acm.mipt.ru

олимпиады по программированию на Физтехе

Парадигмы программирования в примерах

Вы наверняка не раз слышали словосочетания "процедурное программирование", "объектно-ориентированное программирование" и "функциональное программирование". Всё это обозначения различных парадигм (стилей) программирования. Есть целый зоопарк "различных программирований".

Перечислим их с короткими пояснениями:

императивное	программа = последовательность действий, связанных условными и безусловными переходами
процедурное	программа = последовательность процедур, каждая из которых есть последовательность элементарных действий и вызовов процедур, структурированных с помощью структурных операторов if, for и while
объектно-ориентированное	программа = несколько взаимодействующих объектов, функциональность (действия) и данные распределяются между этими объектами
функциональное	прогамма = система определений функций, описание того, что нужно вычислить, а как это сделать – решает транслятор; последовательность действий не прослеживается
продукционное (логическое)	программа = система определений и правил вида "условие => новый факт"
сентенциальное	программа = система правил вида "шаблон => трансформирующее действие"
событийное	программа = система правил вида "событие => новые события" + диспетчер событий
автоматное	программа = конечный автомат или автомат специального типа

Морфологический ящик 2x2

Чтобы разобраться в этом мире живой природы была сделана попытка поместить все парадигмы программирования в морфологический ящик 2x2:

действия \ условия	локальны	глобальны
локальны	императивное	
глобальны		сентенциальное

Эта простая классификация (Н.Н. Непейвода, "Стили программирования") уже может пролить свет на ключевые характеристики парадигм (стилей) программирования.

В **императивном программировании** в условном операторе обычно записываются простые условия на одну или несколько переменных. Более того, обычно, это локальные переменные. И действия, которыми оперирует "императивный" программист обычно просты и касаются изменения небольшого числа локальных переменных. Впрочем, следует отметить что локальность данных типичный но не опеделяющий признак императивного программирования.

В **сентенциальном программировании** условия формулируются как условия на состояние **всех данных**, и действие, которое осуществляется при удовлетворении условий может привести к модификации любой части **всех данных**.

Парадигма программирования проявляется не только в том, как организованы **условия и действия**, но также в том, как организованны **данные** в программе. Например, в **сентенциальном программировании** данные глобальны.

Наличие глобальных переменных, которые активно используются во многих функциях, говорит о том, что эти переменные определяют состояние системы, а это говорит о использовании парадигмы **автоматного** или **сентенциального** программирования.

Если наоборот – глобальных переменных нет, и у функций нет никаких побочных эффектов, то скорее всего программист мыслил согласно парадигме **функционального программирования**.

У **функции нет побочных эффектов** означает: то, что она использует как входные данные, и то, что она вычисляет (выходные данные) явно прописано в её семантике как аргументы и возвращаемое значение.

В первую очередь, это означает, что функция не меняет глобальных переменных.

Можно сказать, что функциональное и автоматное программирование дуальны по отношению к организации данных: в автоматном программировании данные глобальны, а в функциональном – локальны.

Программе в функциональном программировании естественно сопоставить ориентированный граф потока данных, в котором каждая вершина есть функция. В качестве входных данных функция использует результаты тех функций, из которых в неё ведёт стрелочка. Входные данные целиком "закачиваются" в одну из вершин этого графа, а затем по нему "растекаются" и результат вычислений как бы собирается в некоторой точке. Весь это граф с выделенным входом и выходом можно свернуть в вершину и рассматривать как отдельную функцию.

Примечание: Кроме графа потока данных в функциональном программировании можно также нарисовать граф зависимости определений функций – функции обычно бывают определены через другие функции, они вызывают их с аргументами, являющимися частями входных данных исходной функции. Это несколько другой граф. О нём мы здесь не говорим.

Программе в автоматном программировании тоже ставится в соответствие граф. Но по стрелочкам этого графа ничего не "течёт". Вершины этого графа есть множесвто различных состояний, а стрелочки соответствуют обычным условным

переходам между состояниями.

Таблица: Типы входных данных в различных парадигмах

тип входных данных	Парадигмы программирования
аргументы (flow)	функциональное
глобальные (global)	автоматное, сентенциальное, продукционное
локальные (local)	(создание временных локальных объектов используется во многих стилях)
события (event)	событийное

Для того, чтобы различить автоматное, сентенциальное, продукционное программирования, следует посмотреть на то, как устроены данные в глобальном хранилище данных и как устроена обработка данных (тип императива).

