MULTIPARADIGM PROGRAMMING

Викладачі: Баклан Ігор, Очеретяний Олександр, Глушко Богдан

Учасники навчального процесу

Навчальні групи

ІП-01, ІП-02, ІТ-01, ІТ-02, ІТ-03, ІТ-04

Мультипарадигменне програмування

3 курс, весна 2022

- Доц. Баклан І.В.
- Email: iaa@ukr.net
- Web: baklaniv.at.ua

Лекція 1

Парадигми програмування

1 лютого 2022 року

Стислий довідник про парадигми програмування

Одночасне програмування – мають мовні конструкції для паралельності, вони можуть включати багатопотоковість, підтримку розподілених обчислень, передачу повідомлень, спільні ресурси (включаючи спільну пам’ять) або майбутні

Програмування акторів – одночасні обчислення з акторами, які приймають локальні рішення у відповідь на навколишнє середовище (здатні до егоїстичної або конкурентної поведінки)

Програмування обмежень – відносини між змінними виражаються як обмеження (або мережі обмежень), що направляють допустимі рішення (використовує задоволення обмежень або симплексний алгоритм)

Програмування потоку даних – примусове перерахунок формул при зміні значень даних (наприклад, електронні таблиці)

Декларативне програмування – описує те, що має виконувати обчислення, без вказівки детальних змін стану, наприклад, імперативне програмування (функціональне та логічне програмування є основними підгрупами декларативного програмування)

Розподілене програмування – підтримує кілька автономних комп’ютерів, які спілкуються через комп’ютерні мережі

Функціональне програмування – використовує оцінку математичних функцій і уникає даних про стан і зміну

Загальне програмування – використовує алгоритми, написані в термінах пізніших типів, які потрібно вказати, які потім створюються за потребою для конкретних типів, наданих як параметри

Імперативне програмування – явні оператори, які змінюють стан програми

Логічне програмування – використовує явну математичну логіку для програмування

Метапрограмування - написання програм, які пишуть або маніпулюють іншими програмами (або собою) як своїми даними, або які виконують частину роботи під час компіляції, яка в іншому випадку була б виконана під час виконання

Метапрограмування шаблонів – методи метапрограмування, в яких компілятор використовує шаблони для створення тимчасового вихідного коду, який компілятор об’єднує з рештою вихідного коду, а потім компілює

Рефлексивне програмування – методи метапрограмування, за допомогою яких програма змінює або розширює себе

Об'єктно-орієнтоване програмування - використовує структури даних, що складаються з полів даних і методів разом з їх взаємодією (об'єктами) для розробки програм

На основі класів – об’єктно-орієнтоване програмування, в якому успадкування досягається шляхом визначення класів об’єктів, а не самих об’єктів.

На основі прототипів – об’єктно-орієнтоване програмування, яке уникає класів і реалізує успадкування через клонування екземплярів

Конвеєрне програмування – проста зміна синтаксису, щоб додати синтаксис до викликів вкладених функцій до мови, спочатку розробленої без жодного

Програмування на основі правил – мережа практичних правил, які містять базу знань і можуть бути використані для експертних систем, а також для виведення та вирішення проблем

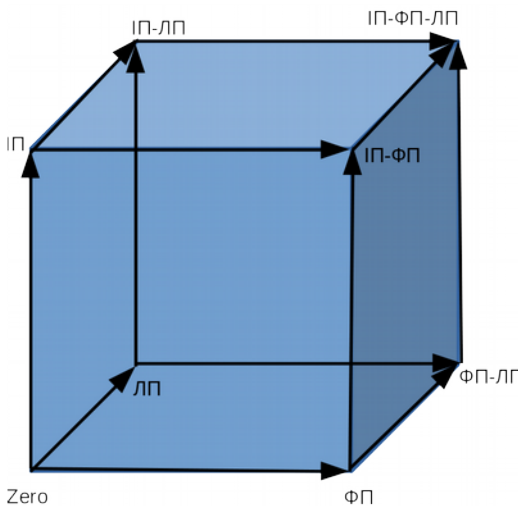
Візуальне програмування – маніпулювання елементами програми графічно, а не шляхом їх текстового вказівки (наприклад, Simulink); також називають діаграмним програмуванням.

Проблемно-орієнтоване програмування — орієнтоване на розв’язання задач з певної проблемної області. В своїй більшості несе мультипарадигменний характер.

Ймовірнісне програмування - спосіб опису статистичних моделей у вигляді декларативного програмного коду, з подальшим застосуванням алгоритмів аналізу та статистичного виведення. Споріднено до Байєсових мереж і прихованих марківських моделей. На відміну від класичного програмування, кінцевою метою якого є виконання програми, імовірнісне програмування наголошує на аналізі імовірнісних програм. Типовий сценарій - опис залежностей даних, що спостерігаються від прихованих параметрів у вигляді програми, після чого використовуються вбудовані алгоритми для зворотного ймовірнісного виведення значень на вході від відомих даних на виході.

