вбудовані монади

У випадку списків монадне зв'язування викликає об'єднання набору обчислень для кожного елемента списку. При використанні зі списками, сигнатура >>= приймає наступний вигляд:

```
(>>=) :: [a] -> (a -> [b]) -> [b]
```

Тобто у випадку заданого списку елементів типу **a** і функції, що відображає **a** на список елементів типу **b**, зв'язування застосовує цю функцію до кожного вхідного **a** і повертає всі згенеровані списки **b**, сконкатеновані в один список.

Функція **return** створює одноелементний список. Таким чином, втілення класу Monad для списку має вигляд наступним чином:

```
instance Monad [] where
    return :: a -> [a]
    return x = [x]
```

-- *створює список, що містить цей єдиний елемент*

```
(>>=) :: [a] -> (a -> [b]) -> [b]
xs >>= f = concat (map f xs)
```

-- *збирає всі результати f (які є списки) і комбінує їх в новий список*

Приклад 1:

```
[10,20,30] >>= \x -> [x, x + 1]
```

На екрані

[10,11,20,21,30,31]

Приклад 2:

```
[10,20,30] >>= \x -> [x, x + 1] >>= \y -> if y > 20
then [] else [y, y]
```

На екрані

[10,10,11,11,20,20]

Приклад 3:

```
f x  | x == '#' = "# #"
     | otherwise = ""
"#" >>= f >>= f >>= f >>= f
```

На екрані

"## ##   ## ##   ## ##   ## ##"

Розглянемо приклад 4:

```
[(X, y) | x <- [1,2,3], y <- [1,2,3], x /= y]
```

Інший варіант:

```
do x <- [1,2,3]
   y <- [1,2,3]
   True <- return (x / = y)
   return (x, y)
```

Третій варіант:

```
[1,2,3] >>= (\ x -> [1,2,3] >> = (\ y -> return (x /= y)
    >>= (\ r -> case r of True -> return (x, y) _-> fail "")))
```

Останнє визначення залежить від визначення методу **fail** в цій монаді як порожнього списку. По суті, кожна <- генерує набір значень, який передається в решту монадного обчислення.

Приклад 5:

```
mvLift2 :: (a -> b -> c) -> [a] -> [b] -> [c]
mvLift2 f x y = do x '<- x
                   y '<- y
                   return (f x 'y')
```

Використання цієї функції має вигляд:

```
mvLift2 (+) [1,3] [10,20,30] => [11,21,31,13,23,33]
mvLift2 (\a b -> [a, b]) "ab" "cd" => [ "ac", "ad", "bc", "bd"]
mvLift2 (*) [1,2,4] [] => []
```

Сенс монад і функторів

Перш за все варто зауважити, що монада – це контейнерний тип, над яким визначено кілька функцій. Монади моделюють різні види обчислень. Як і в списку, всі елементи, які містить монадний контейнер в будь-який момент часу, повинні бути одного типу.

Зазвичай Haskell використовує пару функцій return і bind (>>=), але іноді простіше розглянути функції map (fmap), return та join. Функцію bind можна визначити в термінах map (fmap), return і join.

Як відомо, функція map (fmap в Haskell 98) випливає з визначення функтора. Можна думати про функтор як про тип контейнера, в якому можна застосувати єдину функцію до кожного об'єкту. Тобто, якщо f – функтор, і є функція типу (a -> b), і контейнер типу (f a), то можна отримати новий контейнер типу (f b). Це очевидно з типу fmap:

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

(можна розшифрувати це так: *Якщо ви дасте мені ягоду лохини за кожне яблуко, яке я дам вам* (a -> b), *і у мене є коробка яблук* (f a), *то це значить, що я отримаю коробку з лохиною* (f b)). Кожна монада є функтором.

Метод return специфічний для монад. Якщо m – монада, тоді return приймає елемент типу a, і повертає контейнер типу (m a) з цим елементом в ньому. Таким чином, її тип в Haskell

```
return :: (Monad m) => a -> m a
```

(можна розшифрувати це так: *Якщо у мене є яблуко* (a), *то я можу покласти його в коробку* (m a)).

Метод join специфічний для монад, він приймає контейнер контейнерів m (m a), і комбінує їх в один m a певним осмисленим чином.

```
join :: (Monad m) => m (m a) -> m a
```

(можна розшифрувати це так: *Якщо у мене є коробка коробок яблук* (m (m a)), *то я можу взяти яблука з кожної, і покласти їх в нову коробку* (m a))

Використовуючи ці методи, можна сконструювати важливу операцію, що називається bind або extend, яка зазвичай позначається символом (>>=). Функція bind бере контейнер типу (m a) і функцію типу (a -> m b). Спочатку вона відображає функцію на контейнер (що дасть m (m b)) і потім застосовує join, повертаючи контейнер типу (m b).

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
xs >>= f = join (fmap f xs)
```

(можна розшифрувати це так: *Якщо у мене є коробка яблук* (m a), *і для кожного яблука, ви дасте мені коробку лохини* (a -> m b) *тоді я зможу отримати коробку з усіма ягодами лохини* (m b))

2.9 Модулі та абстрактні типи даних

Модулі

Програма на Haskell складається з колекції модулів. Модуль в Haskell використовується для досягнення двох цілей: для управління просторами імен і для створення абстрактних типів даних. На верхньому рівні модуль оголошення: оголошення пріоритетів, оголошення data і type, оголошення класів і втілень, сигнатури типів, визначення функцій і зв'язування шаблонів. Модулі мають алфавітно-цифрові імена, які повинні починатися з літери у верхньому регістрі. Формальний зв'язок між модулем Haskell і файловою системою, яка (як правило) забезпечує його підтримку, відсутній. Зокрема, немає зв'язку між іменем модуля й іменем файлу, і більше одного модуля можуть імовірно перебувати в одному файлі. Технічно, модуль являє собою одне велике оголошення, яке починається з ключового слова module. Розглянемо приклад:

```
module Tree (Tree (Leaf, Branch), fringe) where
    data Tree a = Leaf a | Branch (Tree a) (Tree a)
    fringe :: Tree a -> [a]
    fringe (Leaf x) = [x]
    fringe (Branch left right) = fringe left ++ fringe right
```

Цей модуль явно експортує Tree, Leaf, Branch і fringe. Якщо список експорту, який слідує за ключовим словом module, опущений, всі імена, доступні на верхньому рівні модуля, будуть експортовані. Слід зазначити, що ім'я типу та його конструктори групуються разом, як в Tree (Leaf, Branch). В якості скорочення можна написати **Tree(..)**. Є припустимим також експортувати тільки частину конструкторів. Ім'я в списку експорту не повинно бути локальним по відношенню до експортованого модуля, будь-яке ім'я в області видимості може бути включено в список експорту.

Модуль **Tree** може тепер бути імпортований в деякий інший модуль:

```
module Main (main) where
    import Tree (Tree (Leaf, Branch), fringe)
    main = print (fringe (Branch (Leaf 1) (Leaf 2)))
```

Різні елементи, імпортовані в модуль або експортовані з модуля, називаються сутностями (entities). В даному випадку вказаний явний список імпорту в оголошенні import; якщо опустити його, то це призведе до імпорту всіх сутностей, що експортуються з Tree.

Кваліфіковані імена

При імпортуванні сутностей з одного модуля в інший може виникнути проблема, якщо будуть збігатися імена імпортованої сутності і сутності, що присутня в модулі. Haskell вирішує її, дозволяючи використовувати кваліфіковані імена. Декларація імпорту може використовувати ключове слово **qualified**, при цьому імпортовані імена повинні використовувати ім'я імпортованого модуля в якості префікса. За такими префіксами повинен слідувати символ «.». Розглянемо приклад:

```
module Fringe (fringe) where
    import Tree (Tree (..))
    fringe :: Tree a -> [a]
```

Інше визначення fringe:

```
    fringe (Leaf x) = [x]
    fringe (Branch x y) = fringe x
module Main where
    import Tree (Tree (Leaf, Branch), fringe)
    import qualified Fringe (fringe)
    main = do print (fringe (Branch (Leaf 1) (Leaf 2)))
              print (Fringe.fringe (Branch (Leaf 1) (Leaf 2)))
```

Абстрактні типи даних

Крім управління просторами імен, модулі забезпечують єдиний в Haskell шлях для створення абстрактних типів даних (ADT). Ключовою властивістю ADT є те, що представляючий тип виявляється прихованим; всі операції над ADT здійснюються на абстрактному рівні, який не залежить від представлення. Наприклад, для типу Tree відповідний ADT міг би включати наступні операції:

```
data Tree a
-- тут вказується тільки ім'я типу
leaf :: a -> Tree a
branch :: Tree a -> Tree a -> Tree a
cell :: Tree a -> a left
right :: Tree a -> Tree a
isLeaf :: Tree a -> Bool
```

Модуль, який підтримує цей ADT, має вигляд:

```
module TreeADT (Tree, leaf, branch, cell, left, right, isLeaf)
where
    data Tree a = Leaf a | Branch (Tree a) (Tree a)
        leaf = Leaf
```

```
branch = Branch
cell (Leaf a) = a
left (Branch l r) = l
right (Branch l r) = r
isLeaf (Leaf _) = True
isLeaf _ = False
```

Розглянемо приклад:

```
module A (f) where
    f :: Int-> Int
    f x = x + x

module B (f) where
    f :: Int-> Int
    f x = x * x

module Main where
    import B (f)
    main :: IO ()
    main = putStrLn $ show $ f $ 10
```

На екрані отримаємо **100**.

```
module Main where
    import A (f)
    main :: IO ()
    main = putStrLn $ show $ f $ 10
```

На екрані отримаємо **20**. Тут f – абстрактний тип.

Інші властивості модулів:

- Оголошення import може вибірково приховувати суті при використанні конструкції **hiding** в оголошенні імпорту. Це корисно для явного виключення імен, які використовуються для інших цілей, при цьому відпадає необхідність використання кваліфікаторів для інших імпортованих з цього модуля імен.

- Оголошення import може містити **as** для вказівки кваліфікатора, відмінного від імені імпортованого модуля. Це можна використовувати для скорочення кваліфікаторів з модулів з довгими іменами або для того, щоб не змінювати всі кваліфіковані імена в разі зміни імені модуля.

2.10 Робота з масивами

Масиви

Масиви в функціональній мові слід розглядати просто як функції з індексів в значення. Будь-який модуль, який використовує масиви, повинен імпортувати модуль **Array**. Бібліотека **Ix** визначає клас типу для індексів масиву:

```
class (Ord a) => Ix a where
   range :: (a, a) -> [a]
   index :: (a, a) a -> Int
   inRange :: (a, a) -> a -> Bool
```

Оголошення втілень забезпечуються для типів **Int**, **Integer**, **Char**, **Bool** і кортежів типів **Ix** до 5 елементів. Крім того, похідні втілення можуть бути автоматично створені для типів перелічень і кортежів. Зазначені примітивні типи розглядаються як індекси вектора, а кортежі – як індекси для багатовимірних прямокутних масивів.

Варто зауважити, що перший аргумент кожної операції класу **Ix** – це пара індексів; вона зазвичай задає межі масиву. Наприклад, межами 10-елементного масиву з відліком від 0 і індексами типу **Int** буде пара **(0,9)**, а матриця 100 на 100 з відліком від 1 може мати межі **((1,1), (100,100))**.

Операція **range** приймає пару границь і створює впорядкований список індексів, що лежать між цими границями. Наприклад,

```
range (0,4) => [0,1,2,3,4]
range ((0,0), (1,2)) => [(0,0), (0,1), (0,2), (1,0), (1,1),
(1, 2)]
```

Предикат inRange визначає, чи лежить індекс всередині заданої пари границь. Нарешті, операція **index** дозволяє адресувати певний елемент масиву:

```
index (1,9) 2 => 1
index ((0,0), (1,2)) (1,1) => 4
```

Функція створення монолітного масиву в Haskell формує масив по його границям і списку пар індекс-значення:

```
array :: (Ix a) => (a, a) -> [(a, b)] -> Array a b
```

Наприклад, визначення масиву квадратів чисел від 1 до 100:

```
squares = array (1,100) [(i, i * i) | i <- [1..100]]
```

Звернення до елементу масиву за індексом виконується за допомогою інфіксного оператора «!», а межі масиву можуть бути отримані з функції **bounds**:

```
squares!7 => 49
bounds squares => (1,100)
```

Приклад:

```
mkArray :: (Ix a) => (a -> b) -> (a, a) -> Array a b
mkArray f bnds = array bnds [(i, f i) | i <- range bnds]
```

Багато масивів визначаються рекурсивно, тобто зі значеннями одних елементів, які залежать від значень інших:

```
fibs :: Int -> Array Int Int fibs n = a where
a = array (0, n) ([(0, 1), (1, 1)] ++ [(i, a!(i-2) + a!(i-1))
                                        | i <- [2..n]])
```

Акумуляція

Можна послабити обмеження на одноразову появу індексу в асоціативному списку, вказавши спосіб комбінування множини значень, асоційованих з одним індексом; результат називається акумульованим масивом:

```
accumArray :: (Ix a) => (b->c-> b) -> b -> (a, a) -> [(a, c)]
                                                   -> Array a b
```

Перший аргумент accumArray – це акумулююча функція, другий – початкове значення (одне й те саме для кожного елемента масиву), а аргументи, що залишилися, – це границі і асоціативний список, як у функції array.

Зазвичай акумулююча функція – це (+), а початкове значення – нуль; наприклад, наступна функція приймає пару границь та список значень і створює таблицю числа появ кожної величини всередині границь:

```
hist :: (Ix a, Integral b) => (a, a) -> [a] -> Array a b
hist bnds is = accumArray (+) 0 bnds [(i, 1) | i <- is,
                                              inRange bnds i]
```

Інкрементальні оновлення

Крім функції створення монолітного масиву, в Haskell є функція інкрементального оновлення масиву, що записується як інфіксний оператор //. У найпростішому випадку зміна і-го елемента масиву **a** на **v** записується так: **a** // **[(i, v)]**.