Таблица: Организация глобального хранилища

Парадигмы программирования	структура хранимых данных	тип императива
автоматное	состояние	диаграмма переходов
продукционное (логическое)	факты (n-арные отношения)	правила вида (логическое условие -> новые факты)
сентенциальное	текст (произвольные данные, обычно записанные на некотором формальном языке)	правила вида (шаблон -> трансформация)

Иногда рассматривают расширенное состояние автомата как прямое произведение макро-состояния (режимов) и состояния памяти (значений глобальных переменных). Вершины соответствуют режиму. Считывание входных данных в автоматном программировании может происходить в каждой вершине (в каждом режиме).

Например, телефон имеет следующие макро-состояния (режимы):

- ожидание звонка
- кто-то звонит
- идет разговор, установлено соединение
- просмотр телефонной книжки
- навигация по меню

Поток входных данных – это последовательность нажимаемых клавиш и (параллельно!) принимаемые телефоном сигналы.

Глобальными переменными (памятью) являются все настройки, адресная книга, списки вызовов, SMS сообщения, флаги о пропущенных вызовах или полученных SMS сообщениях и др.

Множество состояний телефона = Множество режимов x Множество состояний памяти

Конечно, значение режима также отображается в виде значения какой-то ячейки памяти, но программистам часто удобно делить всю память (или множество бит, определяющих состояние телефона) на режим и память. Это деление условно и целиком определяется программистом.

Обычно делят состояние на режимы и память так, чтобы в разных режимах программа работала согласно принципиально разной логике. Если есть два режима, в которых логика одинакова, а различие заключается, например, в цвете отображаемых букв и т.п., то естественно эти два режима объединить в один, а цвет букв вынести из понятия режима и рассматривать как глобальную переменную.

> Программа относится к автоматному стилю тогда, когда значительная часть логики программы заключена в диаграмме переходов между режимами.

Видно, что мобильный телефонный аппарат не является представителем чистой автоматной парадигмы. Во многих режимах присутствуют локальные данные (например в режиме набора номера его естественно помещать в локальную память, а после неудачного дозвона благополучно забыть). Кроме того, есть кусочки данных, которые явно соответствуют определенным режимам и совсем не используются в других режимах (например телефонная книга, SMS сообщения).

Глобальные настройки телефона являются ярко выраженными глобальными данными. Остальные данные связаны с конкретными режимами.

И здесь мы подбираемся к объектно-ориентированному программированию.

> Когда некоторая часть памяти явно связана лишь с малой частью функциональности всей системы, то естественно эту функциональность и соответствующие данные инкапсулировать в один объект.

Например объект "телефонная книжка", естественно реализовать в виде объекта, который отвечает за хранение всех записей, а также предоставляет функциональность добавления новой запись, редактирования и удаления существующей записи, поиска записи по первым буквам имени, поиска имени по номеру телефона.

> Деление данных на локальные и глобальные играет важную роль в проектировании системы. Собственно проектировании системы заключается в выделении базовых сущностей (функций, объектов, состояний), распределении ответственности (функциональности) между этими сущностями и определении, какие данные в каких функциях (объектах, состояниях) будут видны (доступны для чтения и модификации).

В **функциональном программировании** принято жесткое решение: объектов и состояний не существует, читать можно только те данные которые пришли на вход функций. Описание функций задает как из входных данных сконструировать те данные, которые нужно вернуть в качестве результата.

Не менее жесткие решения приняты в других **чистых парадигмах.**

Но на практике же находят применения различные гибриды парадигм.

Примеры программ с разными парадигмами

Здесь мы рассмотрим подробнее различные парадигмы программирования, и постараемся привести наиболее яркие демонстрирующие их примеры программ.

Императивное программирование

Императивное программирование – программирование от "глаголов". Программы представляют собой последовательность действий с уловными и безусловными переходами. Программист мыслит в терминах действий и выстраивает последовательности действий в более сложные макро-действия (процедуры).

```
Procedure Вскипятить_чайник
begin
   Зажечь плиту;
```

```
   Взять чайник;
   Налить в чайник воды;
   Поставить на плиту;
   Подождать 5 минут;
end
begin
   if Чайник не пуст then
      Вылить из чайника воду;
   Вскипятить_чайник;
end.
```

Программист, когда пишет такие программы, держит в голове имеющийся функционал – множество действий, которое на текущий момент реализовано в программе и библиотеках.