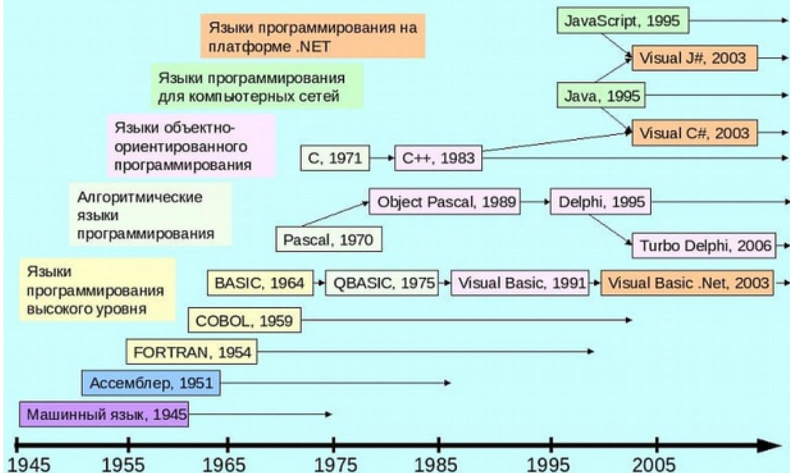
Агентно-орієнтоване програмування (АОП) — це парадигма програмування, де побудова програмного забезпечення зосереджена на концепції програмних агентів. На відміну від об'єктно-орієнтованого програмування, в основі якого є об'єкти (надають методи зі змінними параметрами), в основі АОП є агенти, визначені ззовні (з інтерфейсами та можливостями обміну повідомленнями). Їх можна розглядати як абстракції об’єктів. Повідомлення, якими обмінюються, інтерпретуються шляхом отримання «агентів» у спосіб, специфічний для його класу агентів.

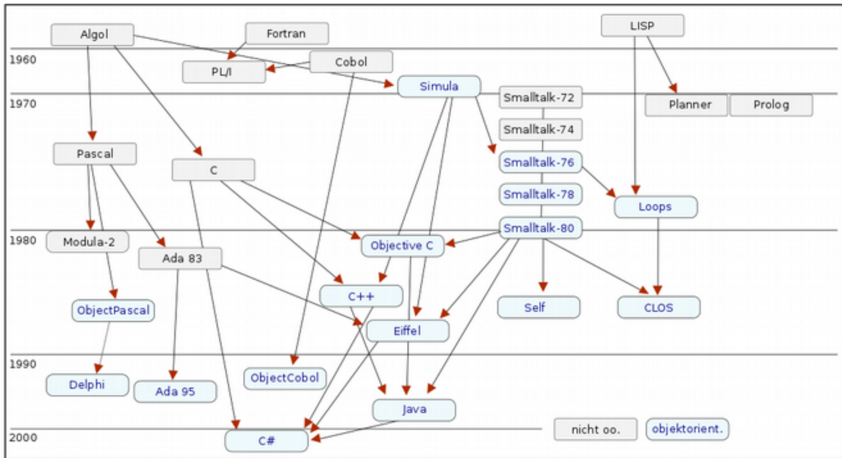
Куб парадигм



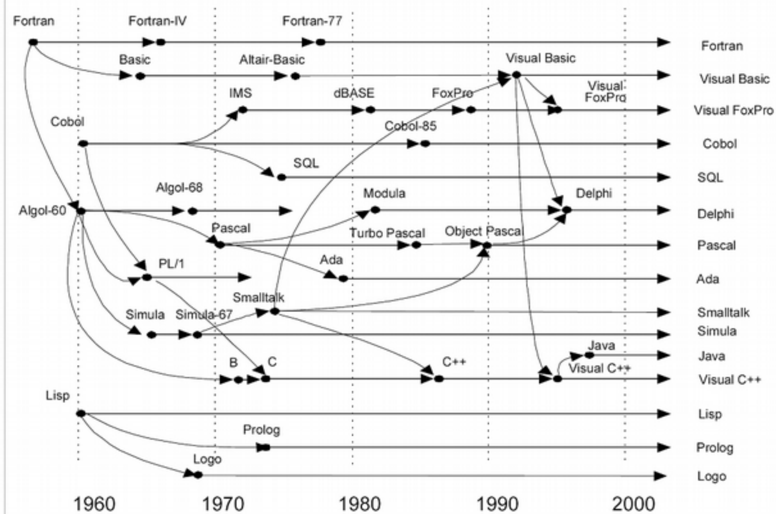
ИСТОРИЯ РАЗВИТИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

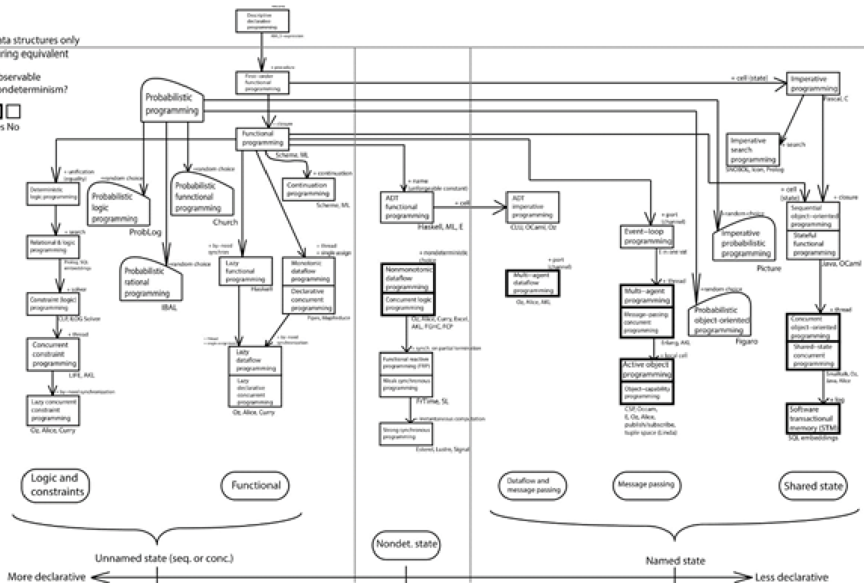
Алгоритм, записанный на «понятном» компьютеру языке программирования, называется программой.