```
(//) :: (Ix a) => Array a b -> [(a, b)] -> Array a b
```

Розглянемо приклад – множення матриць.

```
matMult :: (Ix a, Ix b, Ix c, Num d) =>
    Array (a, b) d -> Array (b, c) d -> Array (a, c) d

matMult x y = array resultBounds
    [((I, j), sum [x! (I, k) * y! (K, j) | k <- range (lj,
    uj)])
        | i <- range (li, ui), j <- range (lj ', uj')] where
            ((Li, lj), (ui, uj)) = bounds x
            ((Li ', lj'), (ui ', uj')) = bounds y
        resultBounds
            | (Lj, uj) == (li ', ui') = ((li, lj '), (ui, uj'))
            | otherwise = error "matMult: error "
```

Тип Maybe

Конструктор типу **Maybe** визначено в Prelude наступним чином:

```
data Maybe a = Nothing | Just a
```

Призначення типу **Maybe** полягає в тому, щоб надати метод обробки неправильних або необов'язкових значень без завершення програми, що сталося б, якби використовувалася функція **error**, і без використання функції **IOError** з монади **IO**, яка вимагала б, щоб вислів став монадним.

Бібліотека **Maybe** має вигляд:

```
isJust :: Maybe a -> Bool
isJust (Just a) = True
isJust Nothing = False

isNothing :: Maybe a -> Bool
isNothing = not . isJust

fromJust :: Maybe a -> a
fromJust (Just a) = a
fromJust Nothing = error "Maybe.fromJust: Nothing"

fromMaybe :: a -> Maybe a -> a
fromMaybe d Nothing = d
fromMaybe d (Just a) = a

maybeToList :: Maybe a -> [a]
maybeToList Nothing = []
maybeToList (Just a) = [a]

listToMaybe :: [a] -> Maybe a
listToMaybe [] = Nothing
listToMaybe (a: _) = Just a

catMaybes :: [Maybe a] -> [a]
catMaybes ms = [m | Just m <- ms]

mapMaybe :: (a -> Maybe b) -> [a] -> [b]
mapMaybe f = catMaybes . map f
```

Nothing можна інтерпретувати як None. Тому, **isNothing x** перевіряє, що x is None. Протилежність Nothing – Just. Наприклад, **isJust x** перевіряє, що x is not None. Розглянемо приклад:

```
maybe someDefault (\x -> ...) mx
```

На С ++ це виглядало би як

```
if (mx is None) {
    x = someDefault}
else {x = mx ...}
```

Більш специфічний варіант для випадку, коли null – це помилка:

```
maybe (error "bad value!") (\ x -> ...) x
```

На Python маємо:

```
if (x is None)
    raise Exception ("bad value!")
```

2.11 Додаткові теми по Haskell

Монада State

Обчислення, що бере певне початкове значення (типу **s**) і повертає кортеж, що містить результуюче значення обчислення (типу **a**) і новий стан (типу **s**), може бути виражено функцією вигляду **s -> (a, s)**.

Наприклад, **s -> ("foo", s)** може бути обчисленням, яке повертає "foo" як результуюче значення і не змінений стан. Для абстрагування можна обернути ці обчислення в наступний тип:

```
data State s a = State {runState :: s -> (a, s)}
```

Функція runState

Функція runState приймає стан обчислення і початкове значення в якості параметрів. Вона «витягає» обчислення з типу State і просто застосовує його до початкового значення.

```
runState :: State s a -> s -> (a, s)
runState (State f) s = f s
```

Функції get, put, return

get – це «обчислення зі змінним станом, яке повертає сам стан».

```
get :: State s s
get = State $ \ s -> (s, s)
```

put – це «обчислення зі змінним станом, що не повертає нічого, проте змінює стан».

```
put :: s -> State s ()
put s '= State $ \s -> ((), s')
```

return a – повертає обчислення зі станом, завжди повертає вказане значення «а».

Розглянемо приклад – обчислення n!. Реалізація без монади State:

```
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Реалізація з монадою State:

```
import Control.Monad.State
factorl :: Int-> State Int Int -- тип стану – Int, тип результату – теж Int
factorl 0 = do
    acc <-get -- отримуємо накопичений результат
    return acc -- повертаємо його
factorl n = do
    acc <- get -- отримуємо акумулятор
    put (acc * n) -- домножуємо його на n і зберігаємо
    factorl (n-1) -- продовжуєм обчислення факторіала
fact :: Int-> Int
fact n = fst $ runState (factorl n) 1
main = putStrLn $ show $ fact 5
```

Починаючи з версії 2.0.0.0 і вище пакета mtl змінилася реалізація монади State: модуль Control.Monad.State більше не експортує конструктор State.

Потрібно замінити цей конструктор функцією **state :: (s -> (a, s)) -> State s a**, яка робить ту ж саму роботу.

Монада ST

Дана монада знаходиться в модулі Control.Monad.ST. Монада ST реалізує роботу зі змінним станом, але в абсолютно функціональному стилі, зберігаючи всі важливі властивості і принципи функціональної парадигми: незалежність результату від порядку обчислень, прозорість посилань, лінивість, можливість вилучення об'єктів з монади ST (тобто, можливість отримати «чисте» значення в результаті обчислення зі станом у монаді ST). Тобто монада ST здійснює роботу з пам'яттю в функціональному стилі. Іншими словами, в мові Haskell є чисто функціональний інструмент для ефективного вирішення імперативних завдань.

У даного типу є два параметри і немає конструкторів:

```
data ST s a
```

Тут тип a повертається «дією» типу s. Сигнатура функції, що дозволяє отримати значення з монади ST, має вигляд

```
runST :: (forall s . ST s a) -> a
```

Аналогом квантора загальності є **forall**. Розглянемо приклад:

```
map :: forall a, b . (A-> b) -> [a] -> [b]
```

Це означає, що **map** є поліморфним алгоритмом, який можна спеціалізувати будь-якими значеннями цих a і b. Наприклад, її можна використовувати в контексті, що вимагає функції типу **(Int -> Int) -> [Int] -> [Int]** або вимагає функції типу **(Char -> Bool) -> [Char] -> [Bool]**.

forall також можна використовувати для визначення нового типу. Наприклад, визначимо тип:

```
data GeoObject = forall a . Figure a => GeoObject a
```

Тоді за допомогою типу GeoObject можна визначити неоднорідний список об'єктів, що складають географічну карту:

```
geolist :: [GeoObject]
```

```
geolist = [GeoObject (Point (0, 0)), GeoObject (Line (1, 1)
(2, 2)), GeoObject (Polygon [(1, 1), (1, -1), (-1, - 1), (-1,
1)]), GeoObject (Ellipse (0, 0) 1 + 1)]
```

Для використання forall необхідно підключити мовні розширення, скориставшись командою

```
{-# LANGUAGE ExistentialQuantification #-}
```

Також для типу ST визначено ряд методів:

```
newSTRef :: a -> ST s (STRef s a)
readSTRef :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
```

Це функції запису в іменований стан, читання і зміни. Проілюструємо на прикладах.

Приклад 1.

```
import Data.STRef
import Control.Monad
import Control.Monad.ST
productST :: Num a => [a] -> a
productST xs = runST $ do n <- newSTRef 1
    mapM_ (\x -> modifySTRef n (* x)) xs
        readSTRef n
```

Результат виконання цієї послідовності знаходиться в монаді ST, тому для отримання «чистого» значення з цієї монади використовується функція runST. Перша дія – запис в іменований стан n початкового значення 1. Далі виконуються циклічні обчислення за допомогою функції mapM_, для кожного елемента x зі списку xs виконується дія modifySTRef n (* x). Ця дія модифікує значення «змінної» n за допомогою заданої функції (* x). Імперативний аналог цього фрагмента коду – n = n * x.

Приклад 2. Реалізація циклу for:

```
result forLoop (result init, int begin) {
    result s = init;
    for (i = begin; i <n; i ++)
        update (i, s);
```

```
        return s;
    }
```

На Haskell цей цикл матиме наступний вигляд:

```
    import Control.Monad
    import Data.STRef
    import Control.Monad.ST
    forLoop :: i -> (i -> Bool) -> (i -> i) -> (i -> s -> s) -> s
    -> s
    forLoop i0 pred next update s0
        = RunST $ do
            refI <- newSTRef i0
            refS <- newSTRef s0
            iter refI refS
            readSTRef refS
            where iter refI refS
                = do
                    i <- readSTRef refI
                    s <- readSTRef refS
                    when (pred i) $ do
                        writeSTRef refI $ next i
                        writeSTRef refS $ update i s
                        iter refI refS
```

Функція when – це стандартна функція з модуля Control.Monad. Вона перевіряє предикат, і якщо він повертає True, виконує серію дій, в яких записуються оновлені значення. Виклик цієї функції може мати вигляд:

```
    Prelude> forLoop 1 (<= 10) succ (+) 0
```

На екрані **55**.

Приклад 3. Цикл while:

```
    result whileLoop (result begin) {
        result s = begin;
        while (pred (s))
            update (s);
        return s;
    }
```

На Haskell даний цикл має наступний вигляд:

```
whileLoop :: (s -> Bool) -> (s -> s) -> s -> s
whileLoop pred update s0
    = RunST $ do
        ref <- newSTRef s0
        iter ref
        readSTRef ref
        where iter ref
            = do
                s <- readSTRef ref
                when (pred s) $ do
                    writeSTRef ref $ update s
                    iter ref
```

Використання даної функцію має вигляд:

```
Prelude> whileLoop ((>0). fst)
                  (\(n, s) -> (pred n, n + s)) (10, 0)
```

Результат **(0,55)**.

Розширення

Розширення підключаються за допомогою спеціального коментаря. Такий коментар називається *прагмою* (pragma). Він поміщається на початку модуля. Розширення діє тільки в поточному модулі.

```
{-# LANGUAGE ExtentionName1, ExtentionName2, ExtentionName3 #-}
```

1. Прагма **LANGUAGE.** Слово LANGUAGE говорить компілятору про те, що ми хочемо скористатися розширеннями з іменами ExtentionName1, ExtentionName2, ExtentionName3.

