|  |  |
| --- | --- |
| лого-РГСУ-2015.png | **Российский государственный социальный университет**  **Факультет информационных технологий** |

Практическое задание № 3

**по дисциплине «Технологии программирования»**

|  |  |
| --- | --- |
| **ФИО студента** | Тарасова Андрей Валерьевич |
| **Направление подготовки** | Программная инженерия |
| **Группа** | ПИН-2016 |

**Москва 2019**

Логическое программирование (языки логического программирования, программа на языке логического программирования, преимущества и недостатки логического программирования)

В программах на языках логического программирования соответствующие действия выполняются только при наличии необходимого разрешающего условия на вывод новых фактов из данных фактов согласно заданным логическим правилам. Логическое программирование основано на математической логике (см. “*Логические операции. Кванторы*”, “*Логические выражения*”).

Первым языком логического программирования был язык Planner, он был разработан Карлом Хьюитом в Лаборатории искусственного интеллекта Массачусетсского технологического института в 1969 г. В этом языке была заложена возможность автоматического вывода (получения) результата из данных и заданных правил путем перебора вариантов (совокупность которых называлась планом). Но самым известным языком логического программирования является ПРОЛОГ (Prolog), который был создан во Франции в Марсельском университете в 1971 г. Аленом Кольмеро (Colmerauer).



Ален Кольмеро   
(Alain Colmerauer)

Программа на языке ПРОЛОГ содержит две составные части: факты и правила. Факты представляют собой данные, с которыми оперирует программа, а совокупность фактов составляет базу данных ПРОЛОГа, которая, по сути, является реляционной базой данных. Основная операция, выполняемая над данными, — это операция сопоставления, называемая также операцией унификации или согласования. Правила состоят из заголовка и подцелей. Выполнение программы, написанной на ПРОЛОГе, начинается с запроса и состоит в доказательстве истинности некоторого логического утверждения в рамках заданной совокупности фактов и правил. Алгоритм этого доказательства (алгоритм логического вывода) и определяет принципы исполнения программы, написанной на ПРОЛОГе.

В отличие от программ, составленных на языках процедурного типа, предписывающих последовательность шагов, которые должен выполнять компьютер для решения задачи, на ПРОЛОГе программист описывает факты, правила, отношения между ними, а также запросы по проблеме. Например, пусть у нас есть следующие факты относительно того, кто является чьей мамой:

мама("Даша","Маша").

мама("Наташа","Даша").

Кроме этого, имеется правило, вводящее отношение бабушка:

бабушка(X,Y):-

мама(X,Z),

мама(Z,Y).

Теперь мы можем делать запросы на предмет того, кто бабушка того или иного человека, или кто является внучкой (внуком) определенной женщины:

бабушка("Наташа",X).

Ответ на этот запрос система ПРОЛОГ выдаст так:

X=Маша

Возможности применения языка ПРОЛОГ весьма обширны. Среди наиболее известных — применение в символической математике, планировании, автоматизированном проектировании, построении компиляторов, базах данных, обработке текстов на естественных языках. Но, наверное, самое характерное применение ПРОЛОГа — это экспертные системы.

На сегодняшний день существует целый класс логических языков; так, от языка Planner также произошли логические языки программирования QA-4, Popler, Conniver и QLISP. Языки программирования Mercury, Visual Prolog, Oz и Fril произошли уже от языка Prolog.

Функциональное программирование (основные принципы функционального программирования, преимущества и недостатки функционального программирования)

Функциональное программирование — это практическая реализация идей Алонзо Чёрча. Не все идеи Лямбда-исчисления переросли в практическую сферу, так как лямбда-исчисления не учитывали физических ограничений. Тем не менее, как и ОО программирование, функциональное программирование — это набор идей, а не набор четких указаний. Существует много функциональных языков, и большинство из них делают одни схожие вещи по разному. В данной статье я объясню наиболее широко используемые идеи из функциональных языков используя примеры на Java (да, вы можете писать функциональные программы на Java если у вас есть склонности к мазохизму). В следующих нескольких разделах мы возьмём язык Java и внесём в него изменения, чтобы он превратился в пригодный к использованию функциональный язык. Начнём наше путешествие.  
  
Лямбда исчисление было придумано для изучения проблем, связанным с вычислениями. Функциональное программирование, стало быть, в первую очередь имеет дело с вычислениями, и, на удивление, использует для этого функции. Функция — это базовый элемент функционального программирования. Функции используются почти для всего, даже для простейших расчётов. Даже переменные заменяются функциями. В функциональном программировании переменные — это просто синонимы (alias) для выражений (чтобы нам не пришлось писать всё в одну строку). Их нельзя изменять. В каждую переменную можно записать только один раз. В терминах Java это означает, что все переменные объявляются как final (или const если имеем дело с C++). В ФП нет не-finalпеременных

final int i = 5;

final int j = i + 3;

Так как все переменные финальные, то можно сформулировать два утверждения. Нет смысла постоянно писать ключевое слово final, и нет смысла называть переменные … переменными. Теперь мы внесём два изменения в Java: каждое объявление переменной будет финальным, мы будем обращаться к переменным как к символам.  
  
Теперь вы, наверное, удивляетесь, как вообще можно написать что-либо достаточно сложное на таком языке. Если все символы неизменяемые, то мы в принципе не можем поменять состояние программы! Это не совсем верно. Когда Алонзо работал над лямбда-исчислением, у него не было нужды сохранять состояние, чтобы изменить его позже. Его интересовало проведение операций над данными. Тем не менее, было доказано, что лямбда исчисление эквивалентно машине Тьюринга. В нём можно делать всё то же, что возможно в императивных языках. Как же нам достичь тех же результатов?  
  