Algol
Fortran
LISP
1960
Cobol
PL/I
Simula
1970
Smalltalk-72
Planner
Prolog
Smalltalk-74
Pascal
C
Smalltalk-76
Smalltalk-78
Loops
1980
Smalltalk-80
Modula-2
Objective C
Ada 83
C++
Self
CLOS
ObjectPascal
Eiffel
1990
Delphi
Ada 95
ObjectCobol
Java
2000
C#
nicht oo.
objektorient.


ГЕНЕАЛОГИЧЕСКОЕ ДЕРЕВО ЯЗЫКОВ ПРОГРАММИРОВАНИЯ
Fortran
Fortran-IV
Fortran-77
Basic
Altair-Basic
Visual Basic
IMS
dBASE
FoxPro
Visual FoxPro
Cobol
Cobol-85
SQL
Algol-68
Modula
Delphi
Algol-60
Pascal
Turbo Pascal
Object Pascal
Ada
PL/1
Smalltalk
Simula
Simula-67
Java
B
C
C++
Visual C++
Lisp
Prolog
Logo
Fortran
Visual Basic
Visual FoxPro
Cobol
SQL
Delphi
Pascal
Ada
Smalltalk
Simula
Java
Visual C++
Lisp
Prolog
Logo
1960
1970
1980
1990
2000



Descriptive declarative programming
Data structures only
Turing equivalent
Observable nondeterminism?
Yes No
First-order functional programming
+ cell (state)
Imperative programming
Pascal, C
Probabilistic programming
Functional programming
Imperative search programming
+ search
Scheme, ML
+ continuation
Continuation programming
Scheme, ML
Deterministic logic programming
Probabilistic logic programming
ProbLog
Probabilistic functional programming
Church
+ name (unforgeable constant)
ADT functional programming
Haskell, ML, E
+ cell
ADT imperative programming
CLU, OCaml, Oz
Event-loop programming
Imperative probabilistic programming
Picture
+ closure
Sequential object-oriented programming
Stateful functional programming
Java, OCaml
Relational & logic programming
Probabilistic rational programming
IBAL
Lazy functional programming
Haskell
Monotonic dataflow programming
Declarative concurrent programming
Nonmonotonic dataflow programming
Concurrent logic programming
Multi-agent dataflow programming
Oz, Alice, AKL
Multi-agent programming
Message-passing concurrent programming
Erlang, AKL
Probabilistic object-oriented programming
Figaro
Constraint (logic) programming
Concurrent constraint programming
LIFE, AKL
Lazy dataflow programming
Lazy declarative concurrent programming
Oz, Alice, Curry
Functional reactive programming (FRP)
Weak synchronous programming
FrTime, SL
Active object programming
Object-capability programming
CSP, Occam, E, Oz, Alice, publish/subscribe, tuple space (Linda)
Concurrent object-oriented programming
Shared-state concurrent programming
Software transactional memory (STM)
SQL embeddings
Lazy concurrent constraint programming
Oz, Alice, Curry
Strong synchronous programming
Esterel, Lustre, Signal
Logic and constraints
Functional
Dataflow and message passing
Message passing
Shared state
Nondet. state
Unnamed state (seq. or conc.)
Named state
More declarative
Less declarative

Історичний екскурс

HAL 9000 (укр. **ЕАЛ 9000)** — вигаданий комп'ютер з циклу творів «Космічна Одіссея» Артура Кларка, що мав здатність до самонавчання і є прикладом штучного інтелекту в науковій фантастиці.

HAL був створений 12 січня 1997 (у фільмі Стенлі Кубрика - 1992) в лабораторіях HAL в Урбані, штат Іллінойс зусиллями доктора Чандри. У фільмі HAL був здатний не тільки розпізнавати мову, зорові образи і спілкуватися на звичайній мові, але також читати по губах, створювати предмети мистецтва, проявляти емоції.

Існували версії, що HAL означає IBM зі зсувом значення кожного із символів на один, проте сам Кларк і його персонаж доктор Чандра, батько і вчитель комп'ютерів серії HAL, заперечують це, стверджуючи: «тепер будь-який дурень знає, що HAL означає *Heuristic ALgorithmic*»(Евристично запрограмований АЛгоритмічний комп'ютер).

Вважається, що на ідею HAL автора надихнув експеримент фізика Ларрі Джона Келлі-молодшого, який в 1961 році на машині IBM 7094 виконав комп'ютерний синтез мови, що стало одним з найвідоміших досягнень в історії Bell Labs.

ILLIAC IV є легендарним ключовим моментом у комп’ютерному проектуванні, і він займає подібну позицію в довгій і складній історії функціонального програмування.

Ідея ILLIAC IV полягала в тому, щоб відійти від послідовної моделі, яка домінувала в обчислювальній роботі з самого початку. До певних проблемних областей, таких як динаміка рідин, найкраще підходити за допомогою паралельної обробки, і ILLIAC IV був спеціально розроблений саме для такого типу паралельних задач: *задач, де одну інструкцію можна було застосувати паралельно до кількох наборів даних.*

Це відомо як **SIMD** (одна інструкція, кілька даних). Більш загальний випадок — кілька інструкцій, що працюють з кількома наборами даних, або MIMD — важчий. Але кожен вид паралельного програмування вимагає нових алгоритмів і нового мислення. І є запрошенням до функціонального програмування.

Згідно з Вікіпедією, функціональне програмування, або FP, «розглядає обчислення як оцінку математичних функцій і уникає зміни стану та даних, що змінюються». Це підійде для визначення, хоча нам доведеться копнути набагато глибше.

