МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное
учреждение высшего образования
«Южный федеральный университет»

Институт математики, механики
и компьютерных наук им. И. И. Воровича

Кафедра алгебры и дискретной математики

Березенцев Данил Андреевич

РАЗРАБОТКА ОБУЧАЮЩЕЙ СИСТЕМЫ ПО РЕГУЛЯРНЫМ ВЫРАЖЕНИЯМ

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
по направлению подготовки
02.03.02 – Фундаментальная информатика и информационные технологии

Научный руководитель –
доц., к. ф.-м. н. Абрамян Михаил Эдуардович

Допущено к защите:
руководитель
направления_____ Михалкович С.С.

Ростов-на-Дону – 2024

Оглавление

Постановка задачи 3
Введение 4
1. Обзор библиотек для регулярных выражений 6
1.1. Регулярные выражения в C# 6
1.2. Регулярные выражения в C++ 11
1.3. Регулярные выражения в Python 16
1.4. Общее описание языка регулярных выражений 21
1.5. Сравнительное описание библиотек 25
2. Группы заданий 27
2.1.Общее описание разработки групп заданий 27
2.2.Примеры реализации заданий с применением конструктора 33
2.3.Примеры решения заданий 36
3. Описание разработанных файлов дополнений 39
3.1.Общее описание разработки файлов дополнений 39
3.2.Примеры реализации файлов дополнений 42
3.3.Примеры решения заданий с файлом дополнения 46
Заключение 49
Литература 50

Постановка задачи

1. Для языков C#, C++ и Python подготовить подробное описание библиотек для работы с регулярными выражениями с примерами их применения; дать сравнительную характеристику возможностей библиотек для языков C#, C++, Python.

2. Разработать несколько (не менее пяти) групп заданий, связанных с различными возможностями языка регулярных выражений; каждая группа должна включать не менее пяти заданий;

3. Реализовать разработанные группы заданий в виде динамических библиотек, используя конструктор для языка C++ задачника Programming Taskbook;

4. Подготовить файлы дополнений для языков C#, C++, Python, содержащие заготовки учебных программ и дополнительные указания для различных заданий;

5. Привести примеры решений задач из разработанных групп для различных языков.

Введение

Обучение программированию представляет собой довольно трудоемкий процесс. Организовать обучающую систему, которая бы отвечала всем критериям качественного обучения таким образом, чтобы это было удобно, с одной стороны, и для самого обучающегося и, с другой стороны, для преподавателя или составителя учебных заданий, очень трудно. Причем если удобство для учеников очевидно и относительно легко оценивается, то удобство для составителей заданий часто упускается из виду и содержит множество неоднозначных моментов.

Примером образовательной системы, которая предоставляет широкие возможности для учеников и составителей заданий, является электронный задачник Programming Taskbook [8]. Он позволяет учащимся выполнять разнообразные учебные задания, используя множество языков программирования и сред разработки. В текущей версии задачника 4.24 поддерживаются языки: Pascal, PascalABC.NET, C, C++, C#, Visual Basic .NET, F#, Python, Java, Ruby и Julia. Programming Taskbook содержит 1100 учебных заданий, охватывающих все основные аспекты программирования: от базовых типов данных и операторов до сложных структур данных и рекурсивных алгоритмов.

Одним из преимуществ образовательной системы Programming Taskbook является доступность большого количества языков программирования для выполнения заданий. Также система позволяет разрабатывать новые группы заданий и модифицировать существующие, расширяя коллекцию задач по программированию. Группы заданий представляют собой динамические библиотеки и могут быть написаны на различных языках. Для создания DLL-библиотек с группами заданий используются специальные программы - конструкторы учебных заданий. Их исходный код позволяет взаимодействовать с ядром задачника Programming

Taskbook и предоставляет разработчику все необходимые функции для создания группы заданий.

В данной работе рассматривается создание групп заданий на тему регулярных выражений. Регулярные выражения — это мощный инструмент для работы с текстовыми данными. Они используются для поиска и манипулирования строками текста, позволяя осуществлять более гибкий и эффективный поиск шаблонов символов в тексте. Изучение и понимание регулярных выражений требует времени и практики, но они могут значительно упростить обработку текстовых данных и повысить производительность при работе с текстом.

1 Обзор библиотек для регулярных выражений

1.1 Регулярные выражения в C#

Класс Regex(C#)

Класс `Regex` и связанные с ним классы описаны в пространстве имен `System.Text.RegularExpressions`.

Конструктор и свойства

```
Regex(string pattern[, RegexOptions options]);
```

При создании экземпляра `Regex` регулярное выражение специальным образом обрабатывается, что в дальнейшем ускоряет его использование. Можно также использовать статические методы класса, не требующие создания экземпляра.

Для экземпляра класса `Regex` доступны два свойства (только для чтения): `Options` (типа `RegexOptions`) — набор опций поиска, переданных в конструкторе; `RightToLeft` (типа `bool`) — порядок поиска регулярного выражения (одна из опций).

Основные методы

`bool IsMatch` — найдено или нет требуемое выражение;

`Match Match` — возвращает первое найденное выражение;

`MatchCollection Matches` — возвращает все найденные выражения;

`string[] Split` — разбивает строку на фрагменты; разделители фрагментов определяются регулярным выражением;

`string Replace` — заменяет найденные выражения.

В списке параметров `input` — строка, в которой выполняется поиск, pattern (и `replacement` для `Replace`) — регулярное выражение, `start` — индекс первого символа, начиная с которого выполняется поиск, `count` — число возвращаемых фрагментов (`Split`) или число выполняемых замен (`Replace`).

Статические методы `IsMatch`, `Match`, `Matches` и `Split`:

```
(string input, string pattern[, RegexOptions options])
```

Статические методы `Replace`:

```
(string input, string pattern, MatchEvaluator evaluator[,
RegexOptions options])
(string input, string pattern, string
replacement[,RegexOptions options])
```

Экземплярные методы `IsMatch, Match и Matches`:

```
string input[, int start])
```

Экземплярные методы `Split`:

```
string input[, int count[, start]])
```

Экземплярные методы `Replace`:

```
string input, MatchEvaluator evaluator[, int count[, int
start]])
(string input, string replacement[, int count[, int start]])
```

Некоторые вспомогательные методы

`static string Escape(string str)` — возвращает вариант строки `str`, в котором экранированы все специальные символы, используемые в регулярных выражениях: \ * + ? | { [( ) ^ $. # и пробельные символы);

`static string Unescape(string str)` — восстанавливает строку, символы которой ранее были экранированы;

Вспомогательные классы

`RegexOptions` — перечисление, определяющее опции поиска. Основные члены (в скобках указывается значение этой опции, при ее задании непосредственно в регулярном выражении):

`IgnoreCase (i)` — игнорировать регистр при поиске;

`Multiline (m)` — режим многострочного текста, при котором символы ^ и $ соответствуют началу и концу каждой строчки текста (а не всей содержащей его строки типа string);

`ExplicitCapture (n)` — считать группами только те пары круглых скобок, которым явно присвоено имя или номер;

`Singleline (s)` — режим, при котором символ . (точка) соответствует любому символу (а не любому символу, кроме \n);

`IgnorePatternWhitespaces (x)` — игнорировать неэкранированные пробельные символы в регулярном выражении;

`RightToLeft (r)` — выполнять поиск справа налево;

`None` — дополнительных опций нет.

Несколько опций должны объединяться операцией |.

`Group` — класс, инкапсулирующий свойства группы регулярного выражения. Основные свойства (только для чтения):

`bool Success` — `true`, если группа найдена, иначе — `false`;

`int Index` — индекс начала найденной группы;

`int Length` — длина найденной группы;

`string Value` — значение найденной группы (если группа не найдена, то равно пустой строке).

Значение группы также возвращается методом `ToString`.

`Match` — класс (потомок `Group`), инкапсулирующий свойства найденного вхождения регулярного выражения. Имеет те же свойства, что и класс `Group`, которые в данном случае относятся не к группе, а ко всему найденному вхождению. Кроме того, имеет свойство только для чтения `Groups` типа `GroupCollections` — коллекцию всех групп, связанных с найденным вхождением; первый элемент этой коллекции (с индексом 0) соответствует нулевой группе, т. е. всему найденному вхождению.

Метод `NextMatch()` типа `Match` позволяет получить следующее найденное вхождение (если оно отсутствует, то свойство `Success` возвращенного объекта `Match` будет равно `false`).

`GroupCollection` — класс-коллекция групп, реализующий интерфейсы `ICollection` и `IEnumerable`. Имеет свойство `Count` типа `int` — количество групп (только для чтения) и два индексатора типа `Group` (только для чтения) с числовым или строковым параметром — номером (от 0) или именем группы.

`MatchCollection` — класс-коллекция найденных вхождений, реализующий интерфейсы `ICollection` и `IEnumerable`. Имеет свойство `Count` типа `int` — количество найденных (только для чтения) и индексатор типа `Match` (только для чтения) с числовым параметром — номером найденного вхождения (от 0).

`MatchEvaluator` — делегат с сигнатурой `string (Match match)`, используемый в методе `Replace` и определяющий строку, на которую надо заменить найденное вхождение `match`.

Группирование и ссылки на группы

`(expr)` — включить соответствие для выражения expr в нумерованную группу (группы нумеруются от 1 в соответствии с порядком следования их открывающих скобок; группа 0 соответствует всему найденному вхождению);

`(?<name>expr)` или `(?'name'expr)` — включить соответствие для выражения expr в именованную группу с именем name;

`(?:expr)` — группирующее выражение, не связываемое с нумерованной или именованной группой;

`\N` — ссылка на ранее найденную группу с номером `N`;

`\k<name>` — ссылка на группу с именем name.

Пример. Чтобы выделить в регулярном выражении для телефонного номера `\d{3}-\d{3}-\d{4}` начальную группу из трех цифр (код региона) и завершающую группу из 7 цифр (собственно телефонный номер), достаточно заключить их в круглые скобки и воспользоваться свойством Groups:

```
Match m = Regex.Match("123-456-7890", @"(\d{3})-(\d{3}-\d{4})");
Console.WriteLine(m.Groups[0]); // 123-456-7890
Console.WriteLine(m.Groups[1]); // 123
Console.WriteLine(m.Groups[2]); // 456-7890
```

Найденные группы можно использовать при продолжении поиска. В следующем примере ищется слово, начинающееся и оканчивающееся на одну и ту же букву:

```
Regex.Match("push pop peek", @"\b(\w)\w*\1\b") // pop
```

Комментарии

`(?#comment)` — комментарий;

`#comment` — комментарий до конца строки (только в режиме `IgnorePatternWhitespaces`).

Некоторые подстановки в выражениях замены

`$$` — символ `$`;

`$0` — все найденное вхождение;

`$_` — вся исходная строка;

`$N` — найденная группа с номером `N` (или пустая строка, если группа не найдена);

`${name}` — найденная группа с именем name (или пустая строка, если группа не найдена).

Пример (заключаем все числа в угловые скобки):

```
Regex.Replace("10+2=12", @"\d+", "<$0>") // <10>+<2>=<12>
```

В случае сложных видов замены удобнее использовать вариант метода `Replace` с параметром-делегатом `MatchEvaluator`.

Пример (удваиваем все найденные числа):

```
Regex.Replace("10+2=12", @"\d+",
              m => (int.Parse(m.Value)*2).ToString()) // 20+4=24
```

Опции поиска

`(?opt)` — в качестве opt указывается буква соответствующей опции (см. описание класса `RegexOptions`). Любая опция, кроме `(?r)`, может указываться в любом месте регулярного выражения и впоследствии может быть отменена директивой `(?-opt)`. Опция `(?r)` должна быть указана в начале регулярного выражения и не может быть отменена. Опции можно объединять: `(?i-ms)` — опция i включена, опции m и s отключены.

