Лабораторная работа №3 по курсу «Проектирование программ в интеллектуальных системах» (2 курс 1 семестр)

Тема: Объектная модель в разработке программной системы. Исключения.

Цель: Получить навыки проведения объектно-ориентированного анализа предметной области.

Содержание

1 Задание

Выбрать вариант задания, провести объектно-ориентированный анализ предметной области, выделить классы и связи между ними. На основании объ ектной модели реализовать программу на C++ в соответствии с требованиями варианта задания.

Общие требования к выполнению лабораторной работы:

- использование наследования;
- использование полиморфизма;
- использование обработки исключений при возникновении ошибочных ситуаций.

При выполнении этой лабораторной работы студентам не стоит бояться выделять классы, даже если они небольшие (например, состоят из одного поля), потому что в этой работе изучается объектно-ориентированный подход.

Для выполнения лабораторной работы необходимо изучить главу 12 «Объектно-ориентированное программирование: Наследование», главу 13 «Объектно-ориентированное программирование: Полиморфизм», главу 16 «Управление исключениями» из книги Х. Дейтел, П. Дейтел «Как программировать на C++» 2008 года издания.

Хорошим подспорьем в выполнении этой лабораторной работы будет изучение семинара 2 из книги Павловская Т. А., Щупак Ю. А., «C++. Объектно-ориентированное программирование: Практикум».

2 Варианты заданий

2.1 Модель банкомата

При выборе этого варианта студенту необходимо изучить работу банкомата «Беларусьбанк». После этого необходимо выявить взаимодействующие объекты в рамках сессии работы банкомата. На основе выявленных объектов сформировать объектную модель, которая описывает работу банкомата. В данном варианте необходимо обратить особое внимание на классы для представления экранов банкомата. Именно они будет формировать основную иерархию классов данного варианта.

Разработанная объектная модель должна соответствовать следующим требованиям:

- включать классы для представления основных взаимодействующих сущностей в процессе работы банкомата (например, карточка, пользователь, сессия, банк, кард-счет и др.);
- включать иерархию классов для представления экранов банкомата.

Реализованная на основе объектной модели программа должна:

- получать в качестве командных аргументов файл с описанием карточки клиента и файл с описанием базы данных банка;
- моделировать работу банкомата в консольном режиме.

2.2 Модель животного мира

При выполнении этого варианта студент должен будет разработать игровую модель животного мира. Собственно мир, в котором живут обитатели (растения и животные), представляется прямоугольным полем произвольного размера. Прямоугольное поле разбито на равные квадраты (клетки), и в каждой клетке может находиться максимум четверо обитателей. Моделирование жизни в таком мире происходит в пошаговом режиме, при котором каждый шаг содержит следующие четко выделенные стадии:

- Передвижение
- Питание
- Размножение
- Умирание от старости или от голода

Способы поведения обитателей в каждой из стадий будут зависеть от типа обитателя. Студенту предлагается три основных типа обитателей со следующими свойствами:

- Растение:
  - в клетке может находиться только одно растение;
  - не передвигается;
  - имеет пункты жизни (как в ролевых играх):
    - теряет некоторое количество пунктов жизни каждый ход (эффект умирания);
    - если не остается пунктов жизни, то растение умирает;
  - не питается;

  - размножается в текущую клетку или на соседние клетки:
    * не нужна парная особь для размножения;
    * растение может размножаться на соседнюю клетку даже, если там уже есть другое растение с неполным процентом жизни (в этом случае размножение обновит жизнь растения до 100

- Травоядное:
  - передвигается (количество клеток, на которое может передвинуться, зависит от вида животного);
  - питается растением:
    * растение должно находиться на той же клетке;
  - имеет размер (зависит от вида животного):
    * более крупные травоядные съедают больше пунктов жизни растения;
    * влияет на то, что может ли оно быть съедено более крупным хищником;
  - имеет срок жизни;
  - может умереть, если голодно определенное количество ходов;
  - имеет один пункт жизни, т.е. жизнь у травоядного одна, и хищник не может съесть только часть, а съедает целое животного;
  - некоторые виды травоядных животных имеют некоторый шанс убежать от хищника;
  - размножение:
    * имеет пол;
    * нужна парная особь для размножения на той же клетке;
    * детеныш рождается сразу на ту же клетку.