Кроме того, он знает входные условия (prerequisites) для каждого действия и старается им удовлетворить. Проверку этих условий полезно производить внутри самой процедуры в самом её начале.

Кроме того для самоконтроля полезно перед возвратом осуществлять проверку всех выходных условий (postrequisites), чтобы убедится, что необходимое макро-действие было произведено и произведено без ошибок.

Это позволяет отловить многие ошибки **на этапе тестирования программы.**

Но в индустрии программирования очень активная деятельность идёт в направлении разработки инструментов отлавливания ошибок **на этапе компиляции программы,** а не в момент её запуска.

Чем **императивное программирование** отличается от **процедурного**?

Ничем. Когда говорят о процедурном программировании, хотят подчеркнуть метасистемный переход от элементарных действий к более высокоуровневым действиям, представленных процедурами и функциями.

Что такое **структурное программирование**?

Термин структурное программирование ввёл Э.Дейкстра в 1975 году:
- Э.Дейкстра "Заметки по структурному программированию" (в составе сборника "Структурное программирование" / М.: "Мир", 1975.
- Э.Йордан "Структурное проектирование и конструирование программ" / М.: "Мир", 1979.

Можно рассматривать структурное программирование как стиль написания программ, исключающий оператор go to.

Сегодня многие по ошибке **структурное программирование** интерпретируют как процедурное или модульное программирование. Отчасти это так. Самое же главное в структурном программировании – это правильное составление правильной логической схемы программы, реализация которой языковыми средствами – дело вторичное. Программа должна представлять собой множество вложенных блоков (или по-другому – иерархии блоков), каждый из которых имеет один вход и один выход. При этом передача управления между блоками и операторами на каждом уровне дерева выполняется последовательно.

По-большому счету событийная модель программ организована точно также, просто роль связующего блока верхнего уровня выполняет ядро системы.

Объектно-ориентированное и событийное программирование

Объектно-ориентированное программирование – это программирование от объектов. Программа представляет собой набор связанных объектов. Каждый объект представляет собой набор каких-то данных и набор действий, которые он умеет делать. Естественно с объектом связывать именно те действия, которые необходимы при выполнении привязанных к нему действий. Эти действия называют **методами объекта.**

Типы объектов называются классами объектов. Объектно-ориентированный-программист описывает именно классы, а не объекты. В объектно-ориентированной программе также присутствует процедура, запускаемая при инициализации, которая создает объект базового класса, а затем этот объект уже сам всё что нужно делает – занимается порождением и уничтожением других объектов.

В примере про кипячение чайника объекты выделяются естественным образом – это плита и чайник.

Плита отвечает за зажигание и тушение конфорок, а также за уровень нагрева включенных конфорок. Поэтому интерфейс (множество действий с указанием семантики действий) плиты может выглядеть так:

```
class Плита {
   Горит Ли Конфорка? (конфорка)

   Зажечь Конфорку  (конфорка);

   Потушить Конфорку (конфорка);

   Установить Уровень Нагрева (конфорка, уровень);
}
```

Интерфейс чайника может быть, например, таким:

```
class Чайник {
   // boolean
   Пустой ли Чайник();

   // boolean
   В Процессе Нагрева();

   // Возвращает boolean (удалось или нет)
   Поставить На Плиту(плита, конфорка);
}
```

Перейдя к объектам, мы сразу же выходим в плоскость параллельных вычислений и событийно-ориентированного подхода.

Действительно, плита у нас становиться управляющим объектом, который управляет четырьмя конфорками -- тоже объектами, решающими некоторую сложную "вычислительную" задачу "нагрева чайников". Все конфорки работают

параллельно, независимо друг от друга, и плита контролирует их работу.