2. Прагма **INLINE.** Дана прагма вказує компілятору на те, що на етапі спрощення програми необхідно замінити виклик функції на її праву частину. Цей процес називають вбудовуванням функцій. Заміна буде проведена тільки в разі повного застосування функції, якщо синтаксична арність (кількість аргументів зліва від знаку рівності) збігається з числом переданих у функцію аргументів. Приклад:

```
{-# INLINE (.) #-}
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g = \x -> f (g x)
```

3. **Прагма RULES**. За допомогою даної прагми можна визначати специфічні для застосування правила оптимізації. Наприклад, додаткові правила для функції map.

```
{-# RULES
  "map/compose" forall f g x. map f (map g x) = map (f . g) x
  "map/id" map id = id
# -}
```

Першим в лапках йде ім'я правила. Воно використовується тільки для підрахунку статистики (приміром, якщо необхідно дізнатися, скільки правил спрацювало в даному прогоні програми). За ім'ям правила пишеться рівняння. В одній прагмі може бути кілька рівнянь. Правила розділяються крапкою з комою або переходом на інший рядок. Всі вільні змінні правила перераховуються в оточенні forall (...). Відбувається тільки перевірка типів. Ніяких інших перевірок не проводиться.

Слід зазначити, що прагма RULES застосовується до тих пір, поки є можливість її застосовувати, при цьому можна увійти в нескінченний цикл:

```
{-# RULES "infinite" forall a b. f a b = f b a #-}
```

4. Прагма **UNPACK**. Прагма UNPACK дозволяє нам боротися з зайвими об'єктами типу CON (об'єкти купи). Значення в Haskell містять додаткову службову інформацію, яка необхідна на етапі обчислення, наприклад, значення спочатку було відкладеним, потім до нього дісталися і порахували, можливо воно виявилося не визначеним значенням (undefined). Такі значення називаються запакованими (boxed). Незапаковане значення – це примітивне значення, як воно представлено в пам'яті комп'ютера. Наприклад, визначення цілих чисел:

```
data Int = I# Int#
```

Всі незапаковані значення пишуться з решіткою на кінці. Запаковані значення дозволяють відкладати обчислення, користуватися undefined при визначенні функції. Однак в цьому випадку збільшується витрата пам'яті.

За допомогою прагми UNPACK можна відмовитися від лінивої гнучкості на користь меншої витрати пам'яті. Ця прагма дозволяє вбудовувати один конструктор в поле іншого. Це поле повинно бути строгим (з поміткою !) і мономорфним (тип поля повинен бути конкретним типом, а не параметром), причому підлеглий тип має містити лише один конструктор (у нього немає альтернатив):

```
data PairInt = PairInt
    {- # UNPACK #-}! Int
    {- # UNPACK #-}! Int
```

В стандартних бібліотеках передбачено багато незапакованих типів. Зокрема, це спеціальні кортежі. Вони пишуться з решітками:

```
data ST s a = ST (STRep s a)
type STRep s a = State # s -> (# State# s, a #)
```

Тут (# .... #) – спеціальні кортежі. Спеціальні кортежі використовуються для повернення декількох значень безпосередньо, без створення проміжного кортежу в купі. У цьому випадку значення будуть збережені в регістрах або на стеку.

Завдання з Haskell

1. На шахівниці розміром n×n на заданому полі стоїть шаховий кінь. Знайдіть такий шлях коня на шахівниці, щоб у кожній клітинці він побував єдиний раз. Кінь ходить за звичайними шаховими правилами.
2. Створіть багатошаровий персептрон з навчанням методом зворотного поширення помилки. Перевірте його роботу на прикладі обчислення XOR.
3. Розв’яжіть задачу комівояжера, використовуючи генетичний алгоритм.
4. Використовуючи бібліотеку для роботи з базами даних HaskellDB, створіть програму з графічним інтерфейсом, яка виводить вміст таблиці з БД і дозволяє його змінювати.
5. Для заданого зв’язного графу визначити всі шляхи з однієї обраної вершини до іншої.
6. Створіть чат з графічним інтерфейсом. Використайте сокети та бібліотеку qtHaskell.
7. Використовуючи веб-фреймворк Happstack, реалізуйте програму, що генеруватиме таблицю за переданими заданими параметрам: заголовок, кількість рядків та стовпців, колір тла.
8. Використовуючи OpenGL, створіть додаток на Haskell, що моделює зіткнення двох куль.
9. Стеганографія – приховування змісту повідомлення без явних ознак наявності цього повідомлення. Зокрема, можна вбудовувати інформацію у графічні файли без помітних змін в зображенні (нижче порогу чутливості людини). Напишіть програму для приховування інформації у GIF-файлі: приховані дані будуть записуватися в останній значущий біт (метод LSB – Least Significant Bit). Використовуйте графічну бібліотеку GD.
10. Створіть Haskell-додаток для ініціалізації масиву об'єктів інформацією з файлу XML. Здійсніть перевірку XML-документа із залученням DTD та XSD. Для роботи з XML використайте бібліотеку HaXML.

Створіть файл XML та відповідне йому DTD-визначення наступної тематики: тарифи мобільних компаній. Має бути описано 4–5 тарифів.

Тарифи мобільних компаній можуть мати таку структуру:

- Name – назва тарифу.
- Operator name – назва мобільного оператора, якому належить тариф.
- Payroll – абонентська плата на місяць (0 – n гривень).
- Call prices (має бути декілька) – ціни на дзвінки: всередині мережі (0 – n гривень за хвилину), на інші мобільні мережі (0 – n гривень за хвилину), на стаціонарні телефони (0 – n гривень за хвилину).
- SMS price – ціна за смс (0 – n гривень).
- Parameters (має бути декілька) – наявність улюбленого номера (0 – n), спосіб тарифікації (посекундна, похвилинна), плата за підключення до тарифу (0 – n гривень) тощо.

Кореневий елемент називається Tariff.

Частина III. Лямбда-числення

3.1 Вступ до λ-числення. Обчислення виразів

Лямбда-числення

Лямбда-числення (або λ-числення) – формальна система, що використовується для дослідження визначення функції, застосування функції та рекурсії. Це числення було запропоноване Алонсо Чорчем (Alonzo Church) та Стівеном Кліні (Stephen Kleene) в 1930-ті роки як частина намагання розробити базис математики на основі функцій, а не множин (однак спроба уникнути перешкод, таких як парадокс Рассела, все одно виявилася неуспішною). Незважаючи на це, лямбда-числення стало зручним інструментом в дослідженні обчислюваності функцій та лягло в основу парадигми функціонального програмування.

Як і машину Тьюрінга, лямбда-числення можна розглядати як ідеалізовану, мінімалістичну абстракцію (мову програмування), здатну визначати будь-який алгоритм. Відмінність між ними полягає в тому, що лямбда-числення відповідає функціональній парадигмі визначення алгоритмів, а машина Тьюрінга – імперативній: машина Тьюрінга має певний «стан», що може змінюватись із кожною наступною інструкцією, лямбда-числення має справу з функціями, котрі отримують значення параметрів та повертають результати обчислень (можливо, інші функції), але не спричиняють зміни вхідних даних.

Лямбда-числення – це обчислення анонімних (безіменних) функцій:

- по-перше, воно дає метод представлення функцій;
- по-друге, множину правил виведення для синтаксичного перетворення функцій.

Розглянемо просту функцію:

func x = 2 * x

Щоб уявити цю функцію в нотації лямбда-числення, необхідно опустити ім'я функції "func", роблячи її анонімною. Вона буде представлена в вигляді λ-виразу наступним чином:

λx. * 2x (або \ **x. * 2x**)

Використання вбудованих функцій записується в префіксному порядку. Такий лямбда-вираз можна читати: символ λ, як «функція від чого», і крапку (.), як «яка повертає». Лямбда-вирази такої форми називаються лямбда-абстракціями.

Символ x після λ називається **зв'язаною змінною** абстракції і відповідає поняттю формального параметра в традиційній процедурі або функції. Вираз справа від крапки називається тілом абстракції, і подібно коду традиційної процедури або функції, воно описує, що потрібно зробити з параметром, який надійшов на вхід функції.

Тіло абстракції може бути будь-яким допустимим лямбда-виразом, і тому воно також може містити іншу абстракцію, наприклад:

λx. λy. * (+ x y) 2

Це вираз читається як «функція від x*,* яка повертає функцію від y*,* яка повертає (x + y) * 2 . В мові Haskell дана функція має вигляд:

```
\x -> \y -> (x+y)*2
```

До функцій в лямбда-нотації застосуємо каррінг (метод обчислення функції від багатьох аргументів шляхом перетворенням її в послідовність функцій одного аргумента), тому всі вони мають тільки один аргумент.

Тому всі абстракції включають тільки один символ λ і єдину зв'язану змінну.

Строго кажучи, це означає, що застосування функцій повинні міститися в дужках, наприклад:

$$\mathbf{(\ldots (((\lambda x_1.\ \lambda x_2.\ \ldots .\ \lambda x_n.E\ )x_1)x_2)\ldots\ x_n)}$$

Однак існує угода, згідно якої функція застосовується завжди до найлівішого аргументу, і тому дужки навколо кожного вкладення можна опустити без зміни сенсу виразу :

$$(\lambda x_1 . \lambda x_2 . \ldots . \lambda x_n . E)\ x_1\ x_2 \ldots x_n$$

Обчислення лямбда-виразів

Для обчислень лямбда-виразів необхідні правила виведення. Найпростіший тип лямбда-виразу – це константа.

Константи є **самовизначеними,** тобто їх неможливо перетворити в простіші вирази. Наприклад, обчислення цілої константи 5 дає ту ж саму константу 5.

δ – правило: застосування константної функції може бути перетворено при наявності достатньої кількості аргументів з використанням вбудованих правил.

Розглянемо приклад. Нехай є константний вираз:

+1 3

Даний вираз може бути перетворено (тобто застосована δ редукція) з урахуванням вбудованого правила +:

+1 3 -> 4

Наведений приклад редукції можна прочитати так:

« **+1 3** перетворюється в **4** »

У загальному випадку аргументи функції можуть не бути в потрібній формі, і тому виконати редукцію неможливо, наприклад:

*** (+ 1 2) (– 4 1)**

Щоб перетворити цей вираз, необхідно спочатку перетворити вираз аргументів функції *. Обчислення його може відбуватися наступним чином :

*** (+ 1 2) (– 4 1)**

-> * (+1 2) 3

-> * 3 3

-> 9

β – правило: описує застосування лямбда-абстракції.

Розглянемо наступний вираз:

(λx. + x x) 2

Ситуація аналогічна виклику процедури або функції мови високого рівня з фактичним параметром, що заміщує її формальний параметр. В лямбда-численні така заміна є текстовою, і ми таким чином фізично замінюємо всі входження зв'язаної змінної x в тілі застосованої лямбда-абстракції на вираз аргументу 2, отримуючи в результаті змінену форму тіла. Отже:

(λx. + x x) 2 -> + 2 2 (це і є β-редукція).

Процес називається редукцією в тому сенсі, що в даному випадку спрощується вираз, з нього прибираються символ λ*,* зв'язана змінна і вираз аргументу. В підсумку виходить змінена форма тіла лямбда-абстракції. Редукція абстракції, застосованої до деякого аргументу, може дати в результаті іншу абстракцію, і в цьому випадку процес може бути продовжений.

Наприклад, в наступному виразі

((λx. λy . + x y) 7) 8

редукція починається з підстановки числа 7 замість *x* в тіло самої зовнішньої абстракції (тобто λy. + x y), при цьому виходить

(λy. + 7 y) 8

і закінчується застосуванням отриманої абстракції до виразу аргументу 8, при цьому отримуємо

+ 7 8

Останній вираз перетворюється в 15 за допомогою δ-правила.

Вираз, до якого можна застосувати редукцію, називається **редексом**, і тому можна зробити висновок, що процес редукції лямбда-виразу складається з множини редукцій, що застосовуються до редексу до тих пір, поки вираз містить хоча б один редекс.

Термін δ-редекс відповідає виразу, який може бути спрощено за допомогою δ-правила, а β-редексом називається вираз, який можна спростити за допомогою β-редукціі.

α – правило: заміна одного імені змінної на інше.

Розглянемо приклад:

$$(\lambda x. (\lambda x.x) (+ 1 x)) 1$$

Тут x в тілі внутрішньої абстракції (λx.x) й x у виразі аргументу (+ 1 x) – це різні змінні, що мають однакове ім'я.