- Хищник:
  - передвигается (количество клеток, на которое может передвинуться, зависит от вида животного);
  - имеет размер (зависит от вида животного):
    * более крупному хищнику надо больше есть;
    * более крупный хищник может съесть более крупное травоядное;
  - имеет срок жизни;
  - может умереть, если голодно определенное количество ходов;
  - питается травоядными:
    * травоядное должно находиться на той же клетке;
    * всегда съедает целое травоядное, если оно не убежало;
  - размножение:
    * имеет пол;
    * нужна парная особь для размножения на той же клетке;
    * детеныш рождается сразу на ту же клетку.

Изучив и уточнив приведенные правила, студенту необходимо разработать объектную модель предметной области и программу, которая будет осуществлять моделирование процесса жизни. Причем обитателями мира будут конкретные виды животных. Например, гепард – хищник среднего размера, который может передвигаться на 2 клетки за ход. Как видно из примера, конкретный вид животного может иметь какие-то специальные качества и какое-то специальное поведение. К растениям это не относится, потому что в модели есть только один вид растения. При разработке иерархии классов обязательно должен быть применен такой атрибут любого объектно-ориентированного языка, как наследование.

Разработанная объектная модель должна соответствовать следующим требованиям:

- включать классы видов животных и растений (количество на усмотрение разработчика)
- классы животных и растений должны составлять иерархию, чтобы общие свойства и логика работы описывалась надклассом
- включать классы описания животного мира (игрового поля)
- предусматривать механизм моделирования животного мира
- включать классы, которые обеспечивает разгрузку модели из текстового файла

Реализованная на основе разработанной модели программа должна:

получать имя входного файла с описанием мира и описание режима работы (автоматическое моделирование или пошаговое моделирование с ожиданием подтверждения от пользователя) через командный аргумент;

выводить в консоль состояние игрового поля на каждом ходу.

Совет

Пример реализации похожей задачи можно найти в главу 10.5 «Абстрактные базовые классы» из книги «А. Пол – Объектно-ориентированное программирование на C++».

2.3 Модель океана

При выполнении этого варианта студент должен будет разработать игровую модель океанического мира. Собственно мир, в котором живут обитатели океана, представляется прямоугольным полем произвольного размера. Прямоугольное поле разбито на равные квадраты (клетки), и в каждой клетке может находиться максимум четверо обитателей. Моделирование жизни в таком мире происходит в пошаговом режиме, при котором каждый шаг содержит следующие четко выделенные стадии:

- Передвижение
- Питание
- Размножение
- Умирание от старости или от голода

Способы поведения обитателей в каждой из стадий будут зависеть от типа обитателя. Студенту предлагается два основных типа обитателей со следующими свойствами:

- Планктон:
  - в клетке может находиться только одно растение;

  - передвигается;
  - имеет пункты жизни (как в ролевых играх):
    * теряет некоторое количество пунктов жизни каждый ход (эффект умирания);
    * если не остается пунктов жизни, то планктон умирает;
  - не питается;
  - размножается в текущую клетку или на соседние клетки:
    * не нужна парная особь для размножения;
    * планктон может размножаться на соседнюю клетку даже, если там уже есть другой планктон с неполным процентом жизни (в этом случае размножение обновит жизнь до 100

- Хищник:
  - передвигается (количество клеток, на которое может передвинуться, зависит от вида обитателя);
  - имеет размер (зависит от вида обитателя):
    * более крупному хищнику надо больше есть;
    * более крупный хищник может съесть более крупное травоядное;
  - имеет срок жизни;
  - может умереть, если голодно определенное количество ходов;
  - питается планктонов или другим хищником (зависит от гастрономических пристрастий обитателя океана):
    * жертва должна находиться на той же клетке;
    * всегда съедает целого обитателя, если он не убежал;
  - размножение:
    * имеет пол;
    * нужна парная особь для размножения на той же клетке;
    * детеныш рождается сразу на ту же клетку.

Изучив и уточнив приведенные правила, студенту необходимо разработать объектную модель предметной области и программу, которая будет осуществлять моделирование процесса жизни. Причем обитателями мира будут конкретные виды обитателей. Например, акула – хищник среднего размера, который может передвигаться на 2 клетки за ход, не питается планктоном и китами. Как видно из примера, конкретный вид обитателя может иметь какие-то специальные качества и какое-то специальное поведение. К планктону это не относится, потому что в модели должен быть только один вид планктона. При разработке иерархии классов обязательно должен быть применен такой атрибут любого объектно-ориентированного языка, как наследование.