Такую независимость процесса жизни объектов друг от друга можно реализовать на современных компьютерах разными способами:

- Во-первых, её можно эмулировать – методы объектов исполняются в одном процессе, но каждый из них может считать, что живет независимо.
- Каждому объекту можно выделять отдельный процесс, и передачу сообщений между процессами можно реализовать с помощью различных механизмов взаимодействия процессов (sockets, pipes, messages, shared memory ...). Процессы могут реально исполняться на различных процессорах многопроцессорного компьютера.
- Объекты могут "жить" на разных компьютерах, и обмениваться сообщениями с помощью одного из возможных протоколов (sockets, TCP/IP, ...)

Таким образом, объектно-ориентированный подход видется достаточно удобных для реализации параллельных вычислений.

Поясним сказанное. Когда мы ставим чайник на плиту, вовсе не обязательно ждать 5 минут – вычислительное время дорого, и кроме того, мы не может предсказать сколько на самом деле продлится процесс "вычисления". Правильнее приделать к чайнику свисток и продолжить заниматься другими важными делами.

Объектно-ориентированный подход активно применяется при разработке графических интерфейсов: окошки, кнопки, текстовые надписи, чекбоксы и другие графические элементы интерфейса (graphical user interface controls) – всё это объекты, вложенные друг в друга и посылающие друг другу сообщения. Например, когда главное окно получает сообщение закрыться, оно должно отправить аналогичные сообщения всем своим дочерним окнам (объектам).

Кроме того, объекто-ориентированный подход естественным образом используется при программировании сложных структур данных. Объект, соответствующий структуре данных, инкапсулирует в себе все данные этой структуры а также всю функциональность связанную с извлечением и модификацией данных.

Приведём реализацию структуры данных **стэк** на объектно-ориентированном языке C++.

Стек – это структура данных для хранения последовательности элементов с двумя операциями – добавить элемент (push) и извлечь элемент (pop). Причём извлекаются элементы в порядке, обратном порядку добавления (последний добавленный элемент – First In First Out (FIFO)).

```
// File: stack.cpp
class Stack {
   int *m_data;
   int m_size;
   int m_pt;
public:
   Stack(int size) {
      m_size = size;
      m_data = (int*)malloc(m_size * sizeof(int));
      m_pt = 0;
   };
   ~Stack() {
      free(m_data);
   };
   int pop(void) {
      if(m_pt)
         return m_data[--m_pt];
      else
         return 0;
   };
   void push(int a) {
      if(m_pt >= m_size-1) {
         m_size = 10 + 2 * m_size;
         m_data = (int*) realloc (m_data, m_size * sizeof(int));
      }
      m_data[m_pt++] = a;
   };
   int empty() {
      return (m_pt == 0);
   }
};
```

Обратите внимание на то, что пользователю этого класса не нужно думать о памяти – ни о выделении, ни об освобождении. Стэк сам "позаботиться" о том, чтобы выделить нужное количество памяти. Если операцией pop() в стэк будет помещаться очередной элемент, который уже не помещается в память, на текущий момент предоставленную стеку, стэк выделит (realloc) под себя бОльшее количество памяти. А при завершении работы функции, в которой стэк был объявлен, автоматически вызовется деструктор ~Stack() и освободит память, которая использовалась под элементы стэка.

Кроме того Stack ответственнен за **консистентность** хранимых данных. Он не допустит того, чтобы указатель стэка m_pt вышел за пределы допустимых значений из полуинтервала [0, m_size).

Итак, объекты – это сущности, которые содержат в себе некоторую группу данных, и предоставляют функциональность, связанную с этими данными. Они ответственны за их консистентность.

И из этого определения можно сразу же вывести все проблемы, которые поджидают объектно-ориентированных программистов:

> В практических задачах невозможно разделить функциональность и данные на группы так, чтобы можно было каждой части данных поставить в соответствие часть функциональности, которая связана именно с этой частью данных. Это приводит к тому, что объекты должны обмениваться недостающими данными, дублировать их у себя. И программистам приходится мучаться с распределением ответственности между объектами, так как часто некоторая функциональность касается нескольких классов объектов.

Например, может оказаться так, что требуется функция, которая использует данные, хранимые в двух разных объектах, и тогда нужно будет решать какому из этих объектов новая функция больше подходит, и включить реализовать эту функцию как метод соответствующего класса. Но при этом придётся продумать механизм передачи необходимых данных из второго объекта. Вопросы эти часто в промышленном программировании решаются наобум, что приводит к разбалансировке архитектуры классов и необходимости **рефакторинга**.