FP існував з 1950-х років, коли Джон Маккарті винайшов Lisp, але привід для паралельної обробки дав новий поштовх для функціонального програмування. Тоді уникати зміни стану було важко проковтнути, але це було якраз для моделі SIMD. Це призвело до розробки нових мов FP та функцій в існуючих мовах. У той час більшість людей, які займалися науковими обчисленнями, використовували Fortran, і було природно його покращити. Програмісти ILLIAC IV могли писати на IVTRAN або TRANQUIL або CFD, розпаралелених версіях FORTRAN. Була також паралелізована версія ALGOL.

Для відповідних проблем, які можна було б вирішити за допомогою SIMD, ILLIAC IV був найшвидшим комп’ютером у світі, і він мав неофіційне звання найшвидшої машини в світі аж до виведення з експлуатації в 1981 році.

Він був на порядок швидше, ніж будь-який інший комп’ютер того часу, і ідеально підходив для цільових додатків і функціонального програмування.

Але ця епоха раптово закінчилася. 7 вересня 1981 року ILLIAC IV назавжди закрили. Ви все ще можете побачити його частину на виставці в Музеї комп’ютерної історії в Маунтін-В’ю, трохи нижче по дорозі від Моффетт-Філд.

Чому закінчилася його епоха? З Вікіпедії: «Ілліак IV був членом класу паралельних комп’ютерів, які називають SIMD... Закон Мура обігнав спеціалізований підхід SIMD ILLIAC, зробивши підхід MIMD кращим для майже всіх наукових обчислень».

Але по дорозі він отримав інше місце в історії, як натхнення для HAL 9000 у 2001 році: Космічна Одіссея. Артур К. Кларк не був новачком у комп’ютерах: він спілкувався з Тьюрингом у Блетчлі-парку і був затятим послідовником розвитку мікрокомп’ютерів. Кларк був заінтригований, коли дізнався про роботу над ILLIAC IV в Університеті Іллінойсу в Урбана-Шампейн, і вшанував кампус у фільмі як місце народження HAL 9000.

Перемотайте вперед до 00-х. Випадком використання, який зробив FP гідним вивчення, знову була паралельна обробка, але викликана появою багатоядерних процесорів. «Виробники чіпів, — стверджував я тоді, — по суті сказали, що робота по дотриманню закону Мура тепер є проблемою програмного забезпечення. Вони будуть зосереджені на тому, щоб розмістити все більше ядер на кристалі, і ви повинні переробити своє програмне забезпечення, щоб скористатися можливостями паралельної обробки їхніх мікросхем».

Поштовхом для FP у 2000-х знову стало бажання звільнитися від послідовної моделі, і пропонувалися різні підходи. Якщо ви дуже інвестували в навички та інструменти Java, і не хотіли відкидати бібліотеки коду, навички та інструменти, на які ви розраховували, ви можете використовувати мову Scala Мартіна Одерського, нещодавно випущену на платформі Java. Він також пропонувався на платформі .NET, хоча (на жаль) підтримка була припинена в 2012 році. Користувачам .NET краще було шукати F#, створеного Доном Саймом на Microsoft Research. Якщо ви хотіли мати більш чисто функціональну мову, у вас був Erlang, розроблений для дуже паралельного програмування Джо Армстронгом з Ericsson, або Haskell, або багатьма іншими варіантами.

Усі ці мови пропонували можливість працювати у функціональній парадигмі. Дві визначальні особливості функціональної парадигми полягають у тому, що всі обчислення розглядаються як оцінка функції і що ви уникаєте зміни стану та змінюваних даних. Але є деякі додаткові загальні особливості функціонального програмування:

- Першокласні функції: функції можуть служити аргументами та результатами функцій.

- Рекурсія як основний інструмент для ітерації.

- Інтенсивне використання відповідності шаблону.

- Ліниве оцінювання, яке робить можливим створення нескінченних послідовностей.

Ітераційні програмісти, які тоді вперше звернулися до FP, можливо, додали: це повільно і заплутано. Насправді це не обов’язково ні те, ні інше, але це вимагає іншого способу мислення про проблеми та інших алгоритмів; і щоб працювати належним чином, йому потрібна була краща мовна підтримка, ніж це було звичайним у 2000-х. Це змінилося б протягом наступного десятиліття, а разом з цим змінилося б і самі цілі, заради яких людей залучали до FP.

Через десятиліття функціональне програмування знову розгорнуто. Тепер мовна підтримка тут, і випадок для FP ширший.

Хоча паралелізм традиційно був рушійною силою будь-якого поштовху до функціонального програмування, тепер, коли люди говорять про FP, вони, швидше за все, посилаються на здатність думати про проблеми на вищому рівні або на переваги незмінності. Замість того, щоб FP розглядали як дивний спосіб писати, щоб ефективно боротися з паралелізмом, новий аргумент полягає в тому, що це більш природний спосіб писати, ближче до концепцій та термінології вашої проблемної області. Люди використовують FP для програм, які не потребують паралелізму, просто тому, що це дозволяє їм програмувати більш ефективно та чітко, працюючи ближче до того, як вони думають про свої проблеми.

Вони кажуть, що замість того, щоб перекладати свою проблему на мову програмування, вони можуть адаптувати мову до проблеми.

І якщо у вас є декілька ядер і паралельність може принести користь вашому коду, ви можете отримати паралелізм безкоштовно. «Нещодавні розумні інновації в бібліотеках [Clojure] переписали функцію карти, щоб її можна було автоматично розпаралелювати, що означає, що всі операції з картою отримують переваги від підвищення продуктивності без втручання розробника», — сказав Ніл Форд.

Визнано, що FP добре підходить для одночасного звернення до людей, які пишуть багатопроцесорні програми, програми високої доступності, веб-сервери для соціальних мереж тощо. Абстракції вищого рівня FP приваблюють тих, хто шукає більш швидкий час розробки або більш зрозумілий код. Акцент FP на незмінності дуже привабливий для всіх, кого турбує надійність.