Примеры:

```
Regex.Match("a", "A", RegexOptions.IgnoreCase) // a
Regex.Match("a", "(?i)A") // a
Regex.Match("BaAaAab", "(?i)A+") // aAaAa
Regex.Match("BaAAaab", "(?i)a(?-i)a") // Aa
```

1.2 Регулярные выражения в C++

Класс regex(C++)

При подготовке данного раздела были использованы материалы, приведенные в [3-7]. Сайты с документацией[3-6] помогли ознакомиться с основными и вспомогательными классами класса regex, а также сайт с документацией, созданной участниками Stack Overflow[7], в котором подробно расписаны методы и приведены примеры работы с ними.

В C++ стандартная библиотека regex предоставляет набор классов и функций для работы с регулярными выражениями.

Конструктор и свойства

Конструкторы регулярных выражений на `C++` используются для создания объекта класса `regex`, который представляет шаблон регулярного выражения. Конструкторы имеют различные формы и параметры, которые определяют образец, который должен соответствовать строке.

Примеры конструкторов:

Этот конструктор принимает строку, содержащую регулярное выражение в виде образца pattern, и необязательный параметр `flags`, который определяет используемый синтаксис.

```
std::regex::regex(const char* pattern, syntax_option_type
flags = std::regex_constants::ECMAScript)
```

Основные методы

`bool std::regex_search` – метод ищет первое вхождение заданного шаблона в строке;

`bool std::regex_match` – метод проверяет, соответствует ли заданная строка заданному шаблону полностью;

`string std::regex_replace` – метод выполняет замену всех вхождений шаблона на заданную строку.

В списке параметров `string` – строка, в которой выполняется поиск, `pattern` – регулярное выражение, `[flags](`по умолчанию = `std::regex_constants::match_default)` – список опциональных флагов,

указывающий на то, как будут интерпретироваться символы в выражении, `repl` – строка, которая заменяет все вхождения шаблона в строке(`regex_replace`).

Статические методы `regex_search, regex_match`:

```
bool regex_search( const std::string& string, std::regex&
pattern, std::regex_constants::match_flag_type flags =
std::regex_constants::match_default );
```

Статические методы `regex_replace`:

```
std::string std::regex_replace(std::string string,
std::regex pattern, std::string replacement);
```

Вспомогательные классы

`std::match_results` – класс, представляющий результаты сопоставления регулярного выражения с текстом. В него записываются позиции соответствующих фрагментов текста. класс для представления результатов поиска. Объект этого класса содержит информацию о том, где найдено соответствие в строке, а также о самих соответствиях. Этот объект обрабатывается и заполняется функцией `std::regex_match` или `std::regex_search`.

`size_t size() const` – возвращает количество соответствий.

`bool empty() const` – возвращает `true`, если ни одного соответствия не было найдено.

`string_type str() const` – возвращает исходную строку, с которой было сопоставление.

`const_reference operator[](size_t n) const` – возвращает соответствующее строковое выражение.

`const_iterator begin() const` – возвращает итератор на начало строки.

`const_iterator end() const` – возвращает итератор на конец строки.

`std::regex_iterator` и `std::regex_token_iterator` – классы для выполнения итерации по строке с помощью регулярного выражения. `std::regex_iterator` используется для итерации по всем соответствиям в

строке, а std::regex_token_iterator – для итерации по отдельным токенам в строке.

std::regex_constants - это класс-контейнер, который определяет константы и флаги для работы с регулярными выражениями в C++. Этот класс поддерживает ряд констант для использования с методом std::regex_match, чтобы указать, какие флаги должны быть включены при сопоставлении регулярного выражения. Класс также содержит флаги для управления поведением сопоставления и замены, а также константы, используемые для ошибок синтаксиса.

Вот некоторые из наиболее важных констант класса std::regex_constants:

std::regex_constants::match_any – флаг, указывающий, что регулярное выражение должно соответствовать любой части строки.

std::regex_constants::match_not_bol – флаг, указывающий, что регулярное выражение должно не начинаться с символа начала строки (^).

std::regex_constants::match_not_eol – флаг, указывающий, что регулярное выражение должно не заканчиваться символом конца строки ($).

std::regex_constants::format_no_copy – флаг, указывающий, что заменяемая строка не должна содержать копии заменяемого текста.

std::regex_constants::error_collate – константа, указывающая ошибку синтаксиса, которая происходит при использовании не поддерживаемой последовательности символов.

std::regex_constants::error_badrepeat – константа, указывающая ошибку синтаксиса, которая происходит при использовании неверного повторения.

std::regex_constants::error_ctype – константа, указывающая ошибку синтаксиса, которая происходит при использовании неподдерживаемой категории символов.

`std::regex_constants::error_escape` – константа, указывающая ошибку синтаксиса, которая происходит при использовании неверной экранировки символов.

Эти вспомогательные классы и функции облегчают работу с регулярными выражениями в языке `C++`.

Группирование и ссылки на группы

`(expr)` – включить соответствие для выражения `expr` в нумерованную группу (группы нумеруются от 1 в соответствии с порядком следования их открывающих скобок; группа 0 соответствует всему найденному вхождению);

`(?<name>expr)` – включить соответствие для выражения `expr` в именованную группу с именем `name`. Именованная группа также является пронумерованной группой, как если бы группа не была названа;

`(\number или \k<name>)` – ссылка на нумерованную группу `number` или с именем `name`.

Комментарии

Синтаксис `(?#expr)` для написания комментариев не поддерживается всеми реализациями регулярных выражений и может привести к ошибкам. Поэтому в `C++` комментарии в регулярных выражениях начинаются со знака `//` и продолжаются до конца строки. Это единственный способ прокомментировать строку в регулярном выражении.

Пример.

```
string text = "Hello, world!";
regex reg(
     "Hello" // комментарий
     );
regex_search(text, reg); // true
```

Подстановки

Некоторые из наиболее часто используемых подстановок в выражениях замены:

`$&` – заменить на найденную подстроку;

`$1` – содержимое первой скобочной группы;

`$2` – содержимое второй скобочной группы;

`$n` – содержимое n-й скобочной группы.

Пример.

```
regex pattern(R"(\d+)");
string str = "10+2=12";
regex_replace(str, pattern, "<$&>"); // <10>+<2>=<12>
```

Опции поиска(Флаги)

`std::regex_constants::icase` — игнорирует регистр символов;

`std::regex_constants::multiline` — включает многострочный поиск;

`std::regex_constants::dotall` — позволяет точке сопоставляться с любым символом;

`std::regex_constants::extended` — игнорирование пробелов и комментариев в регулярном выражении;

`std::regex_constants::nosubs` – возвращать только булево значение, указывающее на то, найдено ли совпадение;

`std::regex_constants::optimize` – произвести оптимизацию регулярного выражения для ускорения обработки;

`std::regex_constants::collate` – использовать настройки локали для определения порядка символов при сравнении;

`std::regex_constants::ECMAScript` – использует синтаксис `ECMAScript`;

`std::regex_constants::basic` – использует базовый синтаксис регулярных выражений;

`std::regex_constants::awk` – использует синтаксис `AWK`;

`std::regex_constants::grep` – использует синтаксис утилиты `grep`.

Пример.

```
string text = "Hello, World!";
regex pattern("hello", std::regex_constants::icase);
regex_search(text, pattern); // true
```

1.3 Регулярные выражения в Python

Модуль re(Python)

При подготовке данного раздела была использованы материалы, приведенные в [1].

В Python, регулярные выражения доступны в библиотеке `re`.

`re` – это модуль Python, содержащий методы для работы с регулярными выражениями.

```
import re
```

Конструктор и свойства

Для создания регулярного выражения необходимо использовать функцию `re.compile(pattern, [flags])`, которая принимает на вход строку с регулярным выражением и опциональный флаг, указывающий на то, как будут интерпретироваться символы в выражении. Можно также использовать статические методы класса, не требующие создание экземпляра.

Основные методы

`Match re.match` – метод проверяет, соответствует ли начало строки регулярному выражению;

`Match re.fullmatch` – метод проверяет, соответствует ли вся строка регулярному выражению;

`Match re.search` – метод ищет первую подстроку в строке, которая соответствует регулярному выражению;

`list<str> re.findall` – метод ищет все подстроки в строке, которые соответствуют регулярному выражению;

`callable_iterator re.finditer` – метод возвращает итератор, выдающий объекты соответствия по всем неперекрывающимся совпадениям для повторного шаблона в строке;

`str re.sub` – метод заменяет все вхождения регулярного выражения в строке на заданную замену;

`list<str> re.split` – метод разделяет строку на подстроки, используя регулярное выражение в качестве разделителя;

`Pattern re.compile` – метод создает объект-регулярное выражение, который может использоваться для поиска соответствий в строках;

`str re.group()` – метод возвращает последнее найденное совпадение шаблона;

`str re.escape(string)` – метод экранирует все специальные символы в строке, чтобы их можно было использовать в качестве обычных символов в регулярных выражениях;

`re.purge()` – метод очищает кэш регулярных выражений.

В списке параметров `string` – строка, в которой выполняется поиск, `pattern` – регулярное выражение, `[flags]`(по умолчанию = 0) – список опциональных флагов, указывающий на то, как будут интерпретироваться символы в выражении, `repl` – строка, которая заменяет все вхождения шаблона в строке(`re.sub`), count(по умолчанию = 0) – это максимальное количество замен шаблона(`re.sub`), `maxsplit`(по умолчанию = 0) – количество разбиений строки(`re.split`).

Статические методы `re.match, re.fullmatch, re.search, re.findall, re.finditer`:

```
(pattern, string, flags=0)
```

Статические методы `re.sub`:

```
(pattern, repl, string, count=0, flags=0)
```

Статические методы `re.split`:

```
(pattern, string, maxsplit=0, flags=0)
```

Статические методы `re.escape`:

```
(string)
```

Статические методы `re.compile`:

```
(pattern, flags=0)
```

Вспомогательные классы

Модуль `re` предоставляет различные вспомогательные классы, которые помогают работать с регулярными выражениями. Некоторые из них:

`re.Match`: Объект, который возвращается, когда происходит совпадение с регулярным выражением.

`Match.expand(template)` – метод возвращает строку, полученную путём подстановки обратной косой черты в шаблоне строки `template`;

`Match.group([group1, ...])` – метод возвращает одну или несколько подгрупп соответствия;

`Match.groups(default=None)` – метод возвращает кортеж, содержащий все подгруппы соответствия, от 1 до любого количества групп в шаблоне;

`Match.groupdict(default=None)` – метод возвращает словарь, содержащий все *именованные* подгруппы соответствия, обозначенные именем подгруппы;

`Match.start([group])`, `Match.end([group])` – метод возвращает индексы начала и конца подстроки, которым соответствует *`group`*; по умолчанию для *`group`* равно нулю (что означает всю подобранную подстроку);

`Match.span([group])` – метод для *m* совпадений возвращает 2 кортежа `(m.start(group), m.end(group));`

`Match.pos` – метод возвращает значение *`pos`*, которое было передано методу `search()` или `match()` объекта регулярного выражения. Это индекс в строке, с которой механизм `RE` начал поиск совпадения;

`Match.endpos` – метод возвращает значение *`endpos`*, которое было передано методу `search()` или `match()` объекта регулярного выражения. Это индекс в строке, за который движок `RE` не пойдет;

`Match.lastindex` – метод возвращает целочисленный индекс последней сопоставленной группы захвата, или `None` если ни одна группа не была сопоставлена вообще;

`Match.lastgroup` – метод возвращает имя последней сопоставленной группы захвата, или `None` если у группы не было имени, или если ни одна группа не была сопоставлена вообще;

`Match.re` – метод возвращает объект регулярного выражения, чей метод `match()` или `search()` создал этот экземпляр соответствия;

`Match.string` – метод возвращает строку, переданную в `match()` или `search()`;

`re.RegexObject`: Объект, который представляет скомпилированное регулярное выражение. Используется для многократного использования одного и того же выражения в разных строках.

`re.Scanner`: Класс для сканирования строки на основе определенного набора правил регулярных выражений.

`re.Pattern`: Это новый класс, добавленный в Python 3.7. Он является альтернативой классу `re.RegexObject` и использует новый способ компиляции регулярных выражений. Скомпилированные объекты регулярного выражения поддерживают следующие методы: `search()`, `match()`, `fullmatch()`, `findall()`, `finditer()`, `sub()`.