Якщо застосувати цей вираз до аргументу (наприклад, 1), то наступне виконання β-редукції буде помилкою:

(λx. (λx.x) (+1 x)) 1

-> (λx.1) (+1 +1)

-> 1

Таким чином, тут необхідне α-перетворення:

(λx. (λx.x) (+1 x)) 1

-> (λx. (λy.y) (+1 x)) 1

-> (λy.y) (+1 1)

-> (λy.y) 2

-> 2

Розглянемо складніший приклад:

(λf.λx. f 4 x) (λy.λx. + x y) 3

- > (λf.λz. f 4 z) (λy.λx. + x y) 3

-> (λf.λz. f 4 z) (λy.λx. + x y) 3

-> (λz. (λy.λ x. + x y) 4 z) 3

-> (λy.λx. + x y) 4 3

-> (λx. + x 4) 3

-> + 3 4

-> 7

Важливо зауважити, що на кроці 3 був вибір між двома редексами, і для перетворення довільно був взятий зовнішній редекс. Однак можна було б застосувати β-редукцію і до внутрішнього редексу, що призвело б до наступного ланцюжку перетворень:

(λf.λx. f 4 x) (λy.λx. + x y) 3

- > (λf.λz. f 4 z) (λy.λx. + x y) 3

-> (λf.λz. f 4 z) (λy.λx. + x y) 3

-> (λz. (λy.λx. + x y) 4 z) 3

-> (λz. (λx. + x 4) z) 3

-> (λx. + x 4) 3

-> + 3 4

-> 7

η-правило: два висловлення (λx. Ex) і E еквівалентні за умови, що x не є вільною змінною в E, тому

(λx. Ex) A -> E A

для будь-якого виразу A . Це називається **функціональною екстенсивністю**. Таким чином, η-перетворення записується у вигляді

(λx. Ex) - > E

3.2 Нормальні форми, редукції та механізми виклику

Порядок редукції і нормальні форми

Кажуть, що лямбда-вираз знаходиться в **нормальній формі**, якщо до нього не можна застосувати правило редукції. Іншими словами, лямбда-вираз знаходиться в нормальній формі, якщо він не містить редексів.

Нормальна форма відповідає поняттю завершення обчислень в традиційному програмуванні. Звідси випливає наступна схема обчислень:

while існує хоча б один редекс

do перетворити один з редексів

end

{Вираз тепер в нормальної формі}

Проблема, пов'язана з такою схемою, полягає в тому, що у виразі може бути кілька редексів і незрозуміло, який з них вибрати для перетворення. Розглянемо приклад :

$$(\lambda x.\lambda y.y) ((\lambda z.z\ z) (\lambda z.z\ z))$$

В цьому виразі два редекси:

$$(\lambda z.z\ z) (\lambda z.z\ z)$$

$$(\lambda x.\lambda y.y) ((\lambda z.z\ z) (\lambda z.z\ z))$$

Вибравши перший з них, отримаємо ланцюжок редукцій, який ніколи не закінчиться:

$$(\lambda z.z\ z) (\lambda z.z\ z)$$

$$\to (\lambda z.z\ z) (\lambda z.z\ z)$$

$$\to (\lambda z.z\ z) (\lambda z.z\ z) \text{ і т.д.}$$

Вибравши другий, отримаємо редукцію, яка закінчується за один крок:

$$(\lambda x.\lambda y.y) ((\lambda z.z\ z) (\lambda z.z\ z))$$

$$\to \lambda y.y$$

Це призводить до необхідності розгляду **порядку редукції**, що визначає в разі виникнення кількох редексів, який з них вибрати для перетворення.

Найлівішим редексом називається редекс, символ λ якого (або ідентифікатор примітивної функції у разі δ-редекса) розташований у λ-виразі лівіше всіх інших редексів. (Аналогічно визначається **найправіший** редекс).

Самим зовнішнім редексом називається редекс, який не міститься усередині ніякого іншого вкладеного редекса.

Самим внутрішнім редексом називається редекс, що не містить інших редексів.

В контексті функціональних мов і λ-числення існують два важливих порядки редукцій:

Аплікативний порядок редукцій (АПР), який передбачає спочатку перетворювати найлівіший з самих внутрішніх редексів.

Нормальний порядок редукцій (НПР), який передбачає спочатку перетворювати найлівіший з самих зовнішніх редексів.

Тоді з попереднього прикладу видно, що найлівішим з самих внутрішніх редексів є

$$\mathbf{(\lambda z.z\ z)\ (\lambda z.z\ z)}$$

а найлівішим з самих зовнішніх є

$$\mathbf{(\lambda x.\lambda y.y)\ ((\lambda z.z\ z)\ (\lambda z.z\ z))}$$

Тоді при аплікативному порядку редукцій першим буде обчислюватися редекс ((λz.z z)(λz.z z)) і обчислення ніколи не закінчиться, тоді як при нормальному порядку редукцій обчислення λ-виразу закінчиться за один крок.

НПР дозволяє ефективно відкладати обчислення будь-яких редексів всередині виразу аргументу до тих пір, поки це можливо, в розрахунку на те, що таке обчислення може виявитися непотрібним. Стратегія вибору найлівішого із самих зовнішніх редексів передбачає виконання підстановки для x в λy.y до будь-якого перетворення виразу аргументу. В результаті за один крок виходить нормальна форма λy.y .

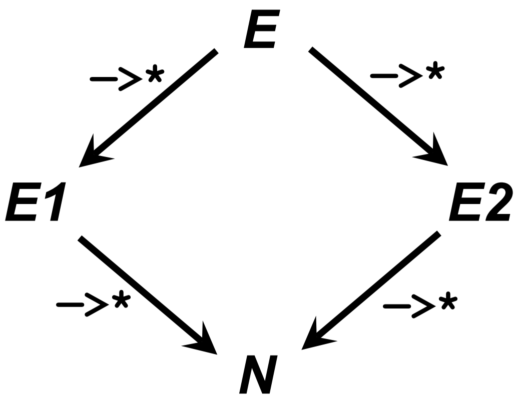
АПР, з іншого боку, обчислює вираз аргументу в першу чергу, що в даному випадку призводить до зациклення.

Таким чином, НПР і АПР відповідають лінивому і енергійному обчисленню.

Фундаментальні теореми лямбда числення.

Теорема Чорча-Россера (наслідок). Якщо вираз *E* може бути приведено двома різними способами до двох нормальних форм, то ці нормальні форми є алфавітно-еквівалентними.

Теорема стандартизації. Якщо вираз *E* має нормальну форму, то



редукція найлівішого із самих зовнішніх редексів на кожному етапі обчислення *E* гарантує досягнення цієї нормальної форми (з точністю до алфавітної еквівалентності).

Ромбічна властивість відношення редукції **->** полягає в тому, що якщо вираз *E* може редуцією бути приведений до двох виразів *E1* і *E2,* то існує вираз *N ,* який можна отримати (повторно застосовуючи **->**) як з *E1*, так і з *E2*.

Символ **->*** позначає довільне число редукцій. Тобто **->*** є **рефлексивно-транзитивним замиканням** відношення **->**.

β - редукція і проблема конфлікту імен

Проблему конфлікту імен можна проілюструвати на наступному прикладі:

λx. ((λy.λx. + x y) x)

Цей вираз містить єдиний β-редекс. Якщо спробувати обчислити цей редекс, то отримаємо

λx. (λx. + x x)

що, очевидно, є помилкою.

Проблема полягає в тому, що в цьому прикладі вираз аргументу містить вільну змінну, що має однакове ім'я з однією із зв'язаних змінних в тілі абстракції. Щоб безпечно виконувати β-редукцію, необхідно спочатку модифікувати абстракцію так, щоб всі імена зв'язаних змінних стали унікальними по відношенню до вільних змінних у виразі аргументу, тобто виконати α-перетворення. Але α-перетворення є досить ресурсномістким.

Виявляється, що проблему можна вирішити без α-перетворення, визначивши обмежену нормальну форму – так звану **слабку заголовну нормальну форму**.

Вираз *E* знаходиться в слабкій заголовній нормальній формі (СЗНФ), якщо:

1) *E* є константою,

2) *E* є виразом виду λx.*E′* для будь-якого *E′*,

3) *E* має форму $PE_1E_2 \ldots E_n$ для будь-якої константної функції *P* арності $k > n$.

Третє правило стверджує, що будь-яка частково застосована константна функція також є СЗНФ. Наприклад, вираз

*** 3**

можна переписати, використовуючи η-редукцію:

λx. * 3 x

який є СЗНФ.

Перевага приведення виразів до СЗНФ замість повної нормальної форми полягає в тому, що немає необхідності застосовувати β-редукцію за присутності вільних змінних. Використовуючи попередній приклад

λx. ((λy.λx. + x y) x)

бачимо, що вираз знаходиться в СЗНФ. Тобто даний редекс обчислювати не треба .

Тільки коли вираз застосовується до аргументу, обчислення має місце, і це обчислення почнеться з видалення найлівішої зв'язаної змінної x відповідно до правил редукції нормального порядку, наприклад:

$$(\lambda x.\ ((\lambda y.\lambda x.\ +\ y\ x)\ x))\ 4$$
$$\to (\lambda y.\lambda x.\ +\ y\ x)\ 4$$
$$\to \lambda x.\ +4\ x$$

Варто зауважити, що в проміжку між нормальною формою і СЗНФ розташована так звана **заголовна нормальна форма** (ЗНФ).

Кажуть, що вираз *E* знаходиться в ЗНФ , якщо:

1) *E* є константою,

2) *E* є виразом виду $\lambda x_1.\ \lambda x_2 \ldots \lambda x_n.E'$, де *E* не є редексом,

3) *E* має форму $PE_1E2 \ldots E_n$ для будь-якої константної функції *P* арності $k > n$.

Зазвичай, приведення до ЗНФ не практикується, тому що в цьому випадку не вдається уникнути конфлікту імен.

Ефект поділу

Розглянемо приклад:

$$(\lambda x.\ +\ x\ x)\ E$$

Застосувавши до нього правило β-редукціі, отримаємо

$$+\ E\ E$$

Щоб перетворити цей вираз, необхідно спочатку перетворити аргументи функції +, а це означає, що над виразом E двічі має бути виконана редукція. Це обчислення дуже неефективне. Один спосіб, який гарантує, що аргумент ніколи не буде обчислюватися більш одного разу, полягає в обчисленні його перед виконанням β-редукції. Це відповідає аплікативному порядку редукції. Замість підстановки аргументу у вихідному вигляді, виконується підстановка значення аргументу, тому повторні входження зв'язаної змінної не призводять до повторної редукції аргументу.

Якщо необхідно реалізувати нормальний порядок редукції, то певним чином треба зробити так, щоб перший обчислений (приведений до СЗНФ) вираз аргументу «розділився» між усіма іншими виразами цього аргументу тобто замінив кожне з них своїм значенням. Це означає, що необхідно відійти від моделі редукції, запропонованої в λ-численні, де обчислення виразів записується в чисто текстовій формі.

Це означає, що можна записувати проміжні вирази, що формують послідовність редукції, за допомогою набору символів на звичайному аркуші паперу (такий процес іноді називають **редукцією рядків**).

Якщо необхідно вивчити ефект поділу, то слід встановити факт, що два входження одного і того ж ідентифікатора відносяться до одного і того ж висловлення.

Один з способів зробити це полягає в поданні виразів в вигляді графів. Це призводить до ідеї **редукції графів**. У такому випадку варто дотримуватися наступної домовленості. Маючи вираз вигляду

(λx. ... x ... x . ..) E

результат його β-редукції будемо записувати у вигляді

... x ... x ..., де **x** є **E**

Зв'язок між іменами змінних і їх значеннями зазвичай називається **контекстом**. Сенс такого підходу в тому, що коли x (тобто E) обчислюється вперше, скорочена форма E замінює не тільки x у виразі, але й E в контексті цього виразу. Розглянемо приклад :

(λx. + x x) (+ 1 2)

- > **+ x x** де **x** є **(+1 2)**

-> **+ 3 x** де **x** є **3**

-> **+ 3 3**

-> **6**

Схеми редукції і механізми виклику

В функціональних мовах програмування виклик по значенню відомий як енергійне обчислення, яке в λ-численні ми можемо визначити наступним чином.

Енергійне обчислення = АПР, що приводить вираз до СЗНФ.

В якості альтернативи АПР можна використовувати НПР.