Така рекламна позиція є справжньою зміною від аргументів, висунутих для функціонального програмування за часів ILLIAC IV. Сьогодні є нові причини дивитися на FP, а також сильніша мовна підтримка FP та більший вибір підходів. Це починається з вибору мови. Ви можете зручно використовувати знайомі мови та їх інструменти, впроваджуючи функціональний підхід саме там, де ви вважаєте його корисним. Або ви можете перейти безпосередньо до мови, створеної з нуля для функціонального програмування.

Це захоплюючий час для вивчення функціонального програмування, і винагорода є значною. Але це все одно вимагає іншого способу мислення.

Функціональне мислення для імперативного розуму

Може бути важко точно визначити, що таке функціональне програмування. Це складно, оскільки цей термін, хоча теоретично добре визначений, на практиці охоплює кілька різних, хоча і пов’язаних, ідей. Якщо ви поговорите з хакером Clojure, ви, ймовірно, отримаєте повну кількість макросів. Програміст на Haskell може говорити про монади, а програміст на Erlang — про акторів.

Це все різні поняття. Макроси дають програмістам надзвичайно потужне метапрограмування, монади дозволяють нам безпечно моделювати змінний стан, а актори забезпечують надійний спосіб виконання розподіленого та паралельного програмування. Проте всі ці різні ідеї розглядаються як визначальні особливості функціональних мов. Широта концепцій може ускладнити розуміння того, про що йдеться. І все ж основна ідея дуже проста.

Тут мова буде йти про функції

По суті, функціональне програмування — це програмування з чистими функціями без побічних ефектів.

Це чиста функція:

```
f(x) = <raise power="2">x</raise>
```

І це:

```
public int incrementCounter(int counter) {

return counter++;

}
```

Але не це:

```
private int counter = 0;

public void incrementMutableCounter() {

counter++;

}
```

Перші два приклади збільшують лічильник, повертаючи нове ціле число, яке на одиницю вище, ніж передане ціле число. Третій приклад робить те ж саме, але робить це шляхом мутації частини стану, яка може бути спільною для багатьох частин програми.

Визначення: така функція, як incrementCounter, яка не залежить від стану, що змінюється, називається чистою функцією. І ця чистота має багато переваг. Наприклад, якщо у вас є чиста функція, яка виконує дорогі обчислення, ви можете оптимізувати свою програму, викликавши функцію лише один раз і кешуючи результат — метод, відомий як запам'ятовування.

Чисті функції також полегшують роздуми про програми. Об’єктно-орієнтована програма — це граф об’єктів, кожен із яких має власний пучок змінюваного стану. Зміна стану одного об’єкта може призвести до зміни стану іншого, можливо, віддаляється багато вузлів у графі. У програмі з лише чистими функціями така дія на відстані неможлива.

Це спрощений опис функціонального програмування.

Тепер мова буде йти про незмінність

На жаль, сувора чистота і реальний світ погано поєднуються. Чисті функції можна використовувати для моделювання одних доменів добре, інших не дуже. Компілятори - це чисті функції. Пошук Google ні.

Практичні функціональні мови програмування підкреслюють незмінність і функціональну чистоту, але вони повинні мати деякі засоби моделювання мінливого світу, який не досягає повної функціональної чистоти. У Haskell, ймовірно, найсуворішій функціональній мові, ви можете моделювати зміни за допомогою монад, а іншим чином підтримувати сувору чистоту.

Інші функціональні мови мають інші методи мінімізації та контролю зміни стану, які можуть бути не такими суворими, як у Haskell. Clojure, наприклад, використовує програмну систему транзакційної пам’яті в поєднанні з набором еталонних типів і диявольсько розумними незмінними структурами даних, щоб підтримувати високий ступінь чистоти, дозволяючи програмістам мати справу зі світом, що змінюється.

А оце про спосіб мислення

Отже, у першому наближенні, функціональне програмування — це програмування з чистими функціями та незмінним станом, на відміну від імперативного програмування, яке значною мірою покладається на змінюваність. Навколо цього незмінного ядра існує набір мовних прийомів і функцій, які замінюють змінні імперативний прийоми. Вивчення їх дасть нам глибше уявлення про те, що таке функціональне мислення та програмування.

Давайте подивимося на простий приклад: фільтрація списку, щоб він містив лише непарні числа.

```
public    List<Integer>    filterOdds(List<Integer>
list) {
      List<Integer>    filteredList    =    new
      ArrayList<Integer>();
            for(Integer current : list) {
                  if(1 == current % 2) {
                  filteredList.add(current);
                  }
            }
      return filteredList;
}
```

Це чудовий імперативний код. Ми перебираємо список і для кожного елемента перевіряємо, чи є він дивним, обчислюючи його модуль. Ми могли б, можливо, трохи прояснити його наміри, якби ми витягнули цю перевірку в допоміжну функцію і назвали її.

```
public List<Integer> filterOdds(List<Integer> list) {
List<Integer> filteredList = new ArrayList<Integer>();
      for (Integer current : list) {
            if (isOdd(current)) {
                  filteredList.add(current);
            }
      }
      return filteredList;
}
private boolean isOdd(Integer integer) {
      return 1 == integer % 2;
}
```

А що робити, якщо ми хочемо створити функцію, яка дозволить нам фільтрувати парні, а не шанси? Єдиний фрагмент коду, який потрібно змінити, це виклик **isEven** замість **isOdd**.

```
public List<Integer> filterEvens(List<Integer> list) {
List<Integer> filteredList = new ArrayList<Integer>();
      for (Integer current : list) {
            if (isEven(current)) {
                  filteredList.add(current);
            }
      }
      return filteredList;
}
private boolean isEven(Integer integer) {
      return 0 == integer % 2;
}
```

Це працює, але ми вчинили один із основних гріхів кодування. Більшість **filter-OutEvens** вирізається та вставляється з **filterOutOdds**. Те, що ми дійсно хочемо, це спосіб мати фільтр, який може використовувати якийсь довільний біт логіки для його фільтрації.