В списке параметров `string` – строка, в которой выполняется поиск, а также необязательные параметры `pos` и `endpos`. Необязательный второй параметр `pos`(по умолчанию = 0) указывает индекс в строке, с которой должен начинаться поиск; по умолчанию он равен 0. Необязательный параметр `endpos`(по умолчанию = 0) ограничивает расстояние поиска в строке.

Экземплярные методы `search()`, `match()`, `fullmatch()`, `findall()`, `finditer()`, `sub()`:

(*string,pos=0, endpos=0*)

Некоторые вспомогательные методы:

`Pattern.flags` – метод возвращает флаги соответствия регулярным выражениям.

`Pattern.groups` – метод возвращает количество групп захвата в шаблоне.

`Pattern.groupindex` – метод возвращает словарь, сопоставляющий любые символьные имена групп, определенные `(?P<id>)`, с номерами групп. Словарь пуст, если в шаблоне не использовались символьные группы.

`Pattern.pattern` – метод возвращает строку шаблона, из которой был скомпилирован объект шаблона.

`re.MatchIterator`: Класс для итерации по всем совпадениям регулярного выражения в исходной строке.

`re.MatchObject`: Базовый класс, от которого наследуются классы `re.RegexObject` и `re.MatchIterator`. Он представляет собой совпадение регулярного выражения в строке.

Все эти вспомогательные классы предназначены для удобной работы с регулярными выражениями, позволяя более гибко и удобно реализовывать различные задачи, связанные с обработкой строк.

Группирование и ссылки на группы

`(expr)` – включить соответствие для выражения `expr` в нумерованную группу (группы нумеруются от 1 в соответствии с порядком следования их открывающих скобок; группа 0 соответствует всему найденному вхождению);

`(?P<name>expr)` – включить соответствие для выражения `expr` в именованную группу с именем `name`; Именованная группа также является пронумерованной группой, как если бы группа не была названа.

`(?P=name)` – ссылка на группу с именем `name`.

Пример.

```
print(re.sub(r"(\d+)", "<\g<0>>", "10+2=12")) # <10>+<2>=<12>
```

Комментарии

`(?#expr)` – комментарий, содержимое в круглых скобок просто игнорируется.

Пример.

```
print(re.findall("a(?#комментарий)", "aaa")) # ['a', 'a',
'a']
```

Опции поиска(Флаги)

класс `re.RegexFlag`, содержащий параметры регулярных выражений, перечисленные ниже:

`re.IGNORECASE` или `re.I` - игнорирует регистр символов;

`re.MULTILINE` или `re.M` - включает многострочный поиск;

`re.DOTALL` или `re.S` - позволяет точке сопоставляться с любым символом;

`re.VERBOSE` или `re.X` - позволяет использовать многострочные выражения и комментарии, игнорирование пробелов и комментариев в регулярном выражении;

`re.UNICODE` или `re.U` - позволяет использовать Юникод-символы в регулярных выражениях;

`re.ASCII` или `re.B` - включает ASCII-режим, при котором некоторые специальные символы не распознаются;

`re.LOCALE` - определяет использование локали для разбора символов;

`re.DEBUG` - выводит отладочную информацию при сопоставлении регулярного выражения с текстом.

Эти опции можно комбинировать между собой, например: `re.IGNORECASE | re.MULTILINE | re.DOTALL`.

Пример.

```
print(re.findall("a", "Aaa", re.I)) # ['A', 'a', 'a']
a = re.compile(r"""\d +   # целая часть
               \.     # десятичная точка
               \d*   # некоторые дробные цифры """, re.X)
    print(a.findall("abc10.11def")) # ['10.11']
```

1.4 Общее описание языка регулярных выражений

Язык регулярных выражений

При подготовке данного раздела была использованы материалы, приведенные в [2].

Некоторые специальные символы

`\t` — табуляция;

\r — возврат каретки;

\f — новая страница;

\n — новая строка;

\xNN — ASCII-символ в 16-ричной системе счисления;

\uNNNN — Unicode-символ в 16-ричной системе счисления.

Множества символов

[abcd] — один из символов в списке (отрицание: [^abcd]);

[a-d] — один из символов в диапазоне (отрицание: [^a-d]);

\d — десятичная цифра, т. е. [0-9] (отрицание: \D);

\w — словообразующий символ, например, для английского языка — [a-zA-Z_0-9] (отрицание: \W);

\s — пробельный символ, т. е. [\t\r\f\n] (отрицание: \S);

. — любой символ, кроме \n (в режиме SingleLine — любой символ).

Символы, указываемые в квадратных скобках, не должны экранироваться (за исключением символа]).

Квантификаторы

* — 0 или более соответствий;

+ — 1 или более соответствий;

? — 0 или 1 соответствие;

{N} — точно N соответствий;

{N,} — не менее N соответствий;

{N,M} — от N до M соответствий.

Примеры.

Ищется имя файла cv.doc, возможно, снабженное нумерацией. Обратите внимание на экранирование точки и на использование «буквального» режима в регулярном выражении. В C# буквальный режим в регулярных выражениях указывается с помощью префикса `@"expr"`, в C++ с помощью префикса `R"(expr)"`, а в Python с помощью префикса `r"expr"`, где внутри кавычек указывается само регулярное выражение:

Пример(C#).

```
Regex.IsMatch("cv12.doc", @"cv\d*\.doc") // true
```

Пример(C++).

```
string str = "cv12.doc";
regex r(R"(cv\d*\.doc)");
regex_search(str, r); // true
```

Пример(Python).

```
re.search(r"cv\d*\.doc", "cv12.doc") // cv12.doc
```

Директивы нулевой длины

`^` — начало строки (в режиме `Multiline` — начало любой строчки многострочного текста);

`$` — конец строки (в режиме `Multiline` — конец любой строчки многострочного текста);

`\A` — начало строки (в любом режиме);

`\z` — конец строки (в любом режиме);

`\Z` — конец строки или строчки многострочного текста;

`\b` — позиция на границе слова;

`\B` — позиция не на границе (т. е. внутри) слова;

`(?=expr)` — продолжать поиск, если для выражения `expr` есть соответствие справа (положительный просмотр вперед);

`(?!expr)` — продолжать поиск, если для выражения `expr` нет соответствия справа (отрицательный просмотр вперед);

`(?<=expr)` — продолжать поиск, если для выражения `expr` есть соответствие слева (положительный просмотр назад);

`(?<!expr)` — продолжать поиск, если для выражения `expr` нет соответствия слева (отрицательный просмотр назад);

Примеры(C#).

```
Regex.Match("zz<i>A<i>B</i>C</i>zz", @".*(?=</i>)")  //
zz<i>A<i>B</i>C
Regex.Match("zz<i>A<i>B</i>C</i>zz", @"(?<=<i>).*(?=</i>)")
// A<i>B</i>C
```

Примеры(C++).

```
string str = "zz<i>A<i>B</i>C</i>zz";
regex r(".*(?=</i>)");
smatch match;
regex_search(str, match, r);
match.str(0) // zz<i>A<i>B</i>C
```

Примеры(Python).

```
re.search(".*(?=</i>)",'zz<i>A<i>B</i>C</i>zz') //
zz<i>A<i>B</i>C
re.search("(?<=<i>).*(?=</i>)",'zz<i>A<i>B</i>C</i>zz') //
A<i>B</i>C
```

Примеры(C#).

Подсчет числа пустых строк в исходном тексте text (квантификатор ? нужен для того, чтобы распознать последнюю пустую строку, после которой может отсутствовать символ \r):

```
Regex.Matches(text, @"^\r?$", RegexOptions.Multiline).Count
```

Границей слова считается позиция, в которой словообразующий символ \w соседствует либо с началом/концом строки, либо с символом, который не является словообразующим (\W).

Пример (поиск отдельных слов в строке s):

```
foreach (Match m in Regex.Matches("  aa   ss cc ",
@"\b\w+\b"))
Console.Write("("+m+")"); // (aa)(ss)(cc)
```

Альтернативные варианты

`a|b` — подходит один из указанных вариантов; при прочих равных условиях предпочтение отдается левому варианту. Можно указывать более двух операндов.

Примеры(C#).

```
Regex.Matches("10", "1|10") // 1
Regex.Matches("10", "10|1") // 10
Regex.Matches("10", "0|1|10") // 1 и 0
```

Примеры(C++).

```
string str = "10";
regex r("1|10");
smatch match;
regex_search(str, match, r);
cout << match.str(0) << endl; // 1
string str = "10";
regex r("10|1");
smatch match;
regex_search(str, match, r);
cout << match.str(0) << endl; // 10
```

Примеры(Python).

```
print(re.findall("a|b", "aaba")) # ['a', 'a', 'b', 'a']
print(re.findall("1|10", "10")) # ['1']
print(re.findall("10|1", "10")) # ['10']
```

1.5 Сравнительное описание библиотек

Таблица соответствия функций из разных библиотек, выполняющих одинаковые действия:

C#	C++	Python	Действие
–	regex_match	match	метод проверяет, соответствует ли начало строки регулярному выражению
Match	regex_search	search	метод ищет первую подстроку в строке, которая соответствует регулярному выражению
Replace	regex_replace	sub	метод выполняет замену всех вхождений шаблона на заданную строку
IsMatch	–	–	метод возвращает найдено или нет требуемое выражение
Matches	–	findall	метод возвращает все найденные выражения

Split	–	split	метод разбивает строку на фрагменты; разделители фрагментов определяются регулярным выражением
–	–	fullmatch	метод проверяет, соответствует ли вся строка регулярному выражению
–	–	finditer	метод возвращает итератор, выдающий объекты соответствия по всем неперекрывающимся совпадениям для повторного шаблона в строке

2 Группы заданий

2.1 Общее описание разработки групп заданий

Процесс разработки новых заданий для задачника Programming Taskbook включает ряд этапов, каждый из которых имеет свои особенности:

Этап 1. Определение формулировки задания, а также набора исходных и результирующих данных.

Этап 2. Разработка алгоритма решения задания, который будет использоваться для генерации контрольных данных.

Этап 3. Разработка алгоритма генерации исходных данных.

Этап 4. Разработка алгоритма визуализации всех данных, связанных с заданием.

Этап 5. Тестирование разработанного задания в различных режимах и для различных языков программирования.

Самыми важными являются этапы 1 и 2. Четкость и однозначность формулировки задания(этап 1) помогает избежать недопониманий и разночтений со стороны учащихся. Если задание сформулировано ясно и понятно, то пользователи смогут легче понять, что от них требуется выполнить. Корректный набор исходных данных позволяет оценить алгоритм, используемый студентом для решения задачи. Разнообразие и уникальность данных помогают избежать стандартных решений. Алгоритм решения(этап 2) задания лежит в основе генерации контрольных данных, которые будут использоваться для проверки правильности выполнения задания.

Не менее важными являются этапы 3 и 4. Важно использовать датчик случайных чисел, чтобы генерировать различные варианты исходных данных(этап 3), включая специальные случаи. Необходимо обеспечить наглядное отображение формулировки задания(этап 4), а также исходных и контрольных данных, уделяя внимание визуализации сложных наборов данных, таких как массивы. Разработанное задание необходимо тестировать(этап 5) в различных режимах и для разных языков

программирования. Все этапы подробно описываются далее на примере разработки одного задания.

Для облегчения создания проектов новых групп заданий используется конструктор TaskGroupCreator. Он создает проект с указанным именем и размещает его в заданном каталоге. Важно отметить, что для работы с TaskGroupCreator необходимо, чтобы на компьютере был установлен Programming Taskbook.