В λ-численні, де відсутнє поняття поділу, така підстановка є чисто текстовою. Тому вона відповідає поняттю **виклику по текстовій** заміні**.** Очевидно, при цьому може виникнути конфлікт імен та необхідність повторного обчислення виразів аргументів.

Якщо реалізується нормальний порядок редукцій та підтримується поділ, тоді механізм редукції відповідає виклику по необхідності. Таким чином, виклик по необхідності можемо охарактеризувати як нормальний порядок редукції, що приводить вираз до СЗНФ разом з поділом виразів аргументів. Отже, ліниве обчислення значення λ-виразу можна охарактеризувати наступним чином.

Ліниве обчислення = НПР , що приводить вираз до СЗНФ + поділ + ліниві конструктори

Або еквівалентно.

Ліниве обчислення = виклик по необхідності + ліниві конструктори .

3.3 Рекурсивні функції

Рекурсивні вирази

Розглянемо проблему рекурсивних функцій в λ-численні, де функції не мають імен. Очевидно, що для подання рекурсії необхідний метод, який дозволяє функції викликати себе не по імені, а якимось іншим чином.

Для представлення рекурсії зручно використовувати погляд на рекурсію, при якому рекурсивну функцію можна представити як функцію, що приймає саму себе в якості аргументу. У цьому випадку функція може виявитися зв'язаною з однією з власних змінних та буде, таким чином, містити в своєму тілі посилання на саму себе.

Розглянемо приклад . Нехай є рекурсивна функція:

```
sum :: Int -> Int
sum n = if n == 0 then 0 else n + sum (n-1)
```

Цей вираз може бути представлено в вигляді λ-абстракції, що має додатковий параметр, який при застосуванні цієї абстракції зв'язується з самою функцією:

SUM = λs. λn.COND (= n 0) 0 (+ n (s (– n 1)))

Тепер необхідно зв'язати змінну s із значенням функції SUM. Це можна зробити, використавши спеціальну функцію що називається **Y-комбінатором,** яка задовольняє наступне рівняння:

Yf = f(Y f)

Тут Y також відомий як комбінатор **нерухомої точки.**

«Нерухома точка» функції f – це вираз, який не змінюється при застосуванні до нього функції f. Функція може мати кілька нерухомих точок; наприклад, функція тотожності λx.x має їх нескінченну кількість. Вираз Yf дає **найменшу нерухому точку** функції f .

Розглянемо застосування Y до функції SUM:

Y SUM

= Y (λs. λn.COND (= n 0) 0 (+ n (s (– n 1))))

-> (λs. λn.COND (= n 0) 0 (+ n (s (– n 1)))) (Y SUM)

-> (λn.COND (= n 0) 0 (+ n ((Y SUM) (– n 1))))

Розписавши внутрішній вираз Y SUM маємо:

-> **(λn.COND (= n 0)**

0

(+ n ((λn.COND (= n 0) 0 (+ n ((Y SUM) (– n 1)))) (– n 1))))

Бачимо, що вираз поводить себе так само, як вихідне рекурсивне визначення sum. Внутрішнє входження Y SUM конструює копію вихідної функції SUM і розміщує саме себе (тобто Y SUM) замість s в тіло копії.

Таким чином, функція sum може бути записана в λ–численні в вигляді Y SUM, тобто

Y (λs. λn.COND (= n 0) 0 (+ n (s (– n 1))))

У загальному випадку рекурсивна функція f з тілом, що задається виразом E, записується в λ-численні в вигляді

Y (λf.E)

Взаємна рекурсія

У загальному випадку можна визначити набір взаємно рекурсивних функцій наступним чином :

$$\mathbf{f_1\ (...) = E_1}$$

$$\mathbf{f_2\ (...) = E_2}$$

....

$$\mathbf{f_n\ (...) = E_n}$$

де E_i може посилатися на будь-яку (або на всі відразу) функцію f_j ($1 \le i, j \le n$). Для реалізації даних визначень використовуються кортежі. Ідея полягає в тому, щоб упакувати набір з n взаємно рекурсивних функцій в n-кортеж і потім транслювати посилання на fi в індекс i-го елемента такого кортежу.

Для реалізації цієї ідеї необхідно використовувати константні функції TUPLE-n та INDEX:

TUPLE-n $\mathbf{E_1}$ $\mathbf{E_2}$ ... $\mathbf{E_n}$ будує кортеж, що складається з виразів E_1, E_2, ..., E_n, який іноді записують в вигляді **< $\mathbf{E_1}$, $\mathbf{E_2}$, ..., $\mathbf{E_n}$ >**.

INDEX k <$\mathbf{E_1}$, $\mathbf{E_2}$, ..., $\mathbf{E_n}$> = $\mathbf{E_k}$ ($1 < k < n$).

Функції f_1, f_2, ..., f_n можна переписати в вигляді λ-виразів L_1, ..., L_n, залишаючи кожне входження будь-якої з функцій f_1 в тілах функцій f_1, ..., f_n

без змін. Після цього можна з'єднати ці вирази в n-кортеж T, використовуючи функцію TUPLE-n:

$$\mathbf{T = TUPLE\text{-}n\ L_1\ L_2\ ...\ L_n}$$

Елементи T можна модифікувати, замінюючи всі посилання на функцію f_i в L_i ($1 \le i\ , j \le n$) на виклики функції INDEX наступним чином:

$$\mathbf{f_i \text{ -> } INDEX\ i\ T}$$

Модифіковані вирази L_i будемо позначати L_i'. Тоді T має вигляд

$$\mathbf{T = TUPLE\text{-}n\ L_1'\ L_2'\ ...\ L_n'}$$

Це визначення рекурсивне, оскільки кожне L_i' може саме містити посилання на T. Для завершення визначення, потрібно до виразу T застосувати Y-комбінатор:

$$\mathbf{T = Y\ (\ \lambda T.TUPLE\text{-}n\ L_1'\ L_2'\ ...\ L_n'\ )}$$

Даний вираз є поданням набору функцій f_1 ... f_n.

Розглянемо приклад. Нехай є дві взаємно рекурсивні функції :

```
f :: Int-> Int
g :: Int -> Int
f n = if n == 0 then 0 else g n
g n = n + f (n-1)
```

Вони можуть бути представлені наступним λ-виразом:

Y (λT.TUPLE-2 (λn.COND (= n 0) 0 ((INDEX 2 T) n))
(λn. + N ((INDEX 1 T) (– n 1))))

Визначення Y

Розглянемо визначення Y-комбінатора в термінах λ-виразу. Вираз, що представляє Y, має вигляд :

λh.(λx.h(x x)) (λx. H(x x))

Для того, щоб переконатися в цьому, розглянемо вираз :

Yf = (λh.(λx.h(x x)) (λx.h(x x))) f
-> (λx.f(x x)) (λx.f(x x))

$$\textbf{-> f((λx.f(x x)) (λx.f(x x)))}$$

$$\textbf{= f(Yf)}$$

3.4 Чисте λ-числення

У чистому λ-численні відсутня поняття типізованих констант. Іншими словами, в чистому λ-численні відсутня δ-редукція.

Булеві константи і булеві операції

Розглянемо вираз:

`if P then Q else R` (або `COND PQR`)

Тут умовний оператор (COND) – вбудована функція, яка вибирає *Q* або *R* в залежності від значення предиката *P*. Іншими словами, предикат умовного оператора можна розглядати як функцію, яка сама вибирає один з виразів *Q* або *R*. Вираз *COND PQR* в цьому випадку транслюється в *PQR* з правилами перетворення

TRUE x y -> x

FALSE x y -> y

Цей підхід дає основу для подання булевих констант і операцій засобами чистого λ -числення.

Умовний оператор COND може бути представлений у вигляді:

COND = λp. λq. λr. pqr

TRUE і FALSE представляються у вигляді:

TRUE = λx. λy. x

FALSE = λx. λy. y

Розглянемо приклад :

COND TRUE A B

= (λp. λq. λr. p q r) (λx. λy.x) A B

-> (λq. λr. (λx. λy. x) q r) A B

-> (λr. (λx. λy.x) A r) B

-> (λx. λy.x) A B

-> (λy.A) B

-> A

Функції AND і OR мають вигляд :

AND = λx. λy. x y FALSE

OR = λx. λy. (x TRUE) y

Вірність такого представлення може бути доведена отриманням відповідних таблиць істинності. Наприклад:

TRUE AND FALSE = and true false

= (λx. λy. x y (λx. λy.y)) (λx. λy.x) (λx. λy.y)

-> (λy. (λx. λy.x) y (λx. λy.y)) (λx. λy.y)

-> (λx. λy.x) (λx. λy.y) (λx. λy.y)

-> (λy. (λx. λy.y)) (λx. λy.y)

-> (λx. λy.y) = FALSE

Списки в чистому λ-численні

Зазвичай, списки будуються за допомогою двох функцій-конструкторів, одна з яких позначає порожній список (NIL), а інша будує новий список з елементами старого списку (CONS). Конструктор CONS в λ-численні має вигляд:

CONS = λh. λt. λs.s h t

Тут змінні h, t, s відповідають голові списку, хвосту списку і функції-селектору.

Селектори – це такі функції, які повертають або перший, або другий аргумент функції-конструктора. Тому вони можуть бути представлені у вигляді наступних λ-виразів :

λh. λt.h і **λh.λt.t**

що в точності відповідає визначенням TRUE і FALSE .

Тоді функції для виділення головного і хвостового елементів списку L (тобто HD і TL відповідно) можуть бути визначені наступним чином:

HD = λL.L TRUE

TL = λL.L FALSE

Розглянемо приклад :

HD (CONS a b) = (λs.s TRUE) ((λh. λt. λs.s h t) a b)

-> ((λh. λt. λs.s h t) a b) TRUE

-> (λs.s a b) TRUE

-> TRUE ab = (λh. λt. h) a b

-> (λt. a) b

-> a

Функція IE, яка повертає TRUE у разі якщо список порожній, має вигляд:

IE = λs.s (λh. λt. FALSE)

Порожній список NIL на мові λ-числення має вигляд:

NIL = λx. TRUE

Розглянемо приклад :

IE (TL (CONS 1 NIL))

= (λs.s (λh. λt. FALSE)) (TL (CONS 1 NIL))

-> (TL (CONS 1 NIL)) (λh. λt. FALSE)

= ((λs.s FALSE) (CONS 1 NIL)) (λh. λt. FALSE)

-> ((CONS 1 NIL) FALSE) (λh. λt. FALSE)

-> (((λh. λt. λs.s h t) 1 NIL) FALSE) (λh. λt. FALSE)

-> (((λt. λs.s 1 t) NIL) FALSE) (λh. λt . FALSE)

-> ((λs.s 1 NIL) FALSE) (λh. λt . FALSE)

-> (FALSE 1 NIL) (λh. λt. FALSE)

= ((λx. λy. y) 1 NlL) (λh. λt. FALSE)

-> ((λy.y) NIL) (λh. λt. F ALSE)

-> NIL (λh. λt. FALSE)

-> (λx. TRUE) (λh. λt. FALSE)

-> TRUE

Натуральні числа в чистому λ-численні

Натуральне число n можна розглядати як n-елементний список об'єктів, що мають довільні значення. Виходячи з цього, можна визначити натуральні числа.

Число 0 в цьому випадку відповідає порожньому списку (NIL). Функція SUCC повертає наступний за n-тим елемент натурального ряду. Отже, вона додає одиницю до даного числа n та повинна тільки розширити на один елемент список з n елементів. Тоді SUCC буде мати вигляд:

SUCC = λn.CONS "any" n

де "any" – це будь-який λ-вираз.

λ-числення де Брейна

В λ-численні цілком можливо записати два вирази, які є семантично ідентичними, але синтаксично відрізняються один від одного через різні імена змінних.

Основна ідея канонічного λ-числення де Брейна полягає в тому, щоб видалити з λ-виразу всі імена змінних, замінивши їх на цілі числа. λ-вираз в такому випадку складається з цілих чисел, що представляють порядкові номери символів λ, що пов'язують змінні, замінені цими цілими числами, і з самих символів λ. При цьому семантично еквівалентні λ–виразу та є також синтаксично еквівалентними.