Давайте подивимося, як ми можемо це зробити на Java. І **isOdd**, і **isEven** беруть один аргумент і повертають логічне значення. Давайте визначимо інтерфейс, який відображає суть цього обчислення. Ми назвемо його предикатом, що є математичною назвою для функції, яка повертає логічне значення.

```
public interface Predicate {
public boolean evaluate(Integer argument);
}
```

Тепер ми можемо переписати **filterEvens** і **filterOdds**, щоб вони були більш загальними.

```
public List<Integer> filter(List<Integer> list,
Predicate predicate) {
List<Integer> filteredList = new ArrayList<Integer>();
for (Integer current : list) {
if (predicate.evaluate(current)) {
filteredList.add(current);
}
}
return filteredList;
}
```

Потім ми визначаємо два наші предикати.

```
class isEven implements Predicate {
      public boolean evaluate(Integer argument) {
            return 0 == argument % 2;
      }
}
class isOdd implements Predicate {
      public boolean evaluate(Integer argument) {
            return 1 == argument % 2;
      }
}
```

Тепер ми можемо просто створити екземпляр одного з предикатів і передати його в метод **filter**. Якщо ми придумаємо новий спосіб фільтрації нашого списку — скажімо, ми хочемо зберегти лише цілі числа, які є ідеальними квадратами — ми зможемо просто визначити предикат **PerfectSquare,** а не вирізати та вставляти всю функцію фільтрації.

Те, що ми щойно зробили з методом фільтра та інтерфейсом **Predicate**, моделює концепцію з функціонального світу: функції вищого порядку. Функція вищого порядку — це функція, яку можна передати в іншу функцію або повернути з неї. Давайте подивимося, як ми робимо подібну фільтрацію в Clojure — сучасному, функціональному варіанті Lisp.

```
(filter odd? [0 1 2 3])
(filter even? [0 1 2 3])
```

Це воно! Перше, що ви, напевно, помітили, це те, що він набагато коротший, ніж ця версія Java. По-друге, ймовірно, що дужки не на своєму звичному місці. Clojure та інші Lisps використовують префіксну нотацію для викликів функцій.

Це означає, що функція, яка викликається, є першою в дужках, а її аргументи йдуть пізніше. Крім синтаксичних відмінностей, зверніть увагу, як версія Clojure використовує всі вбудовані функції та мовні функції? Нам не потрібно визначати інтерфейс предикатів. дивно? це функція, яка приймає число і повертає істину, якщо воно непарне, а парне? робить те саме для парних чисел. Ми можемо передати ці функції безпосередньо у функцію фільтра, використовуючи потужність функцій вищого порядку.

Це перетворює наше оригінальне імперативне рішення, в якому ми написали код, що стосується тонких деталей ітерації списку, у дуже декларативне рішення. Ми працюємо на вищому рівні абстракції, який часто дозволяє нам описувати, які результати ми хочемо, а не деталі їх отримання.

Тому, коли люди говорять про функціональне програмування, вони зазвичай говорять щонайменше про дві окремі, але дуже тісно пов’язані речі. По-перше, вони говорять про програмування з чистими функціями. Оскільки для більшості проблем реального світу це невиправдана мрія, практичні функціональні мови програмування, як правило, спрощують використання незмінності, а також засоби, які контролюють мутацію, коли вам це абсолютно необхідно.

По-друге, вони говорять про стиль програмування, який виріс навколо цього функціонального ядра. Як ми бачили, цей стиль значною мірою покладається на функції вищого порядку та інші пов’язані методи. Ці методи часто створюють код, який працює на більш високому рівні абстракції, наприклад, використовує функцію фільтра, яку ми бачили вище, а не явну ітерацію.
Ці два аспекти функціонального програмування мають значні переваги. Надзвичайний наголос на незмінності полегшує роздуми про програми.
Поведінку функції можна зрозуміти, просто прочитавши код у самій функції, а не турбуючись про якийсь змінний стан, на який вона може покладатися, що знаходиться за сотні чи тисячі рядків. Використання функцій вищого порядку часто призводить до декларативного коду, який є коротшим і більш прямим, ніж імперативний еквівалент.

Тож якщо ви думаєте про ці дві речі, коли думаєте про функціональне програмування, ви не помилитеся: перевага програмування з чистими функціями та стиль програмування, який включає функції вищого порядку в декларативному коді.
Наступні дюжини або близько того розділів організовані за мовами, з кількома розділами, присвяченими функціональному програмуванню на кожній мові. Після цього ви знайдете колекцію розділів, які глибше стосуються функціонального програмування цими мовами. Ви можете перейти безпосередньо до мови, яка вас найбільше цікавить, або просто прочитати до кінця. Але ви не можете зробити краще, ніж почати з наступного розділу, в якому Венкат Субраманіам покаже вам, як Scala, гібридна функціональна мова, реалізує концепції, які ми обговорювали.

Мультипарадигменне програмування - програмування з одночасним використанням множини парадигм.

Основні підходи до організації мультипарадигменного програмування:

- створення нової мови програмування,
- розширення існуючої мови програмування,
- вбудовувані інтерпретатори,
- інтерпретатори, що розширюються,
- трансляція з однієї мови до іншої,
- збирання модулів, написаних різними мовами програмування,
- бібліотечне розширення існуючої мови програмування.

