І.В. Ярош, магістр, гр. ПІ-45м, ФІКТ
Науковий керівник – к.ф-м.н., ст. викладач Л.В. Рудюк
Житомирський державний технологічний університет

МОВА ПРОГРАМУВАННЯ SCALA

Scala – це мультипарадигмова мова програмування, що поєднує властивості об'єктно-орієнтованого та функційного програмування. Назва Scala утворена зі слів «scalable» (маштабовна) та «language» (мова), для того щоб задекларувати, що мова може рости разом з вимогами користувачів.

Мова була створена в 2001-2004 роках в Лабораторії методів програмування EPFL. Scala була випущена для загального користування на платформі JVM в січні 2004 року і на платформі. NET в червні 2004 року. Планується продовжити роботу над формалізацією ключових аспектів мови і над розробкою оптимізації, виконуваних компілятором.

На дизайн Scala вплинули багато мов і дослідницькі роботи. Наступне перерахування включає частину робот втілила значне число концепцій і синтаксичних угод Java і C #. Спосіб вираження властивостей багато в чому запозичений з Sather (англ.). З Smalltalk взята концепція уніфікованої об'єктної моделі. З BETA прийшла ідея, що все, включаючи класи, повинно допускати вкладеність. Абстрактні типи в Scala дуже схожі на абстрактні типи сигнатур в SML і OCaml, узагальнені в контексті повноцінних компонентів. програми багато в чому схожі на Java-програми, і можуть вільно взаємодіяти з Java-кодом. включає одноманітну об'єктну модель в тому сенсі, що будь-яке значення є об'єктом, а будь-яка операція - викликом методу. - це також функціональний мова в тому сенсі, що функції - це повноправні значення.

- В Scala включені потужні і одноманітні концепції абстракцій як для типів, так і для значень.
- Вона містить гнучкі симетричні конструкції домішок для композиції класів і trait-ів.
- Вона дозволяє виробляти декомпозицію об'єктів шляхом порівняння із зразком.
- Зразки і вирази були узагальнені для підтримки природної обробки XML-документів.

В цілому, ці конструкції дозволяють легко висловлювати самостійні компоненти, які використовують бібліотеки Scala, не користуючись спеціальними мовними конструкціями.

В Scala використовується чиста об'єктно-орієнтована модель, схожа на застосовувану в Smalltalk: кожне значення - це об'єкт, і кожна операція - це відправка повідомлення. Кожна функція - це значення. Мова надає легковаговий синтаксис для визначення анонімних і каррінгових функцій. Кожна конструкція повертає значення. Зіставлення із зразком природно розширюється до обробки XML с допомогою регулярних виразів розрахованих на взаємодію з такими провідними платформами, як Java або C #. Вона розділяє з цими мовами більшість основних операторів, типів даних і керуючих структур.

В Scala використовується більшість керуючих структур Java, але традиційне для Java вираз for в їх число не входить. Замість цього існує for-comprehension, який дає можливість прямого перебору елементів масиву (або списку, або перерахування) без необхідності в індексації.

Кожен клас Scala успадкований від класу Scala.Any.

Підкласи Any потрапляють в одну з двох категорій: класи - значення, успадковані від scala.AnyVal, і посилальні класи, успадковані від scala.AnyRef. Будь-яке ім'я примітивного Java-типу відповідає класу-значенням, і відображається на нього за допомогою попередньо визначеного псевдоніма типу. У Java AnyRef ототожнюється з кореневим класом java.lang.Object. посилального класу зазвичай реалізується як покажчик на об'єкт, що зберігається в купі програми також класу-значення зазвичай представляється безпосередньо, без покажчиків-посередників.

Операції це ще один аспект уніфікованої об'єктної моделі Scala - кожна операція є відправкою повідомлення, тобто, викликом методу. Наприклад, додавання x + y інтерпретується як x. + (Y), тобто як виклик методу + з x в якості об'єкта-приймача і y як аргумент методу. Ця ідея, вперше реалізована в Smalltalk, адаптована до більш традиційного синтаксису Scala таким чином. По-перше, Scala розглядає імена операторів як звичайні ідентифікатори. Точніше, ідентифікатор - це або послідовність літер і цифр, що починається з букви, або послідовність операторних символів. Таким чином, можна визначати, наприклад, методи з іменами +, <= або :: Далі, Scala трактує ідентифікатор, що знаходиться між двома виразами, як виклик методу. Наприклад, в лістингу 1 можна було б використовувати синтаксис операторів (arg startsWith «-») як «синтаксичний цукор» для більш традиційного синтаксису (arg.startsWith («-»)).