Разработанная объектная модель должна соответствовать следующим требованиям:

- включать классы видов обитателей океана (количество на усмотрение разработчика)
- классы обитателей должны составлять иерархию, чтобы общие свойства и логика работы описывалась надклассом
- включать классы описания океанического мира (игрового поля)

- предусматривать механизм моделирования океанического мира
- включать классы, которые обеспечивает разгрузку модели из текстового файла

Реализованная на основе разработанной модели программа должна:

- получать имя входного файла с описанием мира и описание режима работы (автоматическое моделирование или пошаговое моделирование с ожиданием подтверждения от пользователя) через командный аргумент;
- выводить в консоль состояние игрового поля на каждом ходу.

Совет
Пример реализации похожей задачи можно найти в главу 10.5 «Абстрактные базовые классы» из книги «А. Пол – Объектно-ориентированное программирование на C++».

2.4 Модель железной дороги

Модель железной дороги представляет ориентированный взвешенный граф, вершинами которого являются станции, а дугами - железные дороги, которые имеют строгое направление и протяженность (отражается весами). Так же вершинами могут быть железнодорожные переезды.

Моделирование происходит пошагово. Поезда ездят по специально заданному маршруту. Звенья цепи маршрута могут предусматривать следующие действия при прохождении через станцию:

- загрузка
- разгрузка
- транзит
- временная стоянка

Поезд состоит из локомотива и вагонов. Локомотивы могут отличаться друг от друга по тяговой силе или скорости и сроку службы. Чем больше к локомотиву прицеплено вагонов и чем больше они загружены, тем меньше скорость локомотива. Чем больше скорость локомотива и чем меньше вес дуги, тем быстрее поезд проходит от начала дуги до ее конца. Вагоны поезда бывают товарные и пассажирские, в то время, как станции бывают трех видов:

- пассажирские (пассажирские выгоны загружаются быстро за фиксированный срок)
- товарные (загружаются товарные вагоны)
- пассажирско-товарные (загружаются и товарные, и пассажирские вагоны)

Изначально на станции имеется какое-то начальное количество товарного или пассажирского ресурса и со временем оно обновляется. Объектная модель должна соответствовать следующим требованиям:

- включать классы видов поездов, вагонов и железнодорожных станций (количество на усмотрение разработчика)
- включать классы описания железной дороги (игрового поля)

- предусматривать механизм моделирования передвижения, загрузки и разгрузки поездов
- включать классы, которые обеспечивает разгрузку модели из текстового файла

Реализованная на основе разработанной модели программа должна:

- получать имя входного файла с описанием мира и описание режима работы (автоматическое моделирование или пошаговое моделирование с ожиданием подтверждения от пользователя) через аргументы командной строки
- выводить в консоль описания произошедших событий

Файл входных данных должен содержать:

- перечисление вершин графа железной дороги с указанием типа этого узла
- перечисление связей между вершинами графа железной дороги
- начальное положение, тип и перечисление вагонов поездов
- задание маршрутов движения для поездов

2.5 Стэковая машина для обработки целых чисел

Абстрактная машина на основе стэка производит все операции через стэк. В стэке могут храниться только целые числа. Приведем пример деления числа 9 (первый аргумент) на число 4 (второй аргумент):

```
# Так задается однострочный комментарий
push 9 # Кладем первый аргумент в стэк.

push 4 # Кладем второй аргумент в стэк.

div    # Этот оператор извлекает из стэка два аргумента, делит первый
# аргумент на второй  и результат заносит в стэк.
```

Обратите, пожалуйста, внимание, что в приведенном выше примере первым считается аргумент, который в стэке находится глубже. В тексте программы для такой стэковой машины могут встречаться следующие конструкции:

- Объявление функции (функция с именем main считается начальной для программы):

  ```
  function func {
  ...
  }
  ```

- Объявление метки (метка локальна в рамках функции, т.е. в разных функциях может быть метка с одним и тем же именем):

  ```
  function func {
  ...
  label: # метка с именем label
  }
  ```

- Использование оператора (оператор может иметь в качестве аргументов целые числа, имена функций и имена меток). Каждый оператор должен быть на новой строке, а перед оператором может быть имя метки:

```
function func {
label: push 1   # Целое число в качесте аргумента
     pop  var # Имя переменной в качестве аргумента, переменную не
    # надо никак специально объявлять
...
goto label # А вот метка в качетсве аргумента
}
```

В тексте программы, как пробелы, так и знаки табуляции используются для разделения различных конструкций языка.