Ціле число, яке заміняє змінну, можна розглядати як «зсув» або «глибину вкладеності» символу λ, що вводить цю змінну, після якого в цьому випадку немає ніякої необхідності вказувати ім'я змінної.

В якості прикладу наведемо представлення в λ-численні де Брейна функції (SUCC):

$$\lambda. + L0\ 1$$

Більш складний λ-вираз

$$\lambda x.\ \lambda y.\ \lambda f.\ f(\ \lambda x'.x\ )\ (\ +\ x\ y\ )$$

має наступне канонічне представлення:

$$\lambda.\ \lambda.\ \lambda.\ L2\ (\ \lambda.L0)\ (+\ L0\ L1)$$

Оскільки в канонічному λ-численні немає імен змінних, там немає еквівалента правилам α-перетворення виразів.

3.5 Типізоване λ-числення

Система виведення типів і перевірка типів

Наступний розгляд виведення та перевірки типів можна вважати неформальним вступом до типізованого лямбда-числення.

Перевірка типів важлива з двох причин.

По-перше, вона дозволяє виявити багато логічних помилок програми ще на етапі компіляції. Більшість з них є результатом невідповідності типів виразів, так що запускати програму для виявлення цих помилок просто немає необхідності.

По-друге, програма на етапі виконання працює більш ефективно, так як будь-яке згадування про типи даних може повністю бути відсутнім в коді, що створюється компілятором.

Неформальна перевірка типів

Тип – це або змінна типу або застосування оператора типу до певного числа аргументів типу.

Базові типи, такі як **int** (цілий тип) і **bool** (логічний тип), є операторами типу без аргументів, а **[]** (для зручності будемо писати **list**) – це оператор типу з одним аргументом. Також будемо використовувати бінарні інфіксні оператори -> (оператор типу функції) та **X** (оператор типу декартового добутку).

Типи, що містять принаймні одну змінну, є поліморфними **(політипами),** а типи, які не містять змінних типу, є мономорфними **(монотипи)**.

Таким чином, базові типи є монотипами, так само як і типи багатьох примітивних функцій, наприклад **succ :: int -> int** та **is–zero :: int -> bool**. Однак не всі примітивні функції є мономорфними, наприклад hd [a] -> a є поліморфною, оскільки її визначення типу містить змінну типу a.

Можна визначати довільно складні вирази типу, наприклад вираз типу для функції **map** може бути записано таким чином:

(a -> b) -> [a] -> [b]

або

(a -> b) X list a-> list b

При перевірці типу виразу необхідно знайти найзагальніший тип, який він може мати. Тобто типи які одержані при присвоєнні змінним типу конкретних значень, є частинними випадками цього найбільш загального типу.

Іншими словами, необхідно сконструювати систему виведення типів, яка б змогла визначити найзагальніший тип, або прийти до протиріччя, яке означає, що цей вираз не має найбільш загального типу.

Основну ідею виведення типу можна зрозуміти, розглядаючи застосування функції f до виразу E.

Якщо відомо, що E має тип σ, то можна зробити висновок, що тип f – це σ -> ρ, де ρ – тип, який повинен бути отриманий.

Якщо включити в розгляд кваліфіковані вирази, отримаємо більш складний випадок, оскільки тип присвоюється ідентифікатору двічі (в кваліфікаторі та в результаті), і тому необхідно ототожнити такі типи, щоб отримати найзагальніший.

Розглянемо приклад . Нехай є вираз вигляду:

```
let f x = x in f 3
```

Щоб мало місце узгодження між типами його підвиразів, потрібно обчислити рівняння:

$\sigma 3$ = Int (базовий тип) (1)

σf = Int -> $\rho 1$ (f 3 – це застосування функції) (2)

σx = $\rho 2$ (тип x невідомий) (3)

σf = $\rho 2$-> $\rho 3$ (f x – це застосування функції) (4)

$\rho 3$ = $\rho 2$ (дві частини рівності повинні мати однаковий тип) (5)

З цих рівнянь можна бачити, що

$\rho 2$ = Int , $\rho 1$ = $\rho 3$ (з рівнянь (2) і (4)),

$\rho 3$ = $\rho 2$ (рівняння (5)), що в результаті дає

$\rho 1$ = $\rho 2$ = $\rho 3$ = Int , і типом виразу f3 є

$\rho 1$ = Int , як і очікувалося .

Той факт , що система рівнянь має розв'язок, говорить про коректність виразу E з точки зору його типу.

Аналогічні міркування справедливі для поліморфних виразів. Розглянемо в якості прикладу функцію map:

```
map f [] = []
map f (x:xs) = f x: map f xs
```

Оскільки примітивні функції обробки списків є поліморфними, призначимо нові невідомі типи η_1, η_2, ... змінним типу цих функцій. Тоді маємо наступні рівняння:

σ ([ ]) = list η_1

σ (:) = (η_3 X list η_3) -> list η_3

Рівняння що гарантують відповідність типів кожного підвиразу в визначенні map мають вигляд:

σ(map) = σf X σm -> ρ_1 Для деякої невідомої змінної типу ρ_1 (1)

σf = ρ_2-> ρ_3 (2)

σ(map) = σf X ρ_2-> ρ_3 (3)

$\boldsymbol{\sigma(:) = \rho_3 \ X \ \rho_4 \text{ -> } \rho_5}$ (4)

Необхідне також рівняння, яке потребує, щоб обидві частини рівності у визначенні map мали однаковий тип:

$\boldsymbol{\rho_1 = \sigma([\ ]) = \rho_5}$ (5)

Розв'язок даної системи рівнянь буде мати вигляд:

$\boldsymbol{\sigma m = \rho_2}$

$\boldsymbol{\rho_1 = \rho_3}$

$\boldsymbol{\rho_3 = \text{list } \eta_3}$

$\boldsymbol{\rho_4 = \text{list } \eta_3}$

$\boldsymbol{\rho_5 = \text{list } \eta_3}$

Звідси отримуємо:

$\boldsymbol{\sigma m = \rho_2}$

$\boldsymbol{\rho_1 = \rho_3 = \text{list } \eta_3}$

$\boldsymbol{\rho_5 = \rho_4 = \text{list } \eta_3}$

Типізоване лямбда-числення

Типи є зручним засобом визначення різних різновидів даних, таких як логічні і цілочисельні значення або функції. У типізованому лямбда-численні лямбда-терму може бути присвоєно тип.

Для чого необхідне типізоване лямбда-числення:

1. Подолання парадоксу Рассела (парадокс цирульника). Джерелом протиріч служить незвичайна циклічна природа прийому, що використовується, – застосування функції до самої себе.
2. Застосування типізації в мовах програмування. Одна з причин введення типів в мовах програмування – це підвищення ефективності породжуваного компілятором коду.
3. Обмежена статична перевірка коректності програм.

Розглянемо введення типу в лямбда-численні. Якщо M – терм і тип α присвоєно M, то пишуть:

$$\mathbf{M : \alpha}$$

Наприклад, для тотожної функції можна написати I : α -> α. Тобто тотожній функції присвоєно тип α -> α. Якщо тип аргументу X тотожної функції α, то її значення теж має тип α. У загальному випадку α -> β є типом функції з α в β.

Існує дві системи типізації:

- по Чорчу,
- по Каррі.

В типізації по Чорчу явно вказується тип. Кожен терм отримує єдиний тип, тобто в ході побудови термів для кожного з розглянутих раніше нетипізованих термів вказується тип. Типи констант є визначеними, але типи змінних можуть бути довільними.

Точні правила побудови типізованих термів мають вигляд:

$$\frac{}{\nu : \sigma}$$

$$\frac{\text{Константа c має тип } \sigma}{c : \sigma}$$

Також

$$\frac{s : \sigma \to \tau \quad t : \sigma}{s\,t : \tau}$$

$$\frac{\nu : \sigma \qquad t : \tau}{\lambda \nu . t : \sigma \to \tau}$$

В типізації Каррі структура термів відповідає нетипізованому випадку, при цьому терм може як мати тип (причому не один), так і не мати його. Наприклад, тотожній функції λx.x може бути цілком обґрунтовано присвоєно довільний тип вигляду σ -> σ.

В мові Haskell існують два взаємопов'язаних аргументи на користь даного підходу до типізації: по-перше, він дозволяє зручніше висловити властивий Haskell поліморфізм; по-друге, він добре узгоджується з практикою програмування на цій мові, коли не завжди потрібно ставити типи явно.

В той же час, деякі формальні аспекти призначення типів по Каррі виявляються досить складними.

Відношення типізації не може бути задане відриваючись від певного контексту, який є скінченною множиною тверджень щодо типів змінних. Нехай

$$\Gamma \vdash t : \sigma$$

Тобто, твердженню в контексті Γ терма t може бути призначений тип σ. (Якщо це твердження справедливе при порожньому контексті, то вираз скорочується до $t: \sigma$.) Елементи множини Γ мають вигляд $\nu : \sigma$, отже вони самі по собі є твердженнями щодо типів окремих змінних, зазвичай тих, які входять в терм t.

Правила типізації в системі Каррі мають вигляд:

$$\frac{\nu:\sigma \in \Gamma}{\Gamma \vdash \nu:\sigma}$$

$$\frac{\text{Константа c має тип } \sigma}{c:\sigma}$$

$$\frac{\Gamma \vdash s:\sigma \to \Gamma \vdash t:\sigma}{\Gamma\ t\ s:\ \tau}$$

$$\frac{\Gamma \cup \{\nu:\sigma\} \vdash t:\tau}{\Gamma \vdash \lambda\nu.t : \sigma \to \tau}$$

Як приклад, розглянемо процедуру виведення типу тотожної функції. Згідно правилу типізації змінних, маємо:

$$\{ \nu : \sigma \} \vdash x : \sigma,$$

звідки, застосувавши останє правило, отримуємо

$$\vdash \lambda x.x : \sigma \text{ -> } \sigma$$

Цей вираз можна скоротити до

$$\lambda x.\ x{:}\ \sigma \text{ -> } \sigma$$

Звідси бачимо важливість контекстів в типізації по Каррі, так само, як і їх необов'язковість в типізації по Чорчу. Опускаючи контекст, можна вивести x: τ для довільного типу τ, після чого, відповідно до останнього правила, отримуємо λx.x: σ -> τ, звідки видно відмінність з інтуїтивним трактуванням тотожної функції.

Ця проблема не виникає в процесі типізації по Чорчу, оскільки в такому випадку або обидві змінні мають тип *σ*, звідки отримуємо λx.x: σ -> σ, або ці змінні є різними (оскільки різні їх типи, які вважаються частиною терма). В останньому випадку тип виразу буде дорівнювати λx: σ.(x: τ): σ -> τ.

Збереження типу

Схожість структури термів типізованого і нетипізованого лямбда-числення породжує природне бажання застосувати в типізованому випадку апарат формальних перетворень, розроблений для нетипізованого лямбда-числення. Однак, в такому випадку буде потрібно довести, що тип виразу в ході перетворень не змінюється (ця властивість називається збереженням типу).

Розглянемо лему, яка говорить про те, що додавання нових елементів в контекст не впливає на типізацію.

Лема 1. Якщо $\Gamma \vdash t : \sigma$ і $\Gamma \subseteq \Delta$, то справедливо $\Delta \vdash t{:}\ \sigma$.

Наступна лема говорить про те, що елементи контексту, що представляють змінні, які не є вільними в заданому термі, можуть ігноруватися.

Лема 2. Якщо $\Gamma \vdash t: \sigma$, то справедливо також $\Gamma_t \vdash t: \sigma$, де Γ_t містить виключно вільні змінні терма t

$$(\Gamma_t = \{x: \alpha \mid x: \alpha \in \Gamma \text{ та } x \in FV(t)\}).$$

І нарешті, теорема про збереження типу (в даному випадку для η-редукції; аналогічну теорему можна довести і для β-редукції).

Теорема. Якщо $\Gamma \vdash t: \sigma$ та $t \;_{\eta}\!\!-\!\!> t'$, то з цього випливає, що $\Gamma \vdash t' : \sigma$.

Зібравши разом результати аналогічних доведень для інших перетворень, отримуємо, що якщо $\Gamma \vdash t: \sigma$ та $t -> t'$, то виконується також $\Gamma \vdash t' : \sigma$.

Важливість цього висновку в тому, що якщо би правила обчислень, що застосовуються в ході виконання програми, могли змінювати типи виразів, це суперечило б основам статичної типізації.