Якщо методи - це значення, а значення - це об'єкти, то самі методи також є об'єктами. Насправді, синтаксис типів і значень функцій - це просто «синтаксичний цукор» для певних типів класів та примірників класів. Тип функції S => T еквівалентний параметри типу класу scala.Function1 [S, T], який визначений в стандартній бібліотеці Scala.

Модель даних для XML в Scala - це незмінне уявлення упорядкованого неранжірованного дерева. У такому дереві у кожного вузла є мітка, послідовність дочірніх вузлів і асоціативний список атрибутів і їх

значень. Все це описано в trait-е scala.xml.Node, який, крім того, містить еквіваленти XPath-операторів child і descendant-or-self, що записуються як і . Для елементів, текстових вузлів, коментарів, інструкцій з обробки і посилань на сутності існують конкретні підкласи.

Види представляють нову концепцію вирішення проблеми зовнішньої розширюваності - види (views). Вони дозволяють розширювати клас новими членами і trait-ами. Види в Scala переводять в об'єктно-орієнтоване уявлення використовувані в Haskell класи типів (type classes). На відміну від класів типів, область видимості видів можна контролювати, причому в різних частинах програми можуть співіснувати паралельні види.

Scala розглядає як кандидатів всі види, до яких є доступ з точки вставки без префіксного вираження. Це включає як види, визначені локально або в деякій області видимості, так і види, успадковані від базових класів або імпортовані з інших об'єктів виразом import. Локальний вигляд не ховає види, визначені в прилеглій області видимості. Вид застосуємо, якщо він може бути застосований до вираження, і він дозволяє відобразити цей вираз на бажаний тип. З усіх кандидатів Scala вибирає найбільш точно відповідний вигляд. У даному випадку точність інтерпретується так само, як при вирішенні перевантаження в Java і Scala. Якщо застосовного виду немає, або серед застосовних не вдається вибрати підходящий - генерується помилка.

Локальність забезпечується тим обмеженням, що в якості кандидатів розглядаються тільки ті види, які доступні без префікса. Види часто використовуються в бібліотеці Scala, щоб дати можливість Java-типам підтримувати trait-и Scala. Прикладом може служити Scala-trait Ordered, що визначає набір операцій порівняння. Види на цей тип від всіх базових типів і класу String визначені в модулі scala.Predef. Оскільки члени цього модуля неявно імпортуються в кожну Scala-програму, ці види завжди доступні. З точки зору користувача, це схоже на розширення Java-класів новими trait-ами.

Види стають ще більш корисними, якщо можна абстрагуватися від конкретного вставляється методу – це консервативне розширення Java, яке додає симетричну множинну диспетчеризацію та відкриті класи. Воно надає альтернативне рішення багатьох проблем, якими займається і Scala. Наприклад, множинна диспетчеризація дає вирішення проблеми бінарних методів, яка в Scala вирішується абстрактними типами. Відкриті класи надають рішення проблеми зовнішньої розширюваності, яка в Scala вирішується за допомогою видів. Тільки в MultiJava зустрічається можливість динамічного додавання нових методів в клас, так як відкриті класи інтегровані з звичайним процесом динамічного завантаження в Java. Навпаки, тільки Scala дозволяє визначати область видимості зовнішніх розширень класу в програмі.

Методи вищого порядку широко застосовуються в обробці послідовностей. У бібліотеці Scala визначено декілька видів послідовностей, серед них списки, потоки і ітераторів. Всі типи послідовностей успадковуються від trait-а scala.Seq; і всі вони визначають набори методів, що спрощують поширені завдання. Наприклад, метод map застосовує зазначену функцію одноманітно до всіх елементів послідовності, видаючи послідовність результатів функції. Інший приклад - метод filter, що застосовує задану функцію-предикат до всіх елементів послідовності, і який повертає послідовність елементів, для яких предикат вірний.

Система типів Scala забезпечує впізнання анотацій варіантності за допомогою відстеження позицій застосування параметрів типів. Якщо параметр типу зустрічається в якості типу значення, що повертається функції або типу, доступного тільки на читання властивості, він розглядається як коваріантний. Якщо параметр типу зустрічається як параметр методу або змінюваного властивості поля, то він розглядається як контрваріантний. Аргументи неваріантних параметрів типів завжди знаходяться в неваріантной позиції. Коваріантні і контрваріантні позиції міняються місцями всередині аргументу типу, відповідного контрваріантному параметру.