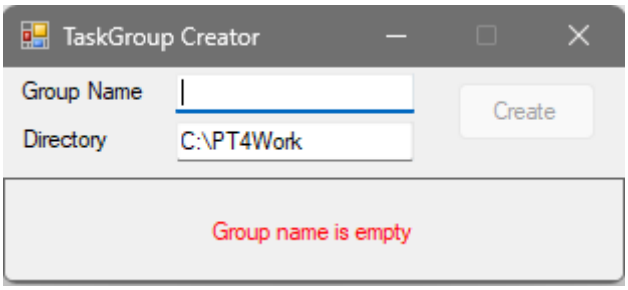

Рис. 2.1.1. Создание нового проекта с помощью TaskGroupCreator

При запуске TaskGroupCreator программа проверяет наличие необходимых компонентов и ожидает ввода имени новой группы заданий. Имя должно быть непустым, не длиннее 9 символов (латинские буквы и цифры) и не заканчиваться цифрой. Например, для группы, посвященной работе с множеством символов в разделе Регулярные выражения, можно выбрать имя Reg1Sym.

После загрузки проекта в Visual Studio необходимо открыть файл Reg1Sym.cpp, где будут реализованы задания группы. Проект также включает файл pt4taskmaker.cpp, реализующий функции конструктора заданий.

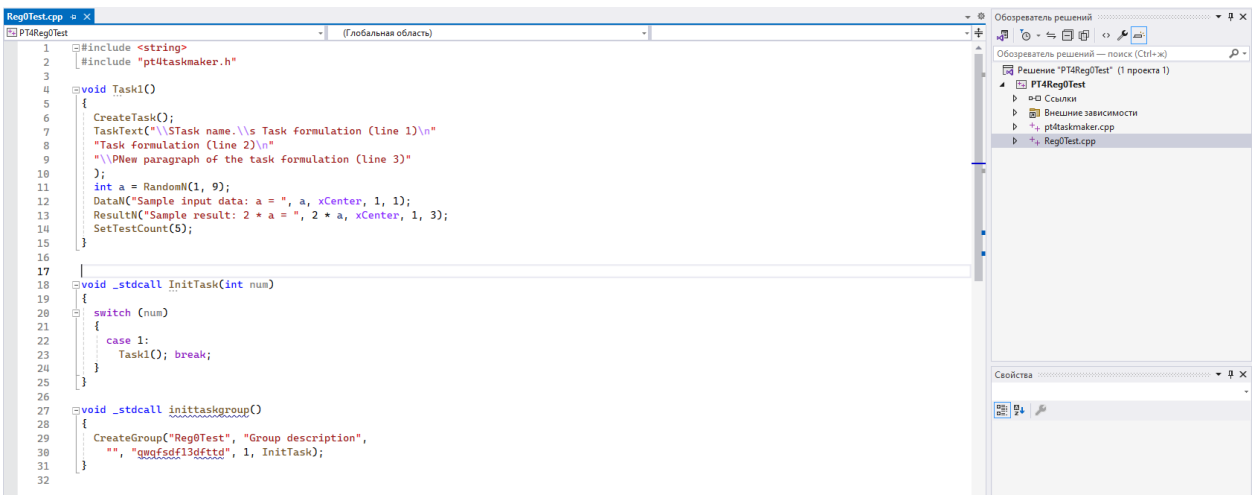

Рис. 2.1.2. Файл Reg0Test.cpp

Файл Reg1Sym.cpp уже содержит заготовку для первого задания (функция Task1). В нем также находятся функции InitTask (инициализирует задание по номеру) и inittaskgroup (определяет общие свойства группы).

Листинг 2.1.1. Функции InitTask и inittaskgroup

```
void _stdcall InitTask(int num)
{
  switch (num)
  {
    case 1:
      Task1(); break;
  }
}
void _stdcall inittaskgroup()
{
  CreateGroup("Reg1Sym", "Group description",
    "", "qwqfsdf13dfttd", 1, InitTask);
}
```

При запуске проекта в каталоге C:\PT4Work создается динамическая библиотека PT4Reg1Sym.dll. Для запуска этой библиотеки используется файл PT4Demo.exe из Programming Taskbook. Передача специальных параметров запускает PT4Demo.exe, который загружает библиотеку и отображает окно задачника с одним из заданий. С помощью клавиш Enter и Backspace можно переходить между заданиями группы.

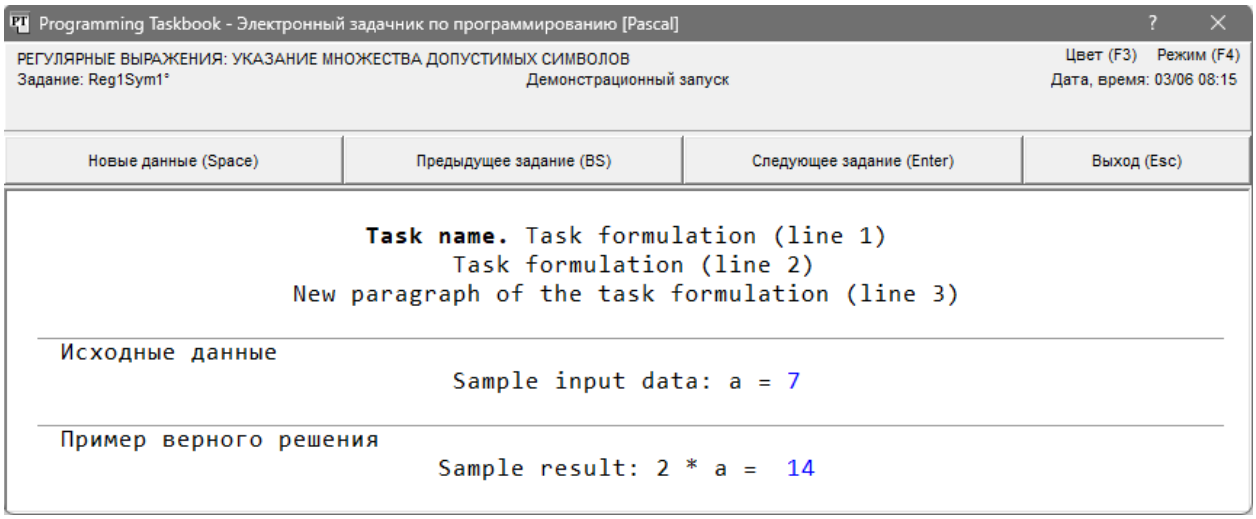

Рис. 2.1.3. Окно задачника Programming Taskbook

Задание пока является простой заготовкой, которая, тем не менее, содержит все элементы, требующиеся в любом задании: это определение

формулировки задания, набора исходных данных и набора контрольных данных.

Листинг 2.1.3. Функция Task1

```
void Task1()
{
    CreateTask();
    TaskText(\\STask name.\\s Task formulation (line 1)\n
          "Task formulation (line 2)\n"
          "\\PNew paragraph of the task formulation (line
     3)"
    );
    int a = RandomN(1, 9);
    DataN("Sample input data: a = ", a, xCenter, 1, 1);
    ResultN("Sample result: 2 * a = ", 2 * a, xCenter, 1,
          3);
    SetTestCount(5);
}
```

В этом случае формулировка задания состоит из трех строк. Все строки формулировки должны располагаться подряд и заканчиваться маркером конца строки \n. В строках можно указывать управляющие команды; в данном случае использованы команды \\S и \\s, выделяющие особым образом название задания, и \\P, помечающая начало нового абзаца.

Примечание. Каждая строка формулировки задания не должна превышать 78 символов. При нарушении этого условия отображение последующего текста формулировки будет отменено, и инициализация задания завершится с ошибкой. Чтобы проверить, как формулировка будет выглядеть в html-формате, можно нажать на метку Mode (F4).

В этом задании набор исходных и результирующих данных простой: одно целое число от 1 до 9 генерируется случайным образом, а результат — это его удвоенное значение. Для отображения данных используются функции DataN и ResultN.

В конце функции Task1 устанавливается количество тестовых испытаний, необходимых для успешного выполнения задания:

```
SetTestCount(5);
```

Данный пример функции Task1 демонстрирует основные принципы разработки учебных заданий. В следующих разделах будет подробно описан процесс создания задания, связанного с регулярными выражениями.

С помощью этого процесса разработки было написано 7 групп заданий: Reg0Begin (общее описание библиотек, 8 задач), Reg1Sym (множества символов, 7 задач), Reg2Esc (Esc-последовательности, 5 задач), Reg3Quanty (квантификаторы, 9 задач), Reg4Group (группирование и ссылки на группы, 3 задачи), Reg5ZLen (директивы нулевой длины, 5 задач) и Reg9Mix (сложные задачи с применением различных методов, 10 задач).

Примеры формулировок для каждой группы заданий:

Reg0Begin(общее описание библиотек):

Reg0Begin1°. Напишите регулярное выражение, которое будет соответствовать любому числу, выведите эти числа.

Reg0Begin4°. Напишите регулярное выражение, которое найдет все числа в строке и заменит их на "число".

Reg0Begin6°. С помощью регулярного выражения найдите и замените все пробелы на символы подчеркивания.

Reg1Sym(множества символов):

Reg1Sym2°. Шестнадцатеричные числа могут содержать цифры 0–9, а также буквы A-F. При использовании их для указания цветов шестнадцатеричные коды могут содержать не более трех символов. Создайте регулярное выражение, чтобы найти действительные шестнадцатеричные коды и выведите их позиции в строке.

Reg1Sym5°. Напишите регулярное выражение, используя диапазон и знак отрицания ^ в квадратных скобках, чтобы найти все года между 1977 и 1982 (включительно) и замените их на 2024, выведите измененную строку.

Reg2Esc(Esc-последовательности):

Reg2Esc1°. Проверить с помощью регулярного выражения, есть ли в строке квадратные скобки '[]'.

Reg2Esc2°. Напиши регулярное выражение и с его помощью найдите все вхождения обратного слэша '\' в строке, замените на слэш '/' и выведите измененную строку.

Reg3Quanty(квантификаторы):

Reg3Quanty3°. Создайте регулярное выражение, которое найдет все фамилии содержащие 4 или более символов и выведите их количество.

Reg3Quanty8°. С помощью регулярного выражения замените все последовательности из 3 или более пробелов на один пробел.

Reg4Group(группирование и ссылки на группы):

Reg4Group1°. С помощью регулярного выражения поместите в первую группу слова, начинающиеся с заглавной буквы, а во вторую со строчной и вывести их количество в строке.

Reg4Group3°. Напишите регулярное выражения, которое найдет все слова, которые содержат 2 или более одинаковых букв подряд, и замените на "слово".

Reg5ZLen(директивы нулевой длины):

Reg5ZLen1°. С помощью регулярного выражения найдите число, которое находится в начале строки.

Reg5ZLen5°. Напишите регулярное выражение, которое находит все вхождения слова "hello" в тексте, но не находит вхождения, если слово "hello" является частью другого слова (например, "hello" в "hello_world" не должно быть найдено) и выведите их количество в строке.

Reg9Mix(сложные задачи с применением различных методов):

Reg9Mix3°. Написать регулярное выражение для проверки корректности записи даты рождения в формате ДД.ММ.ГГГГ или ДД/ММ/ГГГГ и вывести все совпадения.

Reg9Mix6°. С помощью регулярного выражения извлеките из текста все числа, которые написаны словами.

Reg9Mix9°. С помощью регулярного выражения извлеките из текста все HTML-теги, включая атрибуты и значения атрибутов.

2.2 Примеры реализации заданий с применением конструктора

В качестве примера реализации задания рассмотрим первое из заданий раздела Reg0Begin, в котором надо написать регулярное выражение, которое будет соответствовать любому числу.

Начнем с оформления формулировки задания. Нужно написать формулировку задачи в функцию Task1, разбить ее на строки длины, не превосходящей 78.