Поліморфізм.

Типізація по Каррі надає різновид поліморфізму, що дозволяє призначити заданому терму різні типи. Тут слід розрізняти схожі поняття поліморфізму і перевантаження.

Обидва вони мають на увазі, що вираз може мати багато типів. Однак, в разі поліморфізму, всі ці типи структурно пов'язані один з одним, що допускає будь-які з них, що задовольняють заданому зразку. Наприклад, тотожній функції можна призначити тип $\sigma -> \sigma$, або $\tau -> \tau$, або навіть $(\sigma -> \tau) \rightarrow (\sigma -> \tau)$, але всі вони мають однакову структуру.

З іншого боку, суть перевантаження в тому, що задана функція може мати різні типи, структура яких може різнитися, або ж допустима лише обмежена множина типів. Наприклад, функції «+» може бути дозволено мати тип **int -> int -> int** або **float -> float -> float**, але не **bool -> bool -> bool**.

Ще одним близьким поняттям є підтипи, які є більш строгою формою перевантаження. Введення підтипів дозволяє трактувати деякий тип як підмножину іншого.

Проблеми let-поліморфізму

Розглянута вище система типів накладає деякі небажані обмеження на поліморфізм. Зокрема, наступний вираз є абсолютно коректним :

if (λx. x) true then (λx. x) 1 else 0

Доведемо, що це вираз може бути типізованим. Припустимо, що константам можна призначити типи в порожньому контексті, і що можна двічі застосувати правило типізації комбінацій термів для призначення типу if (зауважуючи, що вираз вигляду **if b then t1 else t2** є лише скороченням запису для **COND b t1 t2**).

$$\dfrac{\dfrac{\dfrac{\{\ x\text{: bool}\} \vdash x\text{: bool}}{\vdash (\ \lambda x.\ x)\text{: bool - > bool}} \quad \dfrac{}{\vdash \text{true: bool}}}{\vdash (\ \lambda x.\ x)\text{: true - > bool}} \qquad \dfrac{\dfrac{\{\ x\text{: int }\} \vdash x\text{: int}}{\vdash (\ \lambda x.\ x)\text{: int - > int}} \quad \dfrac{}{\vdash 1\text{: int}}}{\vdash (\ \lambda x.\ x)\text{: } 1 > \text{int}} \quad \vdash 0\text{: int}}{\vdash \ \text{if } (\ \lambda x.\ x)\text{: true then } (\ \lambda x.\ x)\ 1 \text{ else } 0\text{: int}}$$

Два екземпляри тотожної функції отримують типи **bool -> bool** та **int -> int** відповідно.

Розглянемо інший вираз:

let I = λx. x in if I true then I 1 else 0

За означенням, це лише зручніший спосіб запису для

(λI. if I true then I 1 else 0) (λx. x)

Отже, тип цього виразу не може бути виведений в межах описаних вище правил. Тобто, є єдиний екземпляр тотожної функції, якому необхідно призначити єдиний тип.

Подібне обмеження на практиці неприйнятно, оскільки функціональне програмування передбачає часте використання let. Якщо правила типізації не будуть змінені, то втрачається багато можливостей поліморфізму. Тому рішенням буде відмова від трактування конструкції let як скороченого запису на користь реалізації її як примітиву мови, після чого до множини правил типізації слід додати нове правило:

$$\frac{\Gamma \vdash s: \sigma \quad \Gamma \vdash t\,[s / x]: \tau}{\Gamma \vdash \text{let } x = s \text{ in } t: \tau}$$

Це правило, яким вводиться поняття let-поліморфізму, демонструє, що з точки зору типізації let-зв'язані змінні трактуються як прості підстановки відповідних виразів замість їх імен.

Додаткова умова $\Gamma \vdash s: \sigma$ потрібна виключно для того, щоб гарантувати існування коректного типу виразу *s*, причому точне значення цього типу нас не цікавить.

Мета даного обмеження в тому, щоб уникнути помилкових висновків про існування коректних типів для таких термів, як

let x = λ f. F f in 0

Тепер можна вивести тип проблемного let-виразу, користуючись наведеними вище правилами :

$$\cfrac{\cfrac{\cfrac{}{\{ x: \sigma \} \vdash x: \sigma} \qquad \ldots\ldots\ldots\ldots\ldots\ldots}{\vdash (\lambda x.\ x): \sigma -> \sigma \qquad \vdash \text{ if } (\lambda x.\ x): \text{true then } (\lambda x.\ x)\ 1 \text{ else } 0: \text{int}}}{\vdash \text{let } I = \lambda x.\ x \text{ in if } I \text{ true then } I\ 1 \text{ else } 0}$$

Найзагальніший тип

Як було зазначалося, тип деяких виразів (наприклад, **λf. F f** або **λf. (f true, f 1)**, вивести неможливо. Типізовані вирази зазвичай мають множину типів, хоча деякі з них, наприклад, true має лише один.

Різновид поліморфізму, доступний в мові Haskell, називається параметричним, отже будь-які типи виразів повинні мати структурну подібність. Більше того, для кожного типізованого виразу існує так званий найзагальніший (головний) тип, причому всі можливі типи даного виразу представляють собою екземпляри найзагальнішого типу.

Розглянемо розширення типу, доповнене поняттям змінної типу. Це означає, що типи можуть бути побудовані шляхом застосування конструкторів типів як до типів-констант, так і до змінних.

Будемо використовувати грецькі літери α і β для позначення змінних типу та σ і τ для довільних типів.

За допомогою цієї розширеної нотації можна визначити поняття підстановки типу в інший тип замість змінної типу. Така підстановка в точності відповідає підстановці термів, тому може використовувати ті ж самі позначення (наприклад, $(\sigma \text{ -> } \mathbf{bool})\, [(\sigma \text{ -> } \tau) / \sigma] = (\sigma \text{ -> } \tau) \text{ -> } \mathbf{bool}$).

Однак, формальне визначення підстановки типів простіше, ніж для термів, оскільки не вимагає враховувати зв'язування змінних. Тоді маємо

$$\alpha_i\, [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_k] = \tau_i \ \ \mathbf{if}\ \ \alpha_i \neq \beta \ \ \mathbf{for\ 1 \leq i \leq k}$$

$$\beta\, [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_k] = \beta \ \ \mathbf{if}\ \ \alpha_i \neq \beta \ \ \mathbf{for\ 1 \leq i \leq k}$$

$$(\sigma_1, \ldots, \sigma_k)\ \mathbf{con}[\theta] = (\sigma_1[\theta], \ldots, \sigma_k[\theta])\ \mathbf{con}$$

Типи констант трактуються як нуль-арні конструктори типів, тобто () int еквівалентний int. Враховуючи наведене визначення підстановки, можна ввести відношення, що тип σ є загальнішим за тип σ', позначивши його через $\sigma \preceq \sigma'$.

Пара типів утворює таке відношення тоді і тільки тоді, коли знайдеться множина підстановок θ таке що $\sigma' = \sigma\,\theta$. Наприклад:

$$\alpha \preceq \sigma$$

$$\alpha \text{ -> } \alpha \preceq \beta \text{ -> } \beta$$

$$\alpha \text{ -> } \mathbf{bool} \preceq (\beta \text{ -> } \beta) \text{ -> } \mathbf{bool}$$

$$\beta \text{ -> } \alpha \preceq \alpha \text{ -> } \beta$$

$$\alpha \to \alpha \preceq (\beta \to \beta) \to \beta$$

З урахуванням викладеного, можна сформулювати теорему:

Теорема. Кожен типізований терм має головний тип, тобто для довільного $t: \tau$ знайдеться тип σ такий, що $t: \sigma$, і для будь-якого типу σ' з $t: \sigma'$ випливає $\sigma \preceq \sigma'$.

Відношення $\preceq$ є відношенням квазіпорядку, що є рефлексивним і транзитивним. Головний тип не унікальний, але при цьому всі його можливі форми рівні з точністю до перейменування змінних типів, що входять в них.

У більш точному формулюванні, якщо σ і τ є одночасно головними типами виразу, то справедливо $\sigma \sim \tau$, тобто є $\sigma \preceq \tau$ та $\tau \preceq \sigma$ одночасно.

Процедура доведення цієї теореми представляє собою конкретний алгоритм пошуку такого типу. Цей алгоритм відомий як алгоритм Мілнера (чи алгоритм Хіндлі-Мілнера).

Сильна нормалізація

Розглянемо терми, що не мають нормальної форми, такі як

$$((\lambda x.\ xxx)\ (\lambda x.\ xxx)) \to$$
$$((\lambda x.\ xxx)\ (\lambda x.\ xxx)\ (\lambda x.\ xxx))$$
$$\to (\cdot\cdot\cdot)$$

В межах типізованого лямбда-числення подібна ситуація неможлива на підставі теореми про сильну нормалізацію:

Теорема (про сильну нормалізацію). Будь-який типізований терм має нормальну форму, а будь-яка можлива послідовність редукції, яка починається з типізованого терма, завершується.

Маємо очевидну перевагу: функціональна програма, яка задовільняє дані типи, може обчислюватися в довільному порядку, при цьому процес редукції завжди скінченний і призводить до єдиної нормальної форми. (Єдність випливає з теореми Чорча-Россера, яка залишається справедливою і для випадку типізованого лямбда-числення.)

Можливість реалізації функцій, що не завершуються, необхідна для забезпечення повноти за Тьюрінгом, інакше неможливо визначити довільні обчислювальні функції, і більше того, навіть не всю множину всюди визначених функцій.

Можна було б знехтувати цим обмеженням, якщо воно дозволяло використовувати всі функції, що мають практичний інтерес. Однак, це не справедливо, оскільки клас функцій, які можна представити в межах типізованого лямбда-числення, виявиться вузьким.

Оскільки всі визначимі функції є всюди визначеними, то неможливо давати довільні рекурсивні визначення.

Виявляється, звичайні комбінатори нерухомої точки не можуть бути типізовані. Очевидно, що тип $\mathbf{Y = \lambda f. (\lambda x. f (xx)) (\lambda x. f (xx))}$ не існує, оскільки x застосовується до самої себе, бо пов'язана лямбда-абстракцією.

Для відновлення повноти за Тьюрінгом можна ввести альтернативний спосіб задання довільних рекурсивних функцій, що не порушує типізацію.

Визначимо поліморфний оператор рекурсії, всі можливі типи якого мають вигляд

$$\mathrm{Rec}: ((\sigma \to \tau) \to (\sigma \to \tau)) \to \sigma \to \tau$$

і додаткове правило редукції, згідно з яким для довільної функції

$$F: (\sigma \to \tau) \to (\sigma \to \tau)$$

маємо

$$\mathrm{Rec}\ F \to F\ (\mathrm{Rec}\ F)$$

Тоді можна припустити, що рекурсивні визначення вигляду let відображаються на ці оператори рекурсії.

3.6 Елементи теорії категорій

Визначення 1. Категорія складається з:

- сукупності *об'єктів*, при цьому якщо X є об'єктом C, то це записується як $X \in C$;

- множин *морфізмів* з X в Y для кожної пари об'єктів X та Y. Такі множини позначаються як homset.

Таким чином, категорію можна уявити як сукупність множин (які, можливо, мають додаткову структуру) та відображень (що зберігають цю структуру).

Для цих компонентів категорії виконуються такі правила.

- Для довільного об'єкта X існує *тотожний морфізм*.
- Можна робити *композицію морфізмів*, яка також іноді записується як $g \circ f$.
- Тотожний морфізм є одночасно лівою та правою одиницею відносно операції композиції, тобто $f \circ 1_x = f = 1_y \circ f$.
- Операція композиції асоціативна, тобто $(hg)\,f = h\,(gf)$ для довільних морфізмів f, g та h.

Визначення 2. Морфізм називається *ізоморфізмом*, якщо у нього є обернений морфізм, тобто існує морфізм такий, що

$$g \circ f = 1_x \ \text{ та } \ f \circ g = 1_y.$$

Категорія є найпростішою конструкцією, що дозволяє описувати системи (об'єкти) та процеси (морфізми).

Якщо для двох морфізмів $f : X \to Y$ та $g : Y \to X$ виконується рівність $gf = 1_x$, то морфізм g є *лівим оберненим* до f, а f – *правим оберненим* до g.