А теперь рассмотрим операторы, которые должна поддерживать абстрактная машина:

- Положить константное целое число или значение переменной на вершину стэка:

```
push 4
push i # i - это переменная
```

- Достать извлечь вершину стэка и установить ее в качестве значения переменной, если ее имя передано:

```
pop
pop i # i - это переменная
```

- Занести вершину стэка в качестве значения переменной:

```
peek i # i - это переменная
```

- Продублировать число на вершине стэка (т.е. положить в стэк число с вершины стэка):

```
dup
```

- Извлечь два числа из стэка, сложить их и сумму положить в стэк:

```
add
```

- Извлечь два числа из стэка, отнять от первого аргумента второй и разность положить в стэк:

```
sub
```

- Извлечь два числа из стэка, разделить первый аргумент на второй и частное положить в стэк:

```
div
```

- Извлечь два числа из стэка, перемножить их и произведение положить в стэк:

```
mul
```

- Безусловный переход по метке:

```
goto label # label - имя метки
...
label:     # Определение метки. Она локальна в рамках функции.
```

- Извлечь два числа из стэка, сравнить их и перейти по метке, если эти два числа равны:

```
ifeq label # label - имя метки
...
label:     # Определение метки. Она локальна в рамках функции.
```

- Извлечь два числа из стэка и перейти по метке, если первый аргумент больше второго:

```
ifgr label # label - имя метки
...
label:     # Определение метки. Она локальна в рамках функции.
```

- Вызвать внутреннюю функцию по имени (аргументы в функцию передаются через стэк):

```
call func  # func - имя внутренней функции
...
# Определение внутренней функции.
function func {
...
}
```

- Осуществить возврат из вызванной функции:

```
return
```

- Вызвать внешнюю функцию (написанную на C++) по имени (аргументы в функцию передаются через стэк):

```
callext func  # func - имя внешней функции
```

Рассмотрим пример входной программы вычисления факториала для нашей абстрактной стэковой машины:

```
# Функция, с которой начинается выполнение программы
function main
{
  # оператор push кладет на вершину стэка целое число
  # или значение переменной
  push 3

  # Вызов функции factorial с параметром 3
  call factorial

  # Вызов внешней функции вывода на консоль числа,
  # которое находится на вершине стэка. Вершина стэка извлекается.
  callext print

  # Выход из функции
  return
}


# Объявление функции вычисления факториала.
# Можно объявлять функции ниже их места использования.
function factorial
{
  # Продублируем первый аргумент программы (аргумент n)
  dup
  push 0

  # Проверим равна ли копия n нулю, если да,
  # тогда переходим к метке возврата факториала для нуля.
  ifeq zero_factorial

  # Аргумент не равен нулю, значит продолжаем вычисление факториала.

  # Создадим копию n и уменьшим ее на единицу - это n-1
  dup
  push 1
  sub

  # Найдем factorial(n-1)
  call factorial

  # Найдем произведение n * factorial(n-1)
  mul

  # Выходим из функции, а на вершине стэка лежит n * factorial(n-1)
  return

zero_factorial:
  pop
  push 1
```

```
  # Возвращаем управление в вызывающую функцию
  return
}
```

При выборе данного варианта, студенту необходимо разработать объектную модель такой абстрактной машины. Разработанная объектная модель должна соответствовать следующим требованиям:

- включать классы для описания всех элементов программы (программа, функция, текущий контекст значений переменных и др.);
- включать иерархию классов для операторов абстрактной машины, причем общая функциональность должна быть вынесена в надкласс;
- включать иерархию классов исключений (если в процессе интерпретации программы возникает ошибочная ситуация, то должно выбрасываться исключение).

Реализованная программа должна:

- должна быть протестирована при помощи набора unit-тестов;
- получать имя входного файла с интерпретируемой программой через аргумент командной строки;
- если указан аргумент командной строки –debug, то в пошаговом режиме выводить в консоль лог исполнения каждого оператора и состояние стэка;
- после окончания выполнения на консоль должен быть выведен стэк, если он не пуст.