ООП на C vs ООП на C++

Приведём, для примера два альтернативных подхода к организации кипячения чайников на кухне.

```
class Кухня {
   Добавить плиту (плита);
   Добавить чайник (чайник);
   Кипит ли чайник? (чайник);
```

```
    Где находится чайник (чайник);       // номер плиты+номер конфорки  или в шкафу?
    Поставить чайник на плиту (чайник, плита, конфорка);
    Какие чайники стоят на плите (плита); // список чайников
}
class Плита {
    Горит Ли Конфорка? (конфорка);
    Зажечь Конфорку (конфорка);
    Потушить Конфорку (конфорка);
    Установить Уровень Нагрева (конфорка, уровень);
    На тебя поставили чайник (чайник, конфорка);
}
class Чайник {
    Пустой ли Чайник?();
    В процессе нагрева?();
    Где ты находишься();           // плита+комфорка или в шкафу?
    Поставить на плиту(Плита, номер комфорки);
    Тебя поставили на плиту (плита, комфорка);
}
```

Объектно-ориентированный стиль программирования – это веселый стиль, в котором можно писать много строчек кода на "Ура!", думая о красоте и естественности.

Но может так оказаться, что критерий человеческой естественности при выборе типов объектов и распределения функциональности между объектами сыграет с Вами злую шутку.

Вопрос: в каком из данных классов естественно поместить хэштаблицу вида "чайник -> (плита, конфорка)"?

Посмотрим на другой стиль реализации аналогичной функциональности.

```
кухня_создать_кухню     ( <параметры кухни> );
кухня_создать_чайник    (кухня, <параметры чайника> );
кухня_создать_плиту     (кухня,  <параметры плиты> );
кухня_зажечь_конфорку   (кухня, плита, конфорка);
кухня_потушить_конфорку (кухня, плита, конфорка);
кухня_поместить_чайник  (кухня, чайник, плита, конфорка);

кухня_получить_состояние_чайника (кухня, чайник); // где стоит, кипит ли,  др.
кухня_получить_состояние_плиты   (кухня, плита);  // какие конфорки горят, какие чайники стоят.
```

Предложенная реализация объектно-ориентированного подхода без C++ позволяет не думать об группировке данных и функций. Программист думает только о организации данных внутри отдельной "кухни", и о структурах данных, которые нужны для эффективной работы "кухни". Вся функциональность реализуется в виде набора глобальных функций, которые не распределяются между классами.

Такой подход позволяет экономить силы и на этапе проектирования и на этапе разработки. В случае зоопарка из нескольких классов (кухня, чайник, плита, конфорка, ..., – на практике их число может быть очень большим, порядка сотни) в разных классах начинают встречаться функции с одинаковым смыслом. Например, в классе Плита есть функция, определяющая, кипит ли чайник. И в классе чайник есть аналогичная функция. Эти функции должно быть как-то используют друг друга. Здесь общепринятого стиля нет. Некоторые помещают "внутрь" чайника ссылку на плиту, и о своем состоянии чайник узнает у плиты. Некоторые делают наоборот – плита опрашивает стоящий на ней чайник, кипит ли он.

Псевдо объектно-ориентированный подход более удобен для отработки исключительных ситуаций, когда функция вызывается с некорректными данными (например, чайник перемещается на конфорку, на которой уже стоит чайник). В ООП программисты часто запутываются, какой из классов ответственен за обработку исключительных ситуаций.

Пусть например, кухне поступил сигнал переместить чайник на некоторую конфорку. Кухня перенаправила это указание соответствующей плите. Плита указала переместиться чайнику. Но чайник не смог это сделать по какой-либо причине. Если программист не продумает механизм обработки исключительных ситуаций, информация о невозможности перемещения конкретного чайника может не дойти до кухни, дойти в неверном виде или без полной информации о неполадке.

Реализация функциональности в виде набора глобальных функций, первый аргумент которых является ссылкой на базовый объект, позволяет избежать дублирования одной и той же функциональности. Полную информацию о случившихся неполадках удобно хранить внутри структуры данных "кухня".