Мультипарадигменна мова програмування — як правило, мова програмування, розроблена спеціально як інструмент мультипарадигменного програмування, тобто образотворчі можливості якого спочатку передбачалося успадкувати від декількох, найчастіше неспоріднених мов.

Іноді термін **мультипарадигменна мова програмування** визначають як «мову, яка підтримує більше ніж одну парадигму програмування. Таке визначення є недостатньо точним, бо саме поняття парадигми програмування різні автори визначають по-різному. Наприклад, якщо вважати парадигмами програмування рекурсію, структурне програмування і присвоювання, то виявиться, що під це визначення підійдуть чи мало не всі існуючі мови програмування, за винятком деяких особливих випадків (наприклад, мови Haskell, де немає присвоювання в звичному вигляді).

Мета розробки мультипарадигменних мов програмування складається, як правило, у тому, щоб дозволити програмістам використовувати кращий інструмент для роботи, визнаючи, що ніяка парадигма не вирішує всі проблеми найлегшим або найбільш ефективним способом.

Один з найбільш амбітних прикладів — Oz, який є логічною, функціональною, об'єктно-орієнтованою, мовою конкурентного (паралельного) програмування тощо. Oz розроблено за десять років, її мета — об'єднати поняття, які традиційно пов'язані з різними програмними парадигмами.

Як одну з найбільш успішних мультипарадигменних мов програмування часто називають мову C++.

Мультипарадигменні мови

Узагальнене програмування (англ. generic programming) — парадигма програмування, що полягає в такому описі даних і алгоритмів, який можна застосовувати до різних типів даних, не змінюючи сам опис. У тому чи іншому вигляді підтримується різними мовами програмування.

Можливості узагальненого програмування вперше з'явилися в 1970-х роках у мовах CLU та Ada, а потім у багатьох об'єктно-орієнтованих мовах, таких як C++, Java, D і мовах для платформи .NET.

Термін **"Узагальнене програмування"** вперше було введене Девідом Массером і Олександром Степановим (1989) , які описували парадигму програмування, яка заснована на тому, що типи даних і структури даних є абстрактними і не впливають на конкретну реалізацію алгоритмів, а загальні функції реалізовані з використанням узагальнених формалізованих типів.

Приклади мультипарадигменних мов програмування, розділених за кількістю парадигм, що підтримуються:

Дві парадигми

- Функціональна , об'єктно-орієнтована:
  - Dylan .
- Функціональна , процедурна:
  - APL .
- Функціональна , логічна:
  - AFL;
  - Curry ;
  - Mercury.

Dylan — динамічна об'єктно-орієнтована мова програмування, націлена на швидку розробку програм; розроблений насамперед зусиллями Apple.

При необхідності, пізніше можна оптимізувати програми введенням інформації про типи. Dylan підтримує множинну спадковість, поліморфізм і багато інших парадигм. Мова загального призначення, придатна як для прикладного, так і для системного програмування.

Включає в себе збирання сміття, перевірки в ході виконання, відновлення після помилок і модульну систему.

Ім'я мови Dylan означає **«DYnamic LANguage»**.

Ця мова народилася в Apple на початку 1990 р. Її розробники хотіли створити покращений гібрид з елегантного варіанту LISP — Scheme, системи ООП CLOS від потужного промислового варіанту LISP — Common Lisp та ідеями з Smalltalk — і все це з нормальною загальноприйнятою системою позначень алголо/паскале-подібного синтаксису. Незабаром після цього аналогічний проект був запущений в Університеті Карнегі-Меллон — над створенням компілятора Dylan працювала знаменита команда Карнегі-Меллон з реалізації CMU Common Lisp. Іншу, комерційну версію з повною IDE випустила компанія Harlequin.

APL (вимовляють «ей-пі-ель», названа за книгою A Programming Language) — інтерактивна масиво-орієнтована мова програмування та інтегроване середовище розробки, що доступні від низки розробників і для більшості комп'ютерних платформ. Вона ґрунтується на математичній нотації, винайденій Кеннетом Айверсоном і його колегами, що пропонує спеціальні засоби для проектування і розробки цифрових обчислювальних систем, як апаратного забезпечення, так і програм.

APL має поєднання унікальних і порівняно рідкісних функцій, які привертають увагу програмістів і роблять її плідною мовою програмування:

• Вона лаконічна, використовує символи, а не слова і застосовує функції до всіх масивів без використання явних циклів.

• Абстрактна, орієнтована на розв'язання задач, орієнтована на написання програм незалежних від архітектури комп'ютера або операційної системи.

• Має одне просте, послідовне і рекурсивне правило пріоритету: правий аргумент функції — це результат всього виразу праворуч від неї.

• Це полегшує розв'язання проблем на високому рівні абстракції.

Перше втілення того, що пізніше перетворилося на мову програмування APL, було опубліковане і формалізоване в A Programming Language, книзі, що описує нотацію винайдену 1957 року Кеннетом Е. Айверсоном в Гарвардському університеті. Айверсон розробив математичну нотацію для роботи з масивами, якої він навчав своїх учнів.

1960 року він почав працювати на IBM, і, працюючи з Адіном Фалкофом, створив APL на основі своєї нотації. Вона була використана всередині IBM для коротких дослідних звітів на комп'ютерних системах, таких як Burroughs B5000 і його стековому механізмі, коли стекові машини оцінювалися порівняно з регістровими машинами IBM з метою розробки майбутніх комп'ютерів.