2.6 Модель XML-документа

При выборе этого варианта студенту необходимо разработать объектную модель для представления XML-документа. Для знакомства с XML рекомендуется прочитать главы 1 и 2 из книги Э. Рея «Изучаем XML». Глава 1 из этой книги является обзорной и прояснит назначение XML, а в граве 2 описан синтаксис XML-разметки.

Разработанная объектная модель должна обеспечивать представление XML-документа в виде дерева и поддерживать следующие понятия XML-разметки:

- документ
- инструкция обработки
- тэг
- комментарий
- CDATA
- атрибут

Разработанная объектная модель должна соответствовать следующим требованиям:

- включать классы представления XML-документа;
- обеспечивать загрузку документа из потока ввода и сохранение XML-документа в поток вывода.

Реализованная на основе разработанной модели программа должна:

- получать имя входного xml-файла, производить его разбор с формированием XML-документа в виде композиции объектов и выводить объекты на консоль.

  Пример входного файла:

```
<!-- Особый вид инструкции обработки -->
<?xml version="1.0" ?>

<bookstore> <!-- Корневой тэг -->

  <book> <!-- Вложенный тэг -->

     <!-- Тег с атрибутом -->
     <title lang="ru">Гарри Поттер</title>

     <!-- Текст -->
     Это книга про волшебника Гарри Поттера.

     <price>15000</price>

     <!-- Пустой тег -->
     <count />
  </book>

  <book>
    <title lang="ru">Изучаем XML</title>
    <price>20000</price>
    <count>5</count>
  </book>
</bookstore>
```

2.7 Модель XML-документа с возможностью обработки поисковых запросов

Этот вариант является усложненной версией предыдущего варианта. Усложнение достигается через добавления возможности обрабатывать поисковые запросы. В качестве языка поисковых запросов будет использоваться подмножество языка XPath (Э. Рэй «Изучаем XML», глава «Отбор узлов», с. 238).

В нашем поисковом языке путь всегда абсолютный, т.е. начинается с /. Примеры поисковых запросов:

- `/` - найдет список из одного узла, который является корневым для данного xml-документа
- `/bookstore` - найдет список из одного узла, который имеет имя bookstore
- `/bookstore/book/title[attr(‘‘lang’’, ‘‘ru’’)]` - найдет список из двух узлов, которые имеют имя title и атрибут lang равный строке “ru”
- `/bookstore/book/title[attr(‘‘lang’’, ‘‘ru’’) && text(‘‘Изучаем XML’’)]` - найдет список из одного узла, который имеет имя title, атрибут lang равный строке “ru” и текстовое содержимое равное “Изучаем XML”

Обобщим. Ограниченная версия поддерживает только абсолютные поисковые запросы и предикаты, причем предикаты могут быть комплексные. В области предиката могут использоваться следующие логические связки:

- `&&` - логическое И (бинарная связка). Например, `attr(‘‘lang’’, "ru") && text("Изучаем XML")`.
- `||` - логическое ИЛИ (бинарная связка). Например, `attr(‘‘lang’’, "ru") || text("Изучаем XML")`
- `!` - логическое отрицание (унарная связка). Например, `!attr(‘‘lang’’, "ru")`.

Так же в области предиката могут использоваться скобки (). Для упрощения задачи разбора таких выражений рекомендуется считать, что связки `&&` и `||` имеют одинаковый приоритет.

Объектная модель должна соответствовать следующим требованиям:

- включать классы представления XML-документа
- иерархию классов для представления разобранного поискового запроса и интерпретации этого поискового запроса

Реализованная на основе разработанной модели программа должна:

- получать имя входного xml-файла и строку поискового запроса через командную строку и выводить на консоль результаты поиска.

2.8 Древовидное представление и интерпретация теоретико-множественных выражений

При выборе этого варианта студенту необходимо разработать объектную модель для представления теоретико-множественных выражений в виде дерева и реализовать программу-калькулятор, которая осуществляет вычисления выражений такого типа.