Обычно делают так, что все функции возвращают код возврата, который равен 0, если все хорошо, и некоторому ненулевому значению `errno` – иначе. И также добавляют функцию для получения описания ошибок:

```
кухня_описание_ошибки(errno); // возвращает текстовое описание ошибки
```

Методология исключительных ситуация, существующая в C++ и Java, безусловно, заслуживает внимания. Она очень хороша. Плохо, что она позволяет программистам писать на начальных этапах код не думая – это может плохо сказаться на этапе созревания проекта.

С одной стороны, язык программирования и методология должны быть такими, чтобы принуждать программиста тщательно продумывать архитектуру проекта на первом этапе программирования. А с другой – усилия затрачиваемые на проектирование системы должны быть оправданы. С одной стороны, не должно быть лишних структур, которые делают проект костным и неповоротливым, С другой – данные и функциональность должны быть как то структурированы, чтобы исключить хаотизации кода проекта.

Калькулятор

Язык арифметических выражений – это ещё не язык программирования. Но на задаче разработки программы-калькулятора можно обозначить много ключевых моментов теории языков программирования вообще.

Калькулятор выражений в обратной польской нотации на языке C++ (событийное программирование)

Рассмотрим несколько необычную запись (нотацию) арифметических выражений, в которой сначала идут два операнда, разделённые пробелом, а затем знак арифметической операции. Например,

```
    5 6 *                    -->   5 * 6
   (5 6 *)  (2 3 *) +        -->   (5 * 6) + (2 * 3)
    1 2 3 4 5 * + - /        -->   1  / (2 - (3 + (4 * 5)))
```

Эта нотация называется обратной польской нотацией. Заметьте, что во втором примере скобочки можно опустить

выражение по прежнему будет интерпретироваться однозначно.

Транслятор-вычислитель таких выражений естественно построить на основе стека. Каждое считываемое число помещается в стек, а как только встречается арифметическая операция, из стека считываются два элемента (a = pop(), b = pop()), над ними производится соответствующая операция и результат заносится в стэк (push(a * b)).

Ниже приведенеа программа, в которой используется class Stack, который мы определили выше.

```
#include <stdio.h>
#include <malloc.h>
#include <stack.h>

int
main() {
   class Stack s(100);
   int i;
   while(!feof(stdin)) {
      int c = getchar();
      int x;
      switch (c) {
         case EOF: break;
         case ' ': break;
         case '\n': printf("Result = %d\n", s.pop()); break;
         case '+': s.push( s.pop() + s.pop() ); break;
         case '-': s.push(-s.pop() + s.pop() ); break;
         case '*': s.push( s.pop() * s.pop() ); break;
         default:
            ungetc(c, stdin);
            if(scanf("%d", &x) != 1) {
               fprintf(stderr, "Can't read integer\n");
               return -1;
            } else {
               s.push(x);
            }
            break;
      }
   }
   RESULT:
   i = 0;
   while(!s.empty()){
      printf("Result%d = %d\n", i,  s.pop());
      i++;
   }
}
```

В основе этой программы лежит **событийная парадигма**, хотя кого-то это может и удивить.

Да, здесь имеется оператор while – представитель структурного программировния, также испольуется объект стек --- представитель объектно-ориентированного программирования.

Тем не менее, базовая логика программы событийная. Роль событий здесь играют входные символы. Программа устроена как бесконечный цикл обработки приходящих событий (пока не поступит сигнал конца входа EOF).

Результатом обработки события являются события, посылаемые объекту "стек" – это запросы pop и push.

Это не автоматный стиль по той причине, что здесь нет режимов (ярко выраженных, разнотиповых состояний) и какой-то сложной диаграммы переходов между этими режимами.

Нечестный калькулятор на языке Perl:

```
$_ = <>;
print eval $_;
```

Калькулятор на Perl, написанный в сентенциальной парадигме

Приведём честный калькулятор на языке Perl, демонстрирующий метод **сентенциального программирования**.

```
$_ = <>; chomp;
sub mul($$)   { return $1*$2;}
sub sum($$)   { return $1+$2;}
sub minus($$) { return $1-$2;}
while(
   s/(\d+)\s*[*]\s*(\d+)/mul($1,$2)/e ||
   s/(\d+)\s*[+]\s*(\d+)/sum($1,$2)/e   ||
   s/(\d+)\s*[-]\s*(\d+)/minus($1,$2)/e ||
   s/\(\s*(\d+)\s*\)/$1/e
   ) { print "$_\n";};
print "Result=$_\n";
```