Двосторонньо оберненим (або просто оберененим) до морфізму f називається морфізм, який є лівим і правим до f.

Лема. Якщо морфізм має правий і лівий обернений, всі вони співпадають, при цьому f – *ізоморфізм*.

Приклади категорій.

1. Категорія множин та функцій. (Класом об'єктів цієї категорії є клас усіх множин, і для двох множин X та Y множина hom(X,Y) є множиною всіх функцій з X в Y).
2. Категорія топологічних просторів та неперервних відображень.

3. Категорія груп та гомоморфізмів.
4. Категорія нормованих кілець та неперервних груп.
5. Категорія скінченновимірних гільбертових просторів і лінійних операторів на них.

Визначення 3. Функтор із категорії C в категорію D є відображенням, яке:

- відображає кожен об'єкт $X \in C$ в деякий об'єкт $F(X) \in D$;
- відображає кожен морфізм із категорії C в морфізм в категорії D.

Функтор має задовольняти наступні умови:

- для всіх об'єктів $X \in C$ функтор зберігає тотожні морфізми: $F(1_x) = 1_{F(x)}$;
- для довільної пари морфізмів з категорії C функтор зберігає композицію: $F(gf) = F(g)\, F(f)$.

Функтор можна також трактувати як «представлення» категорії C в D. Сенс полягає в тому, що функтор F може відображати об'єкти та морфізми деякої «абстрактної» категорії C в об'єкти та морфізми деякої «конкретнішої» категорії D.

Приклад. Розглянемо абстрактну групу G. Це те саме, що й категорія, яка складається з одного об'єкта, всі морфізми якої оборотні. В нашому випадку об'єкт нас не цікавить, тому його можна просто позначити ·, але морфізми – це елементи G, їх композиція здійснюється за допомогою операції множення в групі.

Тоді *подання* групи G на скінченний гільбертів простір – це те ж саме, що і функтор. Таким же чином, *дія* групи G на множину – це те саме, що й функтор. В обох випадках абстрактна група стає більш конкретною.

Інший приклад функтора:

Для будь-якої множини A можна визначити наступний функтор: (- xA): **Set -> Set** – він відобразить будь-яку множину X в декартів добуток $X \times A$. (Тут **Set** – категорія всіх множин.)

Визначення 4. Нехай F і G – функтори з категорії C в D. Тоді **природне перетворення** зіставляє кожному об'єкту X категорії C морфізм

η_X F(X) -> G(X) в категорії D, що називається **компонентом** η в X, так що для будь-якого морфізму f: X -> Y зображена нижче діаграма комутативна.

$$\begin{tikzcd} F(X) \arrow[r, "F(f)"] \arrow[d, "\eta_X"'] & F(Y) \arrow[d, "\eta_Y"] \\ G(X) \arrow[r, "G(f)"'] & G(Y) \end{tikzcd}$$

Визначення 5. **Природним ізоморфізмом** між функторами називається таке природне перетворення, що η_X є ізоморфізмом для кожного об'єкта $X \in C$.

Приклад. Розглянемо категорію H – групу, тобто категорію з одним об'єктом, наприклад $\cdot$. Тоді функтори F та G будуть представниками групи H в гільбертових просторах $F(\cdot)$ та $G(\cdot)$. Природним перетворенням в цьому випадку є **переплітаючий оператор** з одного подання в інше – лінійний оператор, що задовольняє умову $AF(h) = G(h)A$ для всіх елементів групи H.

Приклади природних перетворень:

1. Нехай є дві множини A та B і функція f: A **->** B. Ця функція визначає природне перетворення між функторами (- xA), (- xB): **Set ->** **Set**.
   (- xf): (- xA) **->** (- xB) за такою формулою: (x, a) **->** (x, f(a)).
2. Для кожного A в **Set** існує природне перетворення 1 **->** P_A. Функтор P_A: **Set ->** **Set** відображає довільну множину X в X^A – множину функцій з A в X. Для будь-якого X в **Set** функція з X в X^A буде відображати кожен елемент x з X в константну функцію з A в X.

Монада

Монада в категорії C – це функтор T: C -> C з двома природними перетвореннями: η: 1 -> T та μ: T ° T -> T. (**Ендофунктор** – функтор з категорії в саму себе.)

Тоді нехай T(η): T -> T ° T – перетворення, результатом якого є застосування T до η та T(μ): (T ° T) ° T -> T ° T.

Тоді маємо такі аксіоми для монад:

- $T(\eta) \circ \mu$ – це тотожність T -> T:

$$T(X) \xrightarrow{T(\eta_X)} T(T(X)) \xrightarrow{\mu_{T(X)}} T(X)$$

- $T(\mu) \circ \mu$ рівносильно $\mu \circ \mu$:

$$\begin{tikzcd} T(T(T(X))) \arrow[r, "T(\mu_X)"] \arrow[d, "\mu_{T(X)}"'] & T(T(X)) \arrow[d, "\mu_X"] \\ T(T(X)) \arrow[r, "\mu_X"'] & T(X) \end{tikzcd}$$

Приклади монад:

1. Для будь-якої категорії C може бути визначена тотожна монада, що складається з тотожного функтора та тотожного морфізму.
2. Розглянемо групу G. Визначимо монаду M_G в **Set**.

   Функтор монади: M_G : X -> X × G.

   u(X): X -> X × G відображає елемент x в пару (x, e), де e – одиниця групи.

   $M_G (M_G (X)) = (id_X, m_G)$, де m_G – множення груп.

Моноїдальні категорії

Визначення 6. Декартовим добутком C × C′ **категорій** C та C′ називається категорія, в якій:

- об'єктами є пари: X∈C та X′∈C′, тобто (X, X');
- морфізм (X_1, X'_1) -> (X_2, X'_2) є парою (f_1, f_2), де f_1: X_1 -> X'_1 і f_2: X_2 -> X'_2.

Визначення 7. Моноїдальна категорія складається з:

- біфунктора, що називається тензорний добуток або моноїдальний добуток,
- об'єкту I, що називається одиницею або тотожним об'єктом,
- трьох природних ізоморфізмів, що виражають той факт, що операція тензорного добутку

– асоціативна: існує природний ізоморфізм (так званий асоціатор)

$$\alpha_{A,B,C}: (A \otimes B) \otimes C \to A \otimes (B \otimes C),$$

– I є одиницею: існують два природні ізоморфізми λ та ρ:

$$\lambda_A: I \otimes A \to A,$$

$$\rho_A: A \otimes I \to A.$$

На ці природні ізоморфізми накладено додаткові умови:

- для всіх об'єктів A, B, C, D наступна п'ятикутна діаграма коммутативна:

$$\begin{tikzcd}
((A\otimes B)\otimes C)\otimes D \arrow[r, "\alpha_{A,B,C}\otimes D"] \arrow[d, "\alpha_{A\otimes B,C,D}"'] & (A\otimes(B\otimes C))\otimes D \arrow[r, "\alpha_{A,B\otimes C,D}"] & A\otimes((B\otimes C)\otimes D) \arrow[d, "A\otimes\alpha_{B,C,D}"] \\
(A\otimes B)\otimes(C\otimes D) \arrow[rr, "\alpha_{A,B,C\otimes D}"'] & & A\otimes(B\otimes(C\otimes D))
\end{tikzcd}$$

- для всіх A та B наступна трикутна діаграма коммутативна:

$$\begin{tikzcd}
(A\otimes I)\otimes B \arrow[rr, "\alpha_{A,I,B}"] \arrow[dr, "\rho_A\otimes B"'] & & A\otimes(I\otimes B) \arrow[dl, "A\otimes\lambda_B"] \\
& A\otimes B &
\end{tikzcd}$$

Приклад моноїдальної категорії: категорію **Set** можна перетворити на моноїдальну, якщо взяти за операцію $X \otimes Y$ об'єднання множин.

Декартів добуток об'єктів

Визначення 7. Для заданих об'єктів X і X' в деякій категорії вважається, що об'єкт $X \times X'$, асоційований з морфізмами $p \times p'$ ($p: X \times X' \to X$, а $p': X \times X' \to X'$ – проекції $X \times X'$ в множини X, X'), є **декартовим добутком**

(або просто добутком) **об'єктів** X та X′ якщо для кожного об'єкта Q і морфізмів f, f'

$$\begin{tikzcd} & Q \arrow[dl, "f"'] \arrow[dr, "f'"] & \\ X & & X' \end{tikzcd}$$

має місце єдиний морфізм g, разом з яким наступна діаграма є комутативною:

$$\begin{tikzcd} & Q \arrow[dl, "f"'] \arrow[d, "g"] \arrow[dr, "f'"] & \\ X & X\times X' \arrow[l, "p"] \arrow[r, "p'"'] & X' \end{tikzcd}$$

Замкнена категорія

Симетрична моноїдальна категорія називається **замкненою**, якщо для будь-якого її об'єкту B функтор, що задається тензорним множенням на B справа A -> A⊗ B, має правий спряжений. (Функтори F і G **спряжені**, якщо вони задовольняють співвідношення Hom (F(X), Y) = Hom (X, G(Y)). При цьому F називається лівим спряженим функтором, а G – правим.)

Категорія C називається **декартово замкненою**, якщо вона задовольняє три умови:

- C містить термінальний об'єкт (об’єкт T – термінальний, якщо для будь-якого об'єкта X в C існує єдиний морфізм X -> T);
- будь-які два об'єкти X, Y в C мають добуток X × Y;
- будь-які два об'єкти Y, Z в C мають експоненціал Z^Y.

Експоненціал – категорний аналог множини функцій в теорії множин. Більш точно експоненціал Z^Y можна визначити як універсальний морфізм з функтора (- xY) в Z. (Функтор (- xY) з C в C відображає об'єкт X в X×Y та морфізми φ у $\varphi\times id_Y$).

Інакше кажучи, експоненціал Z^Y об'єктів Z та Y – це об'єкт, який разом з морфізмом eval: $Z^Y \times Y$ -> Z (відображення оцінки) такий, що для будь-якого об'єкта X і морфізму g: $X \times Y$ -> Z існує єдиний морфізм λg: X -> Z^Y, для якого наступна діаграма комутативна:

$$
\begin{tikzcd}
X \arrow[d, dashed, "\lambda g"'] \\
Z^Y
\end{tikzcd}
\qquad
\begin{tikzcd}
X \times Y \arrow[d, dashed, "\lambda g \times id_Y"'] \arrow[dr, "g"] & \\
Z^Y \times Y \arrow[r, "eval"'] & Z
\end{tikzcd}
$$

В категорії множин Z^Y експоненціал – це множина всіх функцій з Y в Z. Для довільного відображення g: $(X \times Y)$ -> Z, відображення λg: X -> Z^Y має вигляд $\lambda g(x)(y) = g(x, y)$.

Рекомендована література

1. Harold Abelson and Gerald Jay Sussman with Julie Sussman. Structure and Interpretation of Computer Programs, 2nd ed, MIT Press, 1996, 657 p. https://web.mit.edu/6.001/6.037/sicp.pdf
2. Э. Хювёнен, Й. Сеппянен. Мир Лиспа. В 2-х т. Пер.с финск. – М: Мир, 1990.
3. Peter Seibel. Practical Common Lisp, Apress, 2005, 501 p. https://gigamonkeys.com/book/
4. Підручник по Haskell: https://anton-k.github.io/ru-haskell-book/book/home.html
5. Miran Lipovaca. Learn You a Haskell for Great Good!: A Beginner's Guide, No Starch Press, 2011, 400 p.
   Переклад українською: https://haskell.trygub.com/
6. Bryan O'Sullivan, Don Stewart, and John Goerzen. Real World Haskell, O'Reilly Media, 2009, 670 p. http://book.realworldhaskell.org/
7. Will Kurt. Get Programming with Haskell, Manning Publications, 2018, 616 p.
8. C. Hankin. An Introduction to Lambda Calculi for Computer Scientists, Kings College, 2004, 164 p.
9. Х. Барендрегт. Ламбда-исчисление. Его синтаксис и семантика. – М.: Мир, 1985. – 606 с.
10. Математическая логика в программировании: сборник статей (ред. Захарьящев М.В., Янова Ю.И.). – М.: Мир, 1991. – 408 с.