Получаем следующий фрагмент программы:

Листинг 2.2.1. Формулировка задачи

```
TaskText(
"Напишите регулярное выражение, которое будет соответствовать\n"
"любому числу, выведите эти числа."
);
```

Запустив новый вариант программы, мы увидим в окне задачника текст формулировки:

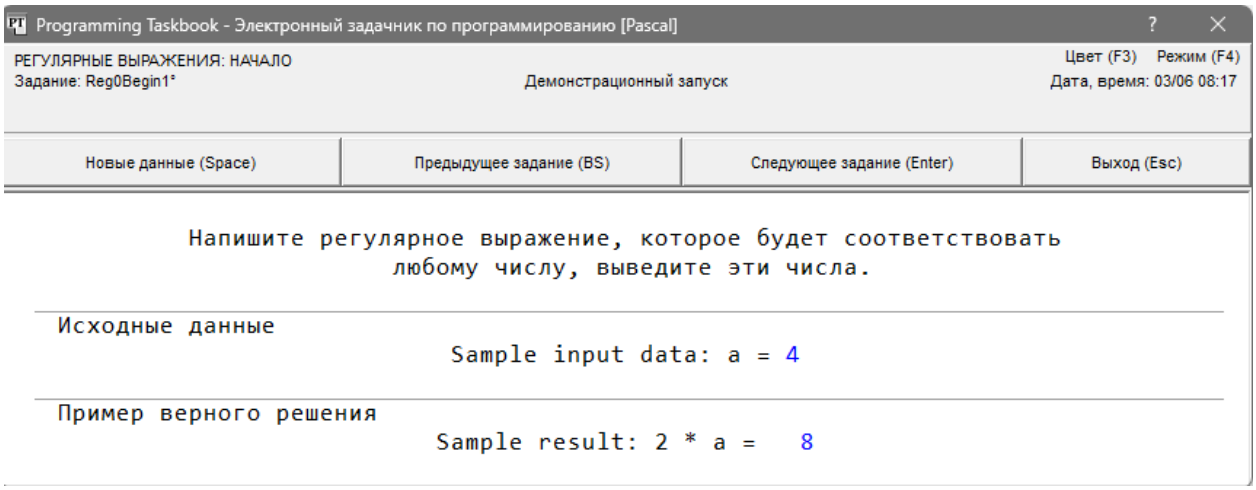

Рис. 2.2.1. Окно задачника Programming Taskbook

Перейдем к генерации исходных данных. Вначале мы должны случайным образом определить количество чисел в строке, а затем — эти числа, учитывая, что итоговая строка, должна достигать не более 78 символов. Для проверки правильности алгоритма вполне достаточно использовать от 5 до 9 чисел:

Листинг 2.2.2. Генерация входных данных

```
int a = RandomN(5, 9);
```

```
string str = "";
for (int i = 0; i < a; i++) {
     str += to_string(RandomN(0, 999)) + (i + 1 != a ? " " :
     "");
}
```

Для генерации случайных целых чисел в диапазоне от a до b удобно использовать вспомогательную функцию RandomN(a, b), входящую в конструктор pt4taskmaker.

Чтобы найти ответ, нужно создать экземпляр класса regex и воспользоваться regex_iterator – класс для выполнения итерации по строке с помощью регулярного выражения, который используется для итерации по всем соответствиям в строке:

Листинг 2.2.3. Решение задачи и вывод результирующих данных

```
regex pattern(R"(\d+)");
sregex_iterator begin(str.begin(), str.end(), pattern);
sregex_iterator end;
int count = 1;
for (sregex_iterator it = begin; it != end; ++it) {
    smatch match = *it;
    cout << match.str() << endl;
}
```

Осталось обеспечить визуализацию исходных и результирующих данных. Чтобы полученный набор элементов отображался по порядку используется переменная count, которая каждый раз увеличивается на единицу, когда выводятся новые результирующие данные и передается в последний параметр функции ResultS, который отвечает за высоту отображаемой строки:

Листинг 2.2.4. Вывод результирующих данных с помощью ResultS

```
regex pattern(R"(\d+)");
sregex_iterator begin(str.begin(), str.end(), pattern);
sregex_iterator end;
int count = 1;
for (sregex_iterator it = begin; it != end; ++it) {
     smatch match = *it;
     ResultS(("Number " + to_string(count) + ": ").c_str(),
     match.str().c_str(), xCenter, count++);
}
```

Итак, мы получили первый вариант реализации данного задания:

Листинг 2.2.5. Полный код задачи Task1

```
void Task1()
{
    CreateTask();
    TaskText(
    "Напишите регулярное выражение, которое будет
     соответствовать\n"
    "любому числу, выведите эти числа."
    );
    int a = RandomN(5, 9);
    string str = "";
    for (int i = 0; i < a; i++) {
          str += to_string(RandomN(0, 999)) + (i + 1 != a ?
          " " : "");
    }
    DataS(str.c_str(), xCenter, 1);
    regex pattern(R"(\d+)");
    sregex_iterator begin(str.begin(), str.end(), pattern);
    sregex_iterator end;
    int count = 1;
    for (sregex_iterator it = begin; it != end; ++it) {
          smatch match = *it;
          ResultS(("Number " + to_string(count) + ":
          ").c_str(), match.str().c_str(), xCenter,
          count++);
    }
    SetTestCount(7);
}
```

Запустив полученный вариант программы, мы увидим окно задачника с образцом исходных данных и примером правильного решения:

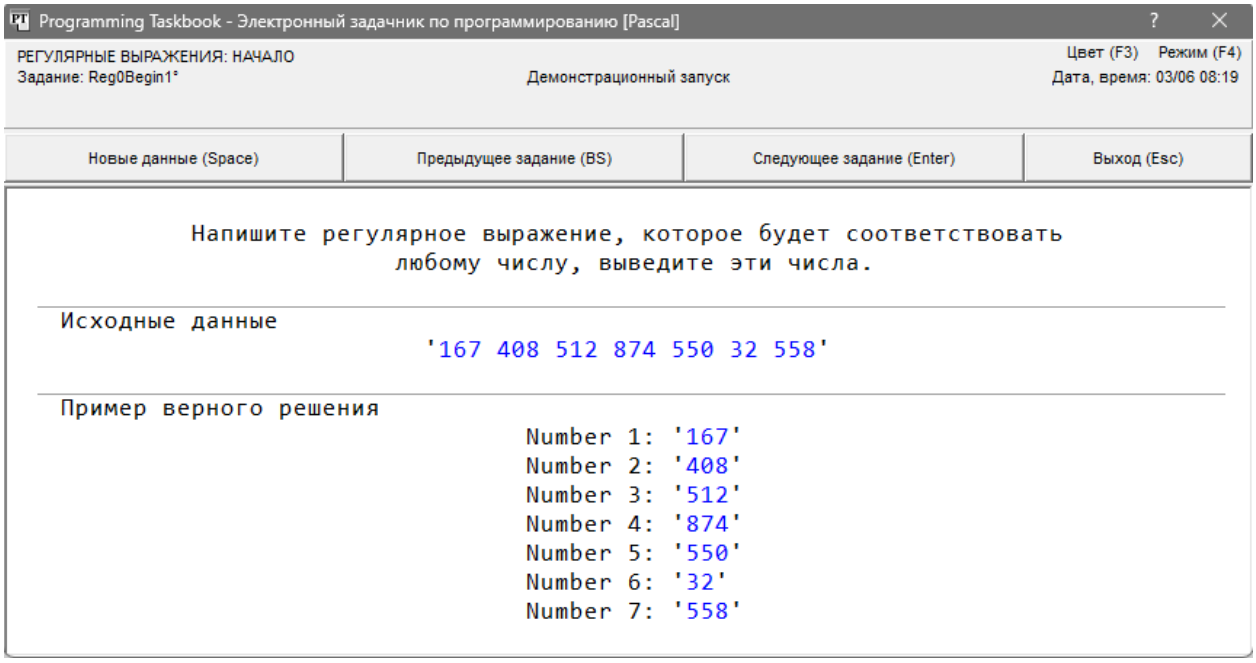

Рис. 2.2.2. Окно задачника Programming Taskbook

При помощи пробела(“Новые данные”) можно просмотреть различные варианты исходных данных, оценить корректность их визуализации, а также правильность примера правильного решения.

2.3 Примеры решения заданий

Наличие динамической библиотеки в каталоге C:\PT4Work позволяет выполнять входящие в нее учебные задания в любых других рабочих каталогах, связанных с электронным задачником. Имеется лишь одно ограничение: задания можно выполнять только для 32-разрядных компиляторов. Для тестирования разработанных заданий на языке Python можно предусмотреть отдельный рабочий каталог (например, C:\PT4WorkPy), настроив его на какую-либо среду разработки для языка Python с помощью программы PT4Setup. После этого для создания заготовки можно, как обычно, использовать программу PT4Load:

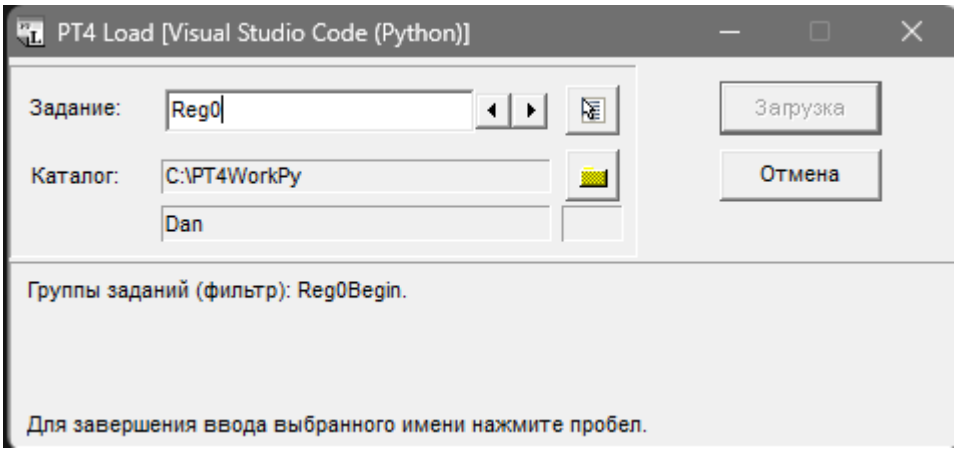

Рис. 2.3.1. Окно программы PT4 Load

Заметим, что программа PT4Load успешно обнаружила новую группу Reg0Begin. Созданная заготовка на языке Python автоматически загружается в

выбранную среду разработки:

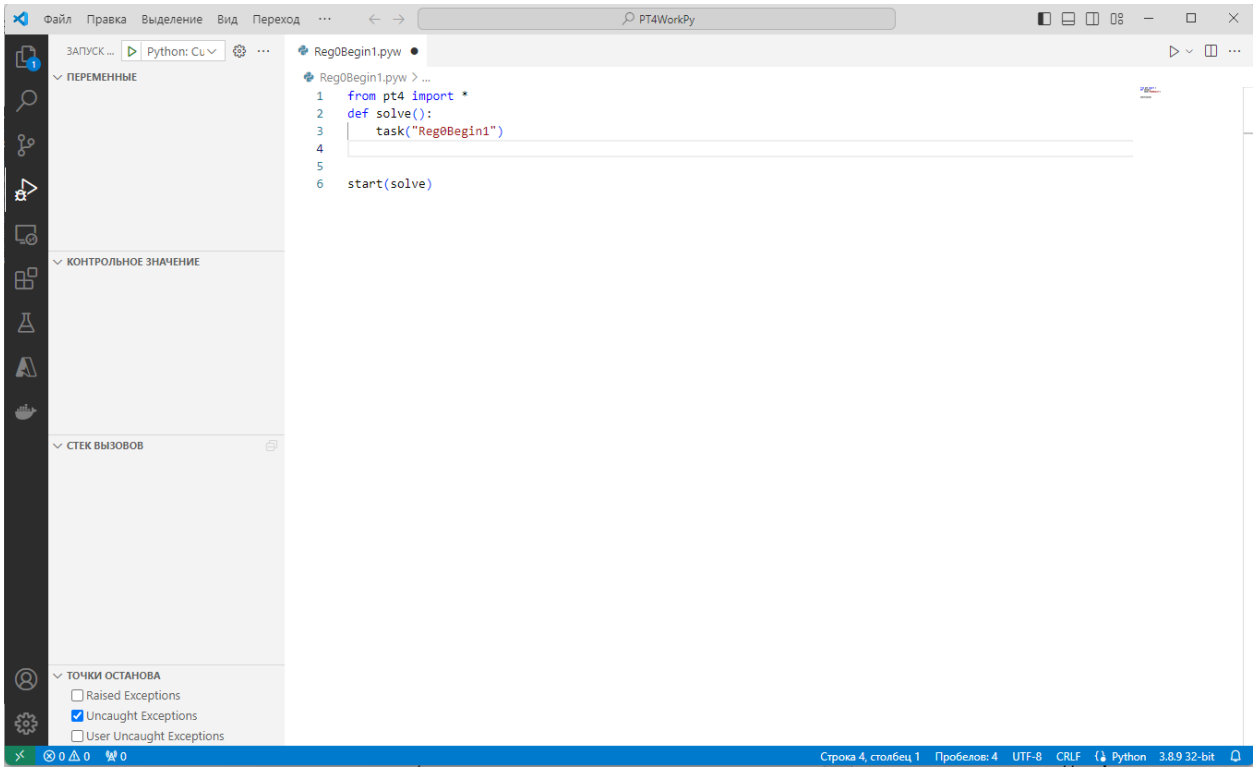

Рис. 2.3.2. Заготовка на языке Python

Для решения задачи Reg0Begin1 на языке Python мы используем другой алгоритм решения, так как возможности библиотек для регулярных выражений в каждом языке разные. Для ввода исходного набора данных достаточно вызвать функцию get_str, а для вывода функцию put:

Листинг 2.3.1. Код решения задачи на языке Python

```
import re
from pt4 import *
def solve():
    task("Reg0Begin1")
    str = get_str()
    regex = R"\d+"
    list = re.findall(regex, str)
    for it in list:
        put(it)

start(solve)
```

Данная программа успешно пройдет все тесты, и в результате на экран будет выведено окно задачника с сообщением об успешном решении задачи:

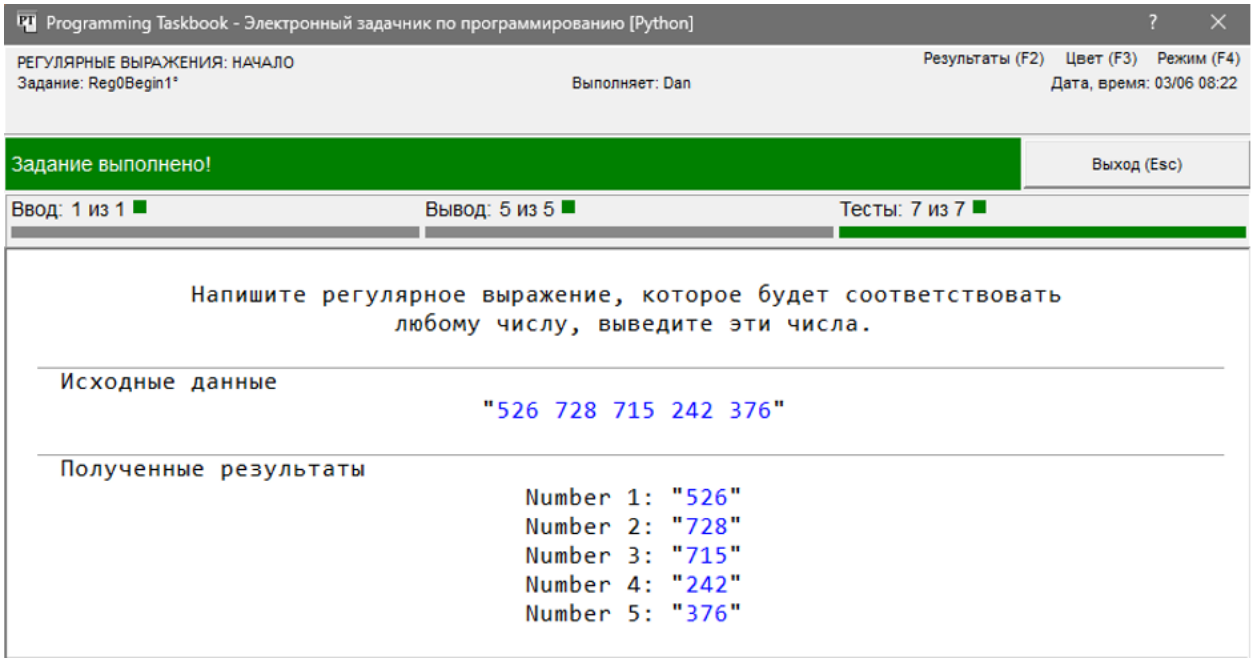

Рис. 2.3.3. Окно задачника Programming Taskbook

Примечание. Аналогичным образом можно разрабатывать и другие задания новой группы. Если предположить, что новое задание реализовано в функции Task2, то достаточно откорректировать функции InitTask и inittaskgroup следующим образом(в функции inittaskgroup надо изменить параметр, определяющий количество заданий в группе):

Листинг 2.3.2. Функции InitTask и inittaskgroup

```
void _stdcall InitTask(int num)
{
  switch (num)
  {
    case 1:
      Task1(); break;
    case 2:
        Task2(); break;
  }
}

void _stdcall inittaskgroup()
{
  CreateGroup("Reg0Begin", "Регулярные выражения: Начало",
          "", "qwqfsdf13dfttd", 2, InitTask);
}
```

3 Описание разработанных файлов дополнений

3.1 Общее описание разработки файлов дополнений

Начиная с версии 4.14, к заданиям электронного задачника можно присоединять дополнительные тексты (дополнения), которые берутся из текстовых файлов специального формата (файлов дополнений) и отображаются в окне задачника или в сгенерированных им html-документах. В окне задачника текст дополнения выводится в разделе отладки, в html-документе — сразу после текста задания (или после текста преамбулы). Кроме того, с помощью файлов дополнений можно определять индивидуальные программы-заготовки для различных учебных заданий.

Процесс разработки файлов дополнения для заданий задачника Programming Taskbook [9] включает ряд этапов, каждый из которых имеет свои особенности:

Этап 1. Определение начальной части заготовки всех заданий.

Этап 2. Указание комментария, который отображается в окне задачника.

Этап 3. Написание кода заготовки для решения задания.

Этап 4. Включение дополнительного комментария, добавленного в конец заготовки задания.

Для определения начальной части(этап 1) заготовки всех заданий используется номер 0. В начальную часть заготовки обычно добавляются директивы подключения модулей.

Переключатели [-<список режимов>] и [+<список режимов>] отключают или включают режимы отображения последующего текста:

W - отображение в окне задачника (этап 2),

H - отображение в html-документе с описанием группы заданий,

L - отображение в программе-заготовке.

По умолчанию все режимы включены.

После указания переключателя [1] последующий текст будет относиться

только к первому заданию группы(этап 3). Для того чтобы последующий текст относился к нескольким языкам или номерам, соответствующие переключатели надо указать на одной и той же строке[7][9]. Можно указывать диапазоны номеров[3-6].

Задание с одним и тем же номером можно определять в виде нескольких фрагментов, расположенных в разных частях файла дополнений. Фрагменты объединяются в том порядке, в котором они располагаются в файле дополнений(этап 4).

Переключатели языка и номера (зеленого цвета) могут находиться на одной строке. Переключатель языка начинается с символа "=", после которого указывается язык. В версии 4.23 поддерживаются следующие переключатели языка: [=Pascal] (для среды Lazarus), [=PascalABC] (для среды PascalABC.NET), [=CPP] (C++), [=CS] (C#), [=VB.NET], [=FS] (F#), [=Python] (вместе с вариантами [=Python-2] и [=Python-3]), [=Java], [=Ruby], [=Julia], [=C].

Некоторые переключатели должны занимать отдельную строку; такие переключатели выделяются черным цветом.

Переключатель [text] (этап 2) устанавливает для последующего текста формат абзаца; текст в этом формате добавляется к преамбуле группы (или подгруппы) или к формулировке задания.

Переключатель [code] (этап 3) устанавливает для последующего текста формат листинга; текст в этом формате добавляется к формулировке задания и к созданной для этого задания программе-заготовке.

Текст преамбулы группы определяется в режиме text для номера 0. Зеленым цветом выделяются управляющие последовательности. Они начинаются с экранирующего символа %, который можно изменить, используя переключатель [escape<новый символ>], например [escape$].

Ниже перечислены все управляющие последовательности для версии 4.15: %{group} - имя группы, %{task} - имя задания, %{number} - номер задания, %{I}, %{i} - выделение курсивом, %{B}, %{b} - полужирное

выделение, %{S}, %{s} - специальное выделение, %{M}, %{m} – моноширинное выделение, %{br} - разрыв строки, %~ - неразрывный пробел, %<<, %>> - угловые кавычки, %-- - короткое тире, %--- - длинное тире, %{mark} - круглый маркер. Команды выделения текста учитываются только в режиме html-документа.

Имя файла дополнений должно начинаться с префикса PT4, после которого следует имя группы заданий. После имени группы заданий может указываться используемый язык программирования (обрамленный квадратными скобками или символами подчеркивания, например: PT4GroupDemo[Python].txt или PT4GroupDemo_Python_.txt), а также локаль (язык интерфейса - русский или английский) _ru или _en(например: PT4GroupDemo_ru.txt или PT4GroupDemo_Python__en.txt). Символы подчеркивания рекомендуется использовать вместо квадратных скобок, если файл дополнений предназначен для размещения в удаленных ftp-репозиториях. Файл дополнений имеет расширение .txt. По умолчанию предполагается, что файл дополнений имеет кодировку CP1251(эта кодировка используется в редакторе программы PTVarMaker). Начиная с версии 4.22, можно использовать файлы дополнений в кодировке UTF-8; в этом случае файл должен содержать строку с переключателем [UTF-8].

Поиск файла дополнений вначале осуществляется в рабочем каталоге учащегося, а затем, при его отсутствии, — в подкаталоге LIB системного каталога задачника. В каждом каталоге вначале ищется файл с явно указанным языком и локалью, соответствующей текущей локали задачника, затем файл с явно указанным языком, но без локали, затем — файл без указания языка, но с локалью, и, наконец, файл без указания языка и локали. Для файлов без указания языка дополнительно проверяется, что в них присутствует хотя бы один переключатель, соответствующий требуемому языку; если такой переключатель отсутствует, то поиск продолжается.

Для разработки файлов дополнений (и файлов внешних групп) можно использовать программу «Конструктор вариантов» PTVarMaker версии 3.0

или выше. Для таких файлов программа PTVarMaker обеспечивает специальную подсветку синтаксиса и предоставляет средства быстрой проверки правильности подготовленных файлов с помощью модулей PT4Demo и PT4Load.