Рассмотрение этого варианта начнем с примера работы программы (бирюзовым цветом выделен ввод от пользователя, черным цветом – вывод программы, символ «?» обозначает нажатие клавиши Enter):

```
?   {a, b, c} ?
= {a, b, c}
?   {a, b, c} + {a, b, d} ?
= {a, b, c, d}
?   {a, b, c} * {a, b, d} ?
= {a, b}
?   B = A = {a, b, c} ?
= {a, b, c}
?   B - A ?
= {}
?   {a} + {} * ( ( {} + {a, {a, b}} ) ) ?
= {a, {a, b}}
?   Boolean( {a, b} ) ?
= {{}, {a}, {b}, {a, b}}
?   Boolean( {a, b}, {a} ) ?
^
Error: Function ‘Boolean’ expects 1 argument, but you pass 2.
?   {a} + ?
        ^
Error: Expect variable name, function call or set definition.
```

Из приведенной сессии работы можно сформулировать возможности калькулятора:

- Вычисление теоретико-множественных выражений с использованием операций объединения (+), пересечения (*) и симметрической разности (-) множеств. Причем приоритет этих операций должен задаваться при помощи скобок.
- Возможность использования заранее определенных функций обработки множество (например, функции Boolean с одним аргументом).
- Возможность установки результата вычисления выражения в качестве значения переменных.
- Указания места ошибки и сообщения об ошибке при некорректном вводе.

Для более формального описания синтаксиса выражений воспользуемся Расширенной формой Бэкуса-Наура (РБНФ). Грамматика для наших выражений будет иметь следующий вид:

```
<предложение>    ::= [ <присвоение> ] <выражение>

<присвоение>     ::= <имя_переменной> '='

<выражение>      ::= <терм_выражение> [ <операция> <выражение> ]

<терм_выражение> ::= <множество>
   | <имя_переменной>
   | <вызов_функции>
   | '(' <выражение> ')'

<множество>      ::= '{' [ <элемент> [ ',' <элемент> ] ] '}'

<элемент>        ::= <имя> | <множество>

<имя_переменной> ::= <имя>

<вызов_функции>  ::= <имя> '(' [ <выражение> [ ',' <выражение> ] ] ')'

<операция>       ::= '+' | '*' | '-'

<имя>            ::= <цифра> | <буква>
    [ <буква> | '_' | <цифра> ]

<цифра>          ::= '0' | ... | '9'
<буква>          ::= 'A' | ... | 'z'
```

Разработанная объектная модель в этом варианте должна обеспечивать представление в памяти в виде дерева и вычисление описанных выше теоретико-множественных выражений. Таким образом, сформулируем более четко требования к объектной модели и реализованной программе. Объектная модель должна соответствовать следующим требованиям:

- включать классы разбора выражения из текстового представления;
- включать классы представления дерева разобранного выражения и вершин дерева различных типов;

- включать классы, обеспечивающие поддержку использования переменных в выражениях;
- включать классы для представления и регистрации функции на C++;
- включать классы для интерпретации выражений; включать классы исключений для сигнализации об ошибках.

Реализованная на основе разработанной модели программа должна:

- проводить диалог с пользователем, в процессе которого происходит вычисление введенных выражений;
- выводить сообщения об ошибках при получении некорректного ввода от пользователя.

Совет

По рекомендации преподавателя можно изучить главу 2 из книги «А. Ахо, Р. Сети, Дж. Ульман – Компиляторы: Принципы, технологии, инструменты». В этой главе можно ознакомиться с методом рекурсивного спуска для разбора арифметических выражений и примером его реализации на языке программирования C.

Если студент последует этому совету, то ему необходимо обратить внимание на то, что приведенная выше грамматика не адаптирована для метода рекурсивного спуска.

Упрощение

По рекомендации преподавателя студент может выполнить упрощенную версию данного вариант. В упрощенной версии исключается возможность использования функций обработки во введенных выражениях.

2.9 Древовидное представление и интерпретация арифметических выражений

Модель представления и интерпретации арифметических выражений - программа калькулятор с представлением арифметического выражения в виде дерева. Должна поддерживать операции +, -, *, / и унарный минус (например, -x, -(5+x)), переменные и функции (sin, cos, sqrt, pow и т.д.). Алгоритм синтаксического разбора и построения дерева выражения рекомендуется разрабатывать на основе рекурсии (метод рекурсивного спуска).