Оказывается, что функциональные программы могут хранить состояние, только они не используют для этого переменные. Они используют функции. Состояние хранится в параметрах функции, в стеке. Если хотите сохранить состояние, чтобы потом изменить его через время, то нужно написать рекурсивную функцию. Например, давайте напишем программу, которая переворачивает Java строку. Не забудьте, что все переменные объявляются как final [[5]](https://habr.com/post/142351/" \l "ref_5_down).

String reverse(String arg) {

if(arg.length == 0) {

return arg;

}

else {

return reverse(arg.substring(1, arg.length)) + arg.substring(0, 1);

}

}

Эта функция довольно медленная, потому что она повторно вызывает сама себя [[6]](https://habr.com/post/142351/" \l "ref_6_down). Здесь возможна утечка памяти, так как множество раз создаются временные объекты. Но это функциональный стиль. Вам может показать странным, как люди могут так программировать. Ну, я как раз собирался вам рассказать.

Преимущества функционального программирования

Вы, наверное, думаете, что я не смогу привести доводы в оправдание монструозной функции выше. Когда я только начинал изучать функциональное программирование, я тоже так думал. Я ошибался. Есть очень хорошие аргументы в пользу такого стиля. Некоторые из них субъективные. Например, программисты заявляют, что функциональные программы проще понять. Я не буду приводить таких аргументов, потому что всем известно, что лёгкость понимания — это очень субъективная вещь. К счастью для меня, есть ещё куча объективных аргументов.

Недостатки

Конечно бесплатный сыр бывает только в мышеловке. Ленивые вычисления тянут за собой ряд недостатков. В основном это недостатки от лени. В реальности очень часто нужен прямой порядок вычислений. Возьмём, например, следующий код:

System.out.println("Please enter your name: ");

System.in.readLine();

В ленивом языке никто не гарантирует, что первая строка выполнится раньше второй! Это означает, что мы не можем делать ввод-вывод, не можем нормально использовать нативные функции (ведь их нужно вызывать в определённом порядке, чтобы учитывать их побочные эффекты), и не можем взаимодействовать с внешним миром! Если мы введём механизм для упорядочивания выполнения кода, то потеряем преимущество математической строгости кода (а следом потеряем все плюшки функционального программирования). К счастью ещё не всё потеряно. Математики взялись за работу и придумали несколько приёмов для того, чтобы убедится в правильном порядке выполняемых инструкций не потеряв функционального духа. Мы получили лучшее от двух миров! Такие приёмы включают в себя продолжения (continuation), монады (monads) и однозначная типизация (uniqueness typing). В данной статье мы поработаем с продолжениями, а монады и однозначную типизацию отложим до следующего раза. Занятно, что продолжения очень полезная штука, которая используется не только для задания строгого порядка вычислений. Об этом мы тоже поговорим.

Языки функционального программирования

Свойства функциональных языков:

* краткость и простота;
* строгая типизация;
* модульность;
* функции — это значения;
* чистота (отсутствие побочных эффектов);
* отложенные (ленивые) вычисления.
* **Lisp** (List processor). Считается первым функциональным языком программирования. Нетипизирован. Содержит массу императивных свойств, однако в общем поощряет именно функциональный стиль программирования. При вычислениях использует вызов-по-значению. Существует объектно-ориентированный диалект языка — CLOS.
* **ISWIM** (If you See What I Mean). Функциональный язык-прототип. Разработан Ландиным в 60-х годах XX века для демонстрации того, каким может быть язык функционального программирования. Вместе с языком Ландин разработал и специальную виртуальную машину для исполнения программ на ISWIM’е. Эта виртуальная машина, основанная на вызове-по-значению, получила название SECD-машины. На синтаксисе языка ISWIM базируется синтаксис многих функциональных языков. На синтаксис ISWIM похож синтаксис ML, особенно Caml.
* **Scheme**. Диалект Lisp’а, предназначенный для научных исследований в области computer science. При разработке Scheme был сделан упор на элегантность и простоту языка. Благодаря этому язык получился намного меньше, чем Common Lisp.
* **ML** (Meta Language). Семейство строгих языков с развитой полиморфной системой типов и параметризуемыми модулями. ML преподается во многих западных университетах (в некоторых даже как первый язык программирования).
* **Standard ML**. Один из первых типизированных языков функционального программирования. Содержит некоторые императивные свойства, такие как ссылки на изменяемые значения и поэтому не является чистым. При вычислениях использует вызов-по-значению. Очень интересная реализация модульности. Мощная полиморфная система типов. Последний стандарт языка — Standard ML-97, для которого существует формальные математические определения синтаксиса, а также статической и динамической семантик языка.
* **Caml Light** и **Objective Caml**. Как и Standard ML принадлежит к семейству ML. Objective Caml отличается от Caml Light в основном поддержкой классического объектно-ориентированного программирования. Также как и Standard ML строгий, но имеет некоторую встроенную поддержку отложенных вычислений.
* **Miranda**. Разработан Дэвидом Тернером, в качестве стандартного функционального языка, использовавшего отложенные вычисления. Имеет строгую полиморфную систему типов. Как и ML преподаётся во многих университетах. Оказал большое влияние на разработчиков языка Haskell.
* **Haskell**. Один из самых распространённых нестрогих языков. Имеет очень развитую систему типизации. Несколько хуже разработана система модулей. Последний стандарт языка — Haskell-98.
* **Gofer** (GOod For Equational Reasoning). Упрощённый диалект Haskell’а. Предназначен для обучения функциональному программированию.
* **Clean**. Специально предназначен для параллельного и распределённого программирования. По синтаксису напоминает Haskell. Чистый. Использует отложенные вычисления. С компилятором поставляется набор библиотек (I/O libraries), позволяющих программировать графический пользовательский интерфейс под Win32 или MacOS.

.