Например

```
3*(4+5)+2
3*(9)+2
3*9+2
27+2
29
Result=29
```

По сути, данный калькулятор работает согласно пяти сентенциям:

- если число окружено скобками, то скобки можно убрать
- подвыражение вида "число * число" можно заменить результат умножения
- подвыражение вида "число + число" можно заменить результат сложения
- подвыражение вида "число - число" можно заменить результат вычитания

Следует отметить, что несмотря на всю простоту и естественность этих сентенций, данная программа работает *неправильно*. Это связано с приоритетами операций. Данная программа не соблюдает правило, что у умножения приоритет больше. Также неверно говорить, что в этой реализации калькулятора приоритеты у сложения и умножения одинаковы.

Пример:

```
(1*2)+3*(4+5)
(2)+3*(4+5)
(2)+3*(9)
```

```
2+3*(9)
5*(9)
5*9
45
Result=45
```

В данном примере получается *неверный результат*. Близость сентенций к естественному языку мешает программисту видеть подводные камни.

Тем не менее, сентенциальный стиль программирования заслуживает внимания. Приведённую выше программу на Perl нужно лишь чуть чуть модифицировать, чтобы получился верно работающий калькулятор, в котором умножение имеет больший приоритет, нежели сложение.

Логику конвертации данных из одного формата в другой часто можно записать в виде сентенций. Например, при конвертации из LaTeX в HTML можно использовать сентенции:

```
СЕНТЕНЦИЯ 1
  \begin{center} <текст, в котором нет begin{center}>  \end{center}
    заменить на
  <center>  <тот же текст> </center>

СЕНТЕНЦИЯ 2
  \section{ текст без переноса строчки }
    заменить на
  <h2>  <тот же текст> </h2>

СЕНТЕНЦИЯ 3
   символ ~ заменить на  

....
```

Опытный программист знает, что сентенциальный подход, в принципе, можно использовать на практике (для своих маленьких нужд) -- программы пишутся достаточно легко и быстро. Но работают они медленно и ненадёжно. Надёжные программы в сентенциальном стиле писать сложно.

Например, символ "тильда" при конвертации из LaTeX в HTML нужно заменять на " " только тогда, когда этот символ находится в тексте (а не в формуле) и когда перед ним не стоит символ "обратный слэш". Такую сентенцию сложно записать в виде регулярного выражения на Perl или на каком-либо другом языке сентенций.

Классификация стилей программирования

Подведём итог.

Приведём базовые признаки языка программирования и стиля программирования (часто один и тот же язык допускает различные стили программирования), которые полно определяют тип программирования.

Грубо говоря, эти признаки касаются способа организаций хранения данных и доступа к данным и способа организации и типа императивных данных (логики действий, зависимостей, правил обработки).

- тип локализации (видимости) данных
  - (Global) данные глобальны
  - (Local) данные локальны
  - (Flow) данные передаются как аргументы функции
  - (NONE) данные отсутствуют
- структура хранимых данных
  - хранилище фактов
  - хранище объектов (переменные, структуры данных, объекты)
  - состояние
- принцип доступа к данным
  - именованный
  - адресный
  - сложный (по запросам)
    - SQL (Relational DBs)
    - Logical queries (Prolog)
  - линейный
    - FIFO (очередь)
    - FILO (стек)
- тип императивных данных (метод описания логики работы программы)
  - действия + условные переходы
  - математические зависимости функций друг от друга
  - диаграмма переходов
  - правила вида "логическое условие -> изменение данных"
  - правила вида "шаблон -> изменение данных"
- принцип взаимной организации имеративных и декларативных данных
  - инкапсуляция в одном (объектно-ориентированный подход)
  - декларативных данных как таковых нет, есть аргументы, поступающие на вход функциям (функциональный подход)
  - простая логика действий привязана к состояниям глобальных данных
  - ....