Объектная модель должна соответствовать следующим требованиям:

- включать классы представления дерева арифметического выражения и вершин дерева различных типов
- интерфейс регистрации и исполнения внешних функций (например, sin, cos, sqrt и т.д.)
- классы, обеспечивающие поддержку переменных в арифметическом выражении
- классы, обеспечивающие интерпретацию арифметического выражения

Реализованная на основе разработанной модели программа должна:

- получать входной арифметическое выражение и значения переменных из командной строки и выводить на консоль вычисленное значение выражения

При выборе этого варианта студенту необходимо разработать объектную модель для представления арифметических выражений в виде дерева и реализовать программу-калькулятор, которая осуществляет вычисления выражений такого типа.

Рассмотрение этого варианта начнем с примера работы программы (бирюзовым цветом выделен ввод от пользователя, черным цветом – вывод программы, символ «?» обозначает нажатие клавиши Enter):

```
> 1
= 1
> 1 + -2
= -1
> ( ( 1 * ( -2 ) ) )
= 2
> B = A = 2
= 2
> B - A
= 0
> 1 + 2 * ( 10 / 2 + pow(3, 2) )
= 29
> pow( 2, 3 )
= 8
> pow( 2 )
   ^
Error: Function ‘Boolean’ expects 2 arguments, but you pass 1.
> 1 +
      ^
Error: Expect variable name, function call or number.
```

Из приведенной сессии работы можно сформулировать возможности калькулятора:

- Вычисление арифметических выражений с использованием операций сложения (+), умножения (*), разности (-), деления (/). Причем эти операции должны иметь приоритет.
- Возможность использования заранее определенных функций обработки чисел (например, функции pow с двумя аргументами для вычисления степени числа).
- Возможность установки результата вычисления выражения в качестве значения переменных.
- Указания места ошибки и сообщения об ошибке при некорректном вводе.

Для более формального описания синтаксиса выражений воспользуемся Расширенной формой Бэкуса-Наура (РБНФ). Грамматика для наших выражений будет иметь следующий вид:

```
<предложение>    ::= [ <присвоение> ] <выражение>

<присвоение>     ::= <имя_переменной> ‘=’

<выражение>      ::= <терм_выражение> [ <операция> <выражение> ]

<терм_выражение> ::= [ ‘-‘ ]
  ( <число>
   | <имя_переменной>
   | <вызов_функции>
   | ‘(’ <выражение> ‘)’
   )

<число>          ::= { <цифра> } ‘.’ [ <цифра> ]
                 | [ <цифра> ] ‘.’ { <цифра> }
```

```
<имя_переменной> ::= <имя>

<вызов_функции>  ::= <имя> ‘(’ [ <выражение> [ ‘,’ <выражение> ] ] ‘)’

<операция>       ::= ‘+’ | ‘*’ | ‘-’ | ‘/’

<имя>            ::= <буква> [ <буква> | ‘_’ | <цифра> ]

<цифра>          ::= ‘0’ | ... | ‘9’
<буква>          ::= ‘A’ | ... | ‘z’
```

Разработанная объектная модель в этом варианте должна обеспечивать представление в памяти в виде дерева и вычисление описанных выше арифметических выражений. Таким образом, сформулируем более четко требования к объектной модели и реализованной программе.

Объектная модель должна соответствовать следующим требованиям:

- включать классы разбора выражения из текстового представления;
- включать классы представления дерева разобранного выражения и вершин дерева различных типов;
- включать классы, обеспечивающие поддержку использования переменных в выражениях;
- включать классы для представления и регистрации функции на C++;
- включать классы для интерпретации выражений;
- включать классы исключений для сигнализации об ошибках.

Реализованная на основе разработанной модели программа должна:

- проводить диалог с пользователем, в процессе которого происходит вычисление введенных выражений;
- выводить сообщения об ошибках при получении некорректного ввода от пользователя.

Совет

По рекомендации преподавателя можно изучить главу 2 из книги «А. Ахо, Р. Сети, Дж. Ульман - Компиляторы Принципы, технологии, инструменты». В этой главе можно ознакомиться с методом рекурсивного спуска для разбора арифметических выражений и примером его реализации на языке программирования C.

Если студент последует этому совету, то ему необходимо обратить внимание на то, что приведенная выше грамматика не адаптирована для метода рекурсивного спуска.

Упрощение

По рекомендации преподавателя студент может выполнить упрощенную версию данного вариант. Упрощение может происходить по следующим направлениям:

- исключить приоритет операторов
- исключить унарный минус
- исключить возможность регистрации и вызова пользовательских функций