Крім того, 1960 року Айверсон уже використовував свою нотацію в чернетках 6-ї глави, що називалася «Мова програмування» для книги, яку він писав з Фредом Бруксом, Automatic Data Processing, яка потім буде опублікована 1963 року.
1962 року відома перша спроба використати нотацію для стандартизації набору інструкцій для машин, які пізніше стали сімейством IBM System/360.
1963 року д-р Герберт Хеллерман, що працював в науково-дослідному інституті IBM Systems, реалізував частину позначень на комп'ютері IBM 1620, і він був використаний студентами в спеціальному курсі середньої школи для розрахунків трансцендентних функцій підсумовуванням рядів.

Студенти випробували свій код в трансляторі доктора Хеллермана. Цю реалізацію частини позначень називають PAT (Personalized Array Translator).
1963 року Фалькоф, Айверсон, та Едвард Сассенгут, що на той час працювали на IBM, використали нотацію для формального опису архітектури і функціональності серії машин IBM System/360, що зрештою втілилося в статті, опублікованій в IBM Systems Journal 1964 року. Після публікації команда звернула свою увагу на втілення нотації в комп'ютерній системі. Одним з мотивів для цього фокусу на реалізації був інтерес з боку John L. Lawrence, який мав нові обов'язки в Science Research Associates, освітній компанії, купленій IBM 1964 року.

Лоуренс попрохав Айверсона і його групу, щоб вони допомогли із використанням мови як інструменту для розробки та використання комп'ютерів в освіті.
Після того, як Lawrence M. Breed і Philip S. Abrams зі Стенфордського університету приєднались до команди IBM Research, вони продовжували свої попередні роботи з реалізації запрограмованих в FORTRAN IV частини нотацій, що було зроблено для IBM 7090 під управлінням операційної системи IBSYS. Ця робота була закінчена в кінці 1965 року і пізніше стала відома як IVSYS (Iverson System, система Айверсона).

Основи цієї реалізації були докладно описані Abrams в Stanford University Technical Report, «An Interpreter for Iverson Notation» in 1966. Як і система PAT Геллермана раніше, ця реалізація не включала набір символів APL, а використовувала спеціальні зарезервовані слова англійською для функцій і операторів. Система була пізніше адаптована для системи з розділенням часу і, в листопаді 1966 року, була перепрограмована для комп'ютерів IBM/360 Model 50, що працювали в режимі розділення часу, і далі була використана всередині IBM.

Curry (Каррі) - вбудована мова проґрамування загального призначення.
У Curry об'єднані дві парадигми декларативного проґрамування - функціональна і лоґічна. Більш того, в цій мові використані найбільш важливі операційні принципи подібних декларативних мов. Названа на честь американського ученого Гаскелла Каррі.
Мова Каррі поєднує в собі можливості функціонального програмування (вкладені вирази, функції вищого порядку, ліниві обчислення), лоґічного програмування (лоґічні змінні, часткові структури даних, вбудована система пошуку) і методів проґрамування для паралельних систем (паралельне обчислення виразів з синхронізацією).

Більше того мова Каррі надає додаткові механізми в порівнянні з чистими мовами проґрамування (у порівнянні з функціональними мовами - пошук і обчислення за неповними даними, в порівнянні з лоґічними мовами - більш ефективний механізм обчислень завдяки детермінізму і викликом за необхідністю для функцій).

Mercury - мова функціонально-логічного програмування зі строгою типізацією, покликаний вирішити наступні дві проблеми, що виникають при використанні класичної мови логічного програмування Prolog:

1. проблема продуктивності. Сучасні реалізації мов логічного програмування за продуктивністю поступаються реалізаціям мов програмування імперативного типу.
2. проблема налагодження. Реалізації мов логічного програмування здійснюють менше перевірок під час компіляції, ніж реалізації мов програмування імперативного типу. Це змушує програміста знаходити помилки самому без будь-якої істотної допомоги з боку наладчика.

Язык разработан в Мельбурнском университете. Первую версию выпустили Fergus Henderson, Thomas Conway и Zoltan Somogyi 8 апреля 1995 года. Синтаксис Mercury частично унаследован от Пролога, система типов похожа на Haskell. Это чисто декларативный язык, разработчики полностью убрали из него все императивные возможности, что позволило улучшить встроенные в компилятор возможности оптимизации. Название Mercury дано в честь бога скорости Меркурия и отражает направленность на получение быстродействующих программ. Операции, при реализации которых обычно отказываются от чисто декларативного подхода, такие как ввод-вывод, выражаются в Mercury с помощью декларативных конструкций, используя линейные типы.

Три парадигми

- Функціональна, процедурна, об'єктно-орієнтована:
  - Perl; (з версії 5)
  - Python;
  - JavaScript;
  - Tcl;
  - PHP; (з версії PHP 5.3 частково підтримується функціональне програмування)
- узагальнена, процедурна, об'єктно-орієнтована:
  - C++;
  - D.

Чотири парадигми

- Функціональна, узагальнена, процедурна, об'єктно-орієнтована:
  - OCaml.
  - Common Lisp;
- Функціональна, процедурна, об'єктно-орієнтована, конкурентне:
  - Рубі.
- Об'єктно-орієнтована, узагальнена, процедурна, аспектно-орієнтована:
  - Java.

Шість парадигм

- Об'єктно-орієнтоване програмування, узагальнене програмування, процедурне програмування, функціональне програмування, подієво-орієнтоване програмування, рефлексивне програмування:
    - C#.

Сім парадигм

- логічна, програмування з обмеженнями, функціональна (як ледачі, так і «енергійні» обчислення), процедурна (імперативна), об'єктно-орієнтована, розподілена, паралельна
    - Oz

Наступна лекція буде присвячена функціональній мові — Лісп.