Таблица базовых парадигм программирования

стили\признак	Единица программы	Входные декларативные данные	Выходные декларативные данные	Входные императивные данные	Выходные императивные данные	Модульность
Автоматное программирование	transition_act	dequeued token from global_queue	changed global_state	make transition (atom)	none, {events}, {enqueue tokens to global_queues of other automats}	Libraries: automats
Функциональное программирование	definition	flow	flow	calculate	calculate others (composite)	Libraries: functions
Процедурное программирование	procedure	flow,{global}	flow,{global}	execute	execute others (composite)	Libraries: procedures

Сентенциальное программирование	transform rule (pattern->transform action)	global	global	query	query subqueries	Libraries: transformation rules
Логическое программирование	inference rule (logical condition->new fact)	global	global	query	query subqueries	Libraries: inference rules + facts

Таблица характеристик типичных парадигм программирования для базовых языков программирования

язык\признак	Единица программы	Декларативные данные	Императивные данные	Модульность
Assembler	action_simple(Global:restricted) + data_simple	Global:(named+addressed)_simple+FIFO	Procedures:Local+Global	Libraries: Functions
Си	action_macro(Flow+{Global}) + data_simple	(Global + Local): (named+addressed)_simple+Flow	Procedures:Local+Global	Libraries: Functions
Prolog	Rule+Fact+Query	Global:[d]named_relations	Rules:Global	Libraries: (Rules+Facts)
Haskell	Function(Flow)	Flow	Functions:Global	Libraries: Functions
Tcl	Function(Flow+Global)	(Global+Local):[d]named_structures	Functions	Namespace: (Functions+Objects)
Java	Class()	(Global+Local):named_(simple+dobjects)	Functions	Namespace: (Functions+Classes+Objects)

Другие характеристики языков

Остальные характеристи языков хотелось бы вынести за понятие парадигмы программирование. Их очень много, они касаются конкретных маленьких (но довольно важных) деталей.

Динамическое пространство имен

Если язык допускает именованние единиц данных, то возникает вопрос о возможности изменения имён и создания новых именованных единиц во время выполнения программы.

Скриптовые языки Tcl, Perl, ... позволяют это делать. Языки C, C++. Java, Pascal – нет. Имена переменных в этих языках задаются на этапе написания программы.

В языке Prolog хранимые данные есть множество фактов – n-арных отношений. Эти отношения имеют имена, а отдельные факты обычно не именуют (хотя потенциальная возможность есть). Таким образом, в Prolog есть возможность динамического именования отношений.

Наличие типизирования

Язык называется *типизированным*, если и к каждой единице данных привязан определенный тип данных (целое число, символ, строка, объект класса A, ...).

C, Pascal, Java, Ruby, ... – типизированные языки программирования.

Perl, Python, – нетипизированные (слаботипизированные) языки программирования. Типы в этих языках присутствуют -- они различают функции, хэштаблицы, скаляры, массивы. Про эти языки правильно говорить, что у них отсутствует типизация скалярных типов данных.

SQL – типизированный язык программирования, но без возможности именования отдельной единицы данных (ячейки таблицы).

Динамическое типизирование

Если для именованных единиц данных в типизированном языке программирования есть возможность менять тип данных в ходе выполнения программы, то говорят, что язык с динамическим типизированием.

Ярким примером является язык Ruby, где в случае, когда результат арифметической операции приводит к целому числу, большему 2^{32}, результат (и переменная, в которую он помещается) автоматически меняет тип с FixedInteger на BigInteger.

Полиморфизм сущностей

Если в языке программирования за одним и тем же именем могут скрываться различные сущности, то говорят, что в языке реализован полиморфизм объектов.

Какую из сущностей использовать в каждом конкретном случае, транслятор языка определяет из контекста, в котором встретилось имя.

Например, если программе на языке Perl если после имени стоит квадратная скобка, то используется массив с данным именем, а если стрелка (->) – то используется хэштаблица с данным именем. Это две сущности, которые содержать принципиально различные данные, но имеют одно и то же имя.

Полиморфизм функций

Если в языке программирования есть функции и можно создавать различные функции с одним и тем же именем, но различающиеся входными аргументами, то говорят, что в языке поддерживается полиморфизм функций.

Обычно, какую именно функцию имел в ввиду програмист, транслятор языка определяет из количества переменных и типов переданных аргументов.

Полиморфизм функций встречается в типизированных языках программирования, но бывают и исключения.

Можно считать, что полиморфизм функции есть частный случай полиморфизма сущностей.