3.2 Примеры реализации файлов дополнений

В качестве примера реализации файла дополнения рассмотрим первую группу заданий Reg0Begin. Создаем файл дополнение для группы Reg0Begin, с названием PT4Reg0Begin:

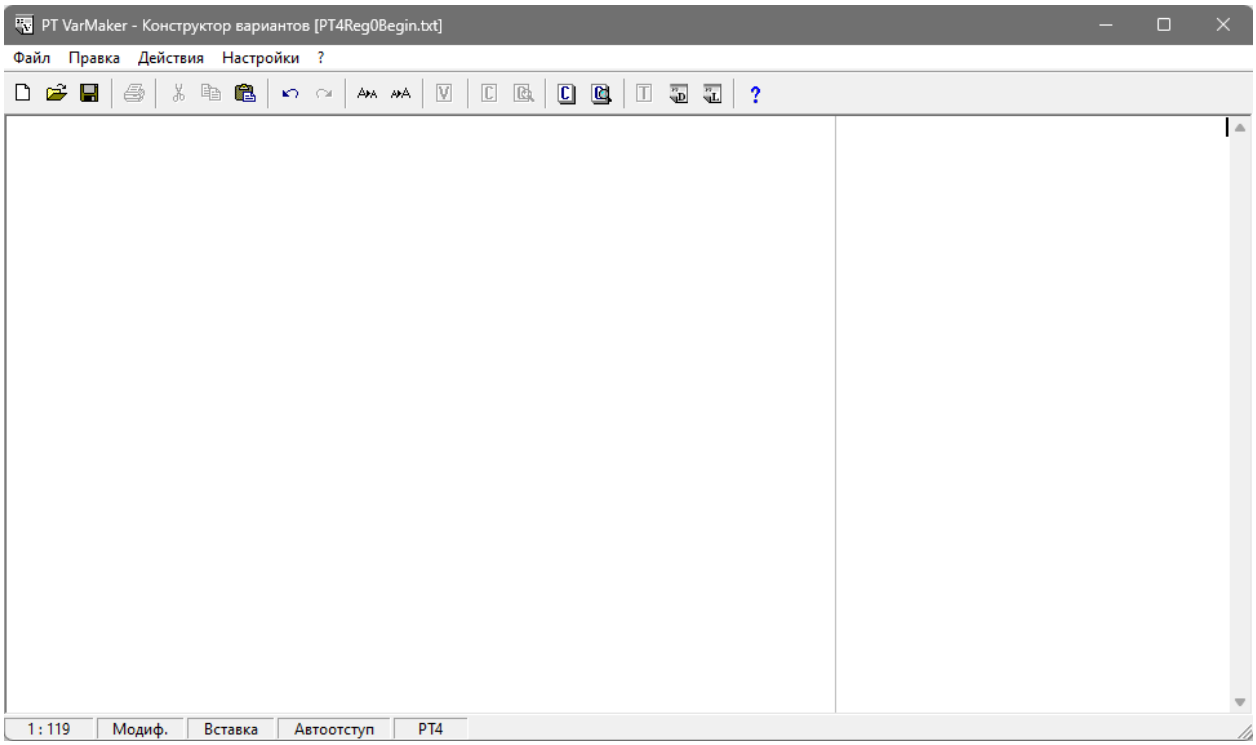

Рис. 3.2.1. Окно конструктора PT VarMaker

Для начала следует подключить соответствующую библиотеку для работы с регулярными выражениями для каждого языка. Для этого нужно изменить переключатель номера на начальный и для каждого языка добавить директивы подключения модулей:

Листинг 3.2.1. Текст файла дополнений с подключением модулей

```
[0][=Python]
# В Python, регулярные выражения доступны в библиотеке re.
# re – это модуль Python, содержащий методы для работы с
регулярными //выражениями.
```

```
import re
[=CS]
// Класс Regex и связанные с ним классы описаны в пространстве
имен
// System.Text.RegularExpressions.
using System.Text.RegularExpressions;
[=CPP]
// В C++ регулярные выражения реализуются как часть библиотеки
STL //(Standard Template Library), и для работы с регулярными
выражениями //требуется использование класса regex и связанных с
ним функций и //методов.
#include <regex>
```

Теперь при загрузке заготовки задания из группы будет автоматически подключена библиотека для работы с регулярными выражениями:

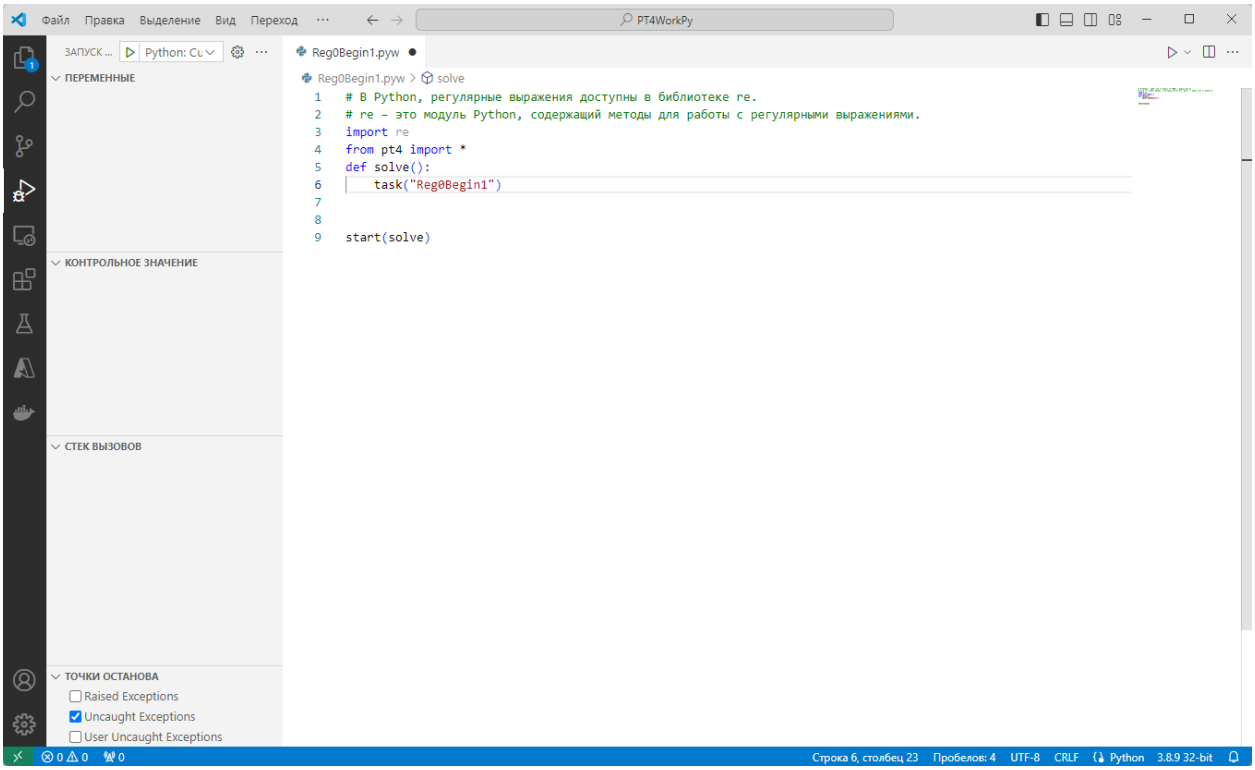

Рис. 3.2.2. Заготовка на языке Python

Следующим этапом будет написание комментария для отображения в окне задачника. Для этого нужно добавить переключатель с номером задания, для которого нужно написать комментарий, указать язык, для которого будет соответствовать текст[text](этот переключатель также нужно добавить) к формулировке задания:

Листинг 3.2.2. Комментарий, который отображается в окне задачника

```
[1][=Python]
[text]
```

```
В первом задании достаточно написать регулярное выражение в
переменную regex. Функция
findall найдет все вхождения в строке %{B}str%{b} по шаблону
%{B}regex%{b}. И с помощью
цикла %{I}for%{i} выведем все найденные совпадения через
функцию %{I}put%{i}.
%{S}Используйте множества символов%{s}: \d — десятичная
цифра, т. е. [0-9].
%{S}Используйте квантификаторы%{s}: + — 1 или более
соответствий.
%{S}Используйте функцию библиотеки re%{s}: list<str>
re.findall(regex, str) - метод ищет все
подстроки в строке str, которые соответствуют регулярному
выражению regex;
```

Теперь, в результате запуска заготовки, появится новый текст в окне задачника, который опишет некоторые возможности языка регулярных выражений и библиотеки, которая работает с ними:

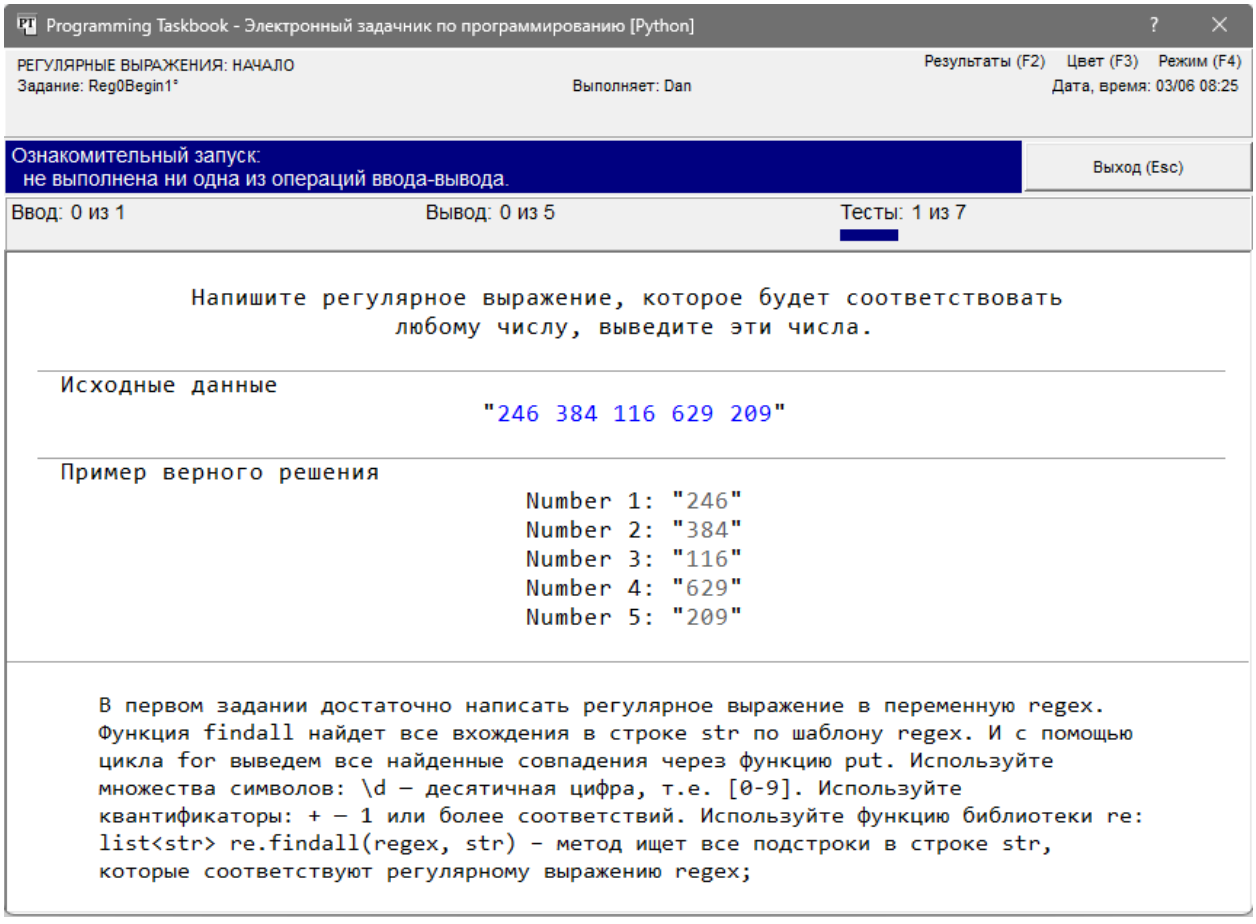

Рис. 3.2.3. Окно задачника Programming Taskbook

Осталось написать код заготовки задания. Для этого нужно изменить переключатель отображения текста в окне задачника, чтобы код следующий

текст не отображался в окне задачника[-W]. После чего можно добавить сам код заготовки:

Листинг 3.2.3. Код заготовки к задаче Task1

```
[1][=Python]
[-W]
[code]
def solve():
    task("%{task}")
    str = get_str()
    regex = R""
    list = re.findall(regex, str)
    for it in list:
        put(it)
```

В данном примере будет создана заготовка с практически полноценным решением, ведь пользователю остается только написать регулярное выражение для решения задачи в переменную regex(о чем и указано в комментарии в окне задачника). Таким образом код заготовки задания после загрузки с помощью программы PT4Load:

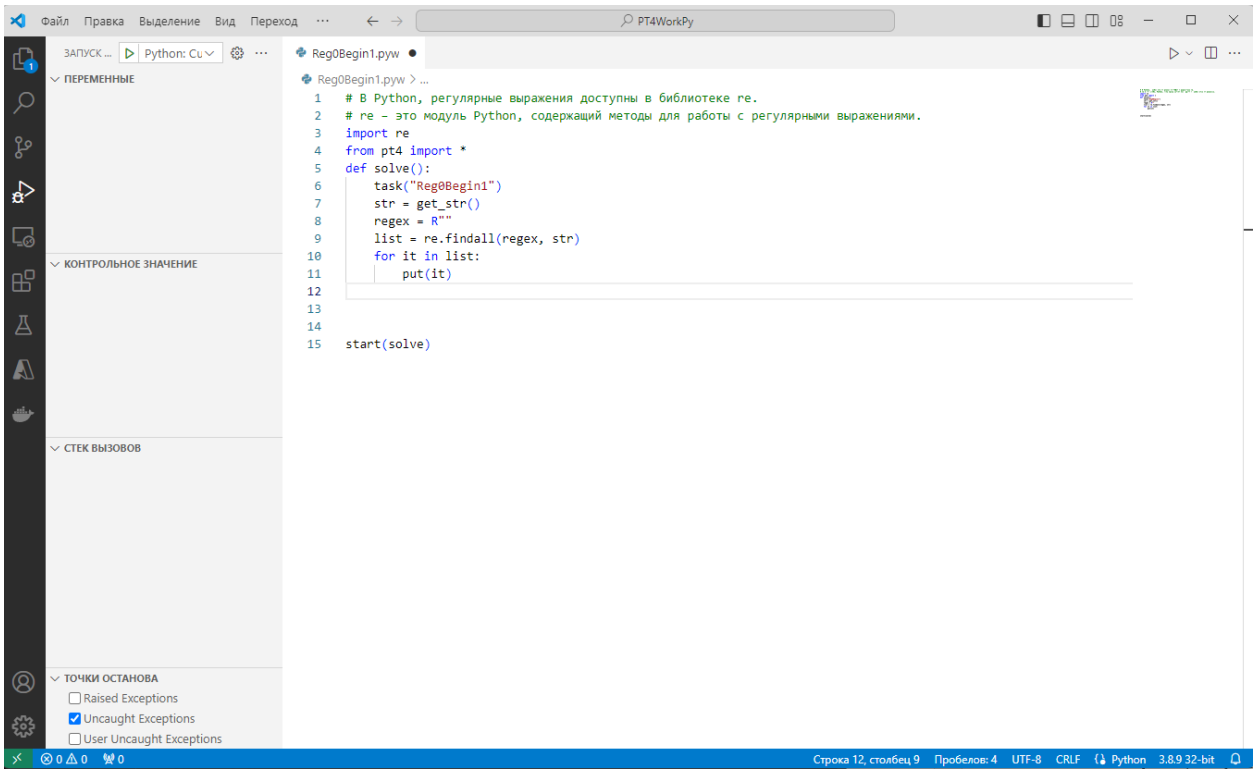

Рис. 3.2.4. Заготовка на языке Python

Дополнительно можно добавить еще один комментарий после кода функции, который более подробно опишет некоторые моменты в заготовке, а также даст подсказку для решения(для начальных групп). Для этого нужно

повторить предыдущие переключатели, связанные с этим заданием, чтобы объединились фрагменты по порядку:

Листинг 3.2.4. Дополнительный комментарий в конце заготовки

```
[1][=Python]
[code]
# re.findall(regex, str) -  ищет все вхождения шаблона regex в строке `str`.
# \d+ - соответствует одному или более цифрам (0-9).
```

Итоговое отображение заготовки задания после добавления всех комментариев и кода, который поможет разобраться в изучаемой теме:

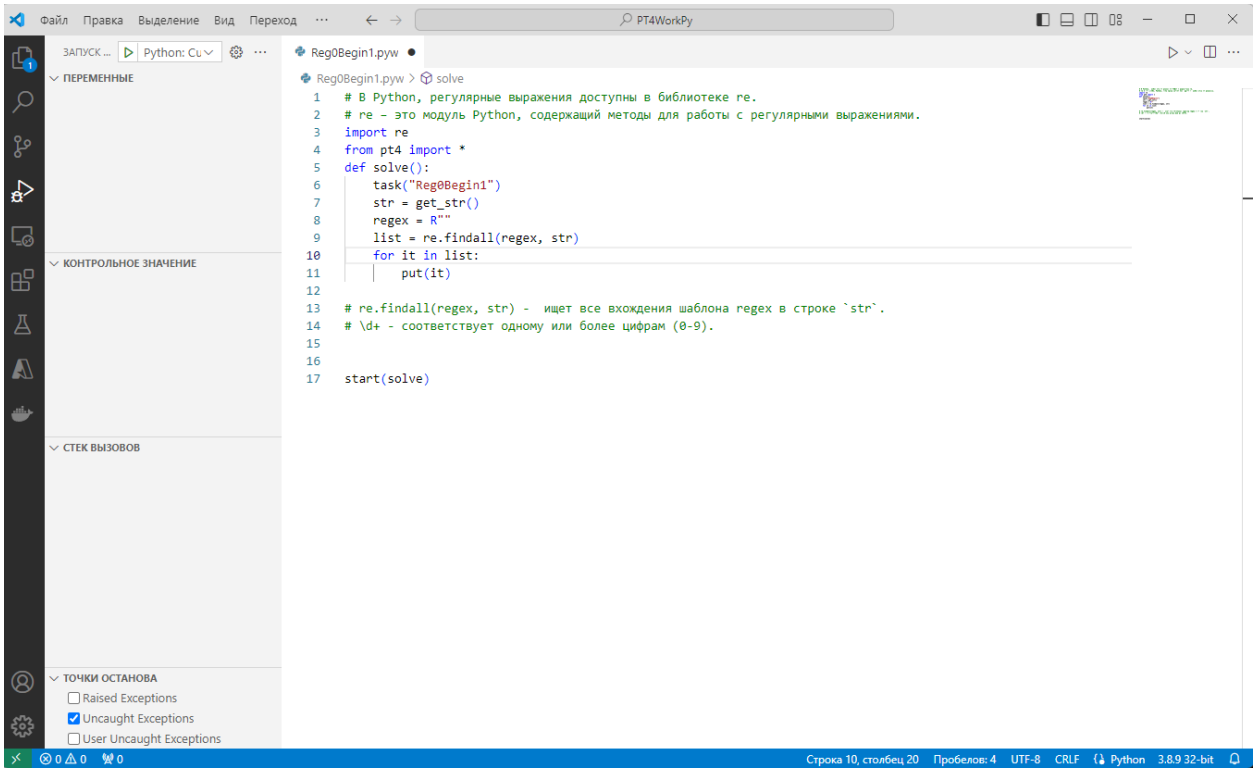

Рис. 3.2.5. Заготовка на языке Python

3.3 Примеры решения заданий с файлом дополнения

Рассмотрим первую задачу из группы Reg1Sym, которая связана с множеством символов в регулярном языке. Формулировка звучит таким образом: “Найти названия всех животных из трёх букв(bat, cat, rat) в следующей строке”. После загрузки задания из библиотеки Reg1Sym.dll с помощью программы PT4Load, получаем заготовку со следующим кодом:

Листинг 3.3.1. Полный код заготовки задачи Reg1Sym1

```
# В Python, регулярные выражения доступны в библиотеке re.
```

```
# re – это модуль Python, содержащий методы для работы с
регулярными выражениями.
import re
from pt4 import *
def solve():
    task("Reg1Sym1")
    str = get_str()
    regex = R""
    list = None
    for it in list:
        put(it)

start(solve)
```

Теперь вместо пустой функции solve и ее вызова мы получаем: подключенный модуль re для работы с регулярными выражениями, строку str, в которую получили исходные данные, заготовку regex для написания регулярного выражения, переменную list, в которую будет записан полученный список совпадений, цикл for с выводом результирующих данных в функцию put. А при запуске получаем дополнительный комментарий для решения задачи:

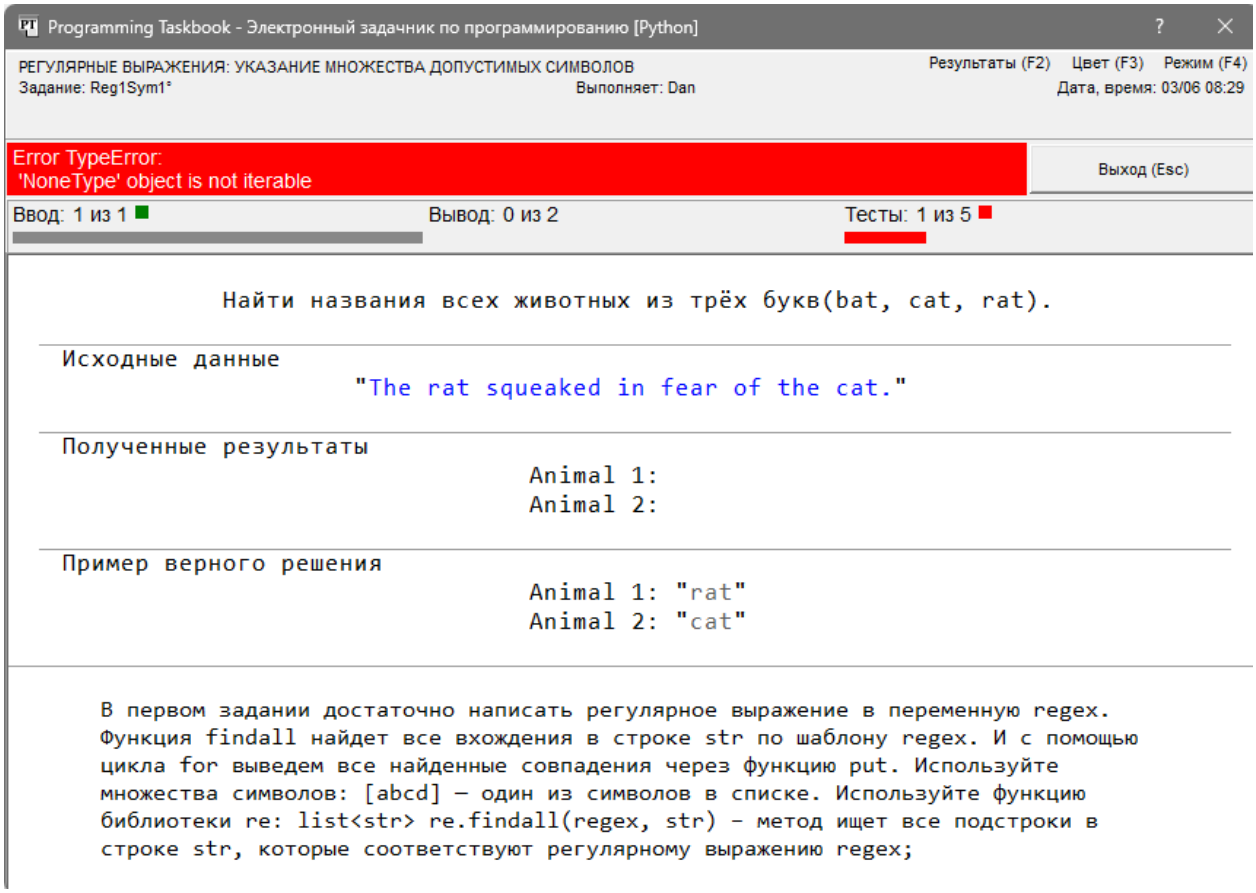

Рис. 3.3.1. Окно задачника Programming Taskbook

С его помощью становится понятно, что нужно использовать функцию findall, которая принимает строку и шаблон для нахождения всех совпадений. Также описывается возможность регулярного выражения для написания шаблона. Изменив код, с помощью данных указаний получаем следующее:

Листинг 3.3.2. Решение задачи Reg1Sym1

```
# В Python, регулярные выражения доступны в библиотеке re.
# re – это модуль Python, содержащий методы для работы с
регулярными выражениями.
import re
from pt4 import *
def solve():
    task("Reg1Sym1")
    str = get_str()
    regex = R"[bcr]at"
    list = re.findall(regex, str)
    for it in list:
        put(it)

start(solve)
```

Теперь данная программа успешно пройдет все тесты, и в результате на экран будет выведено окно задачника с сообщением об успешном решении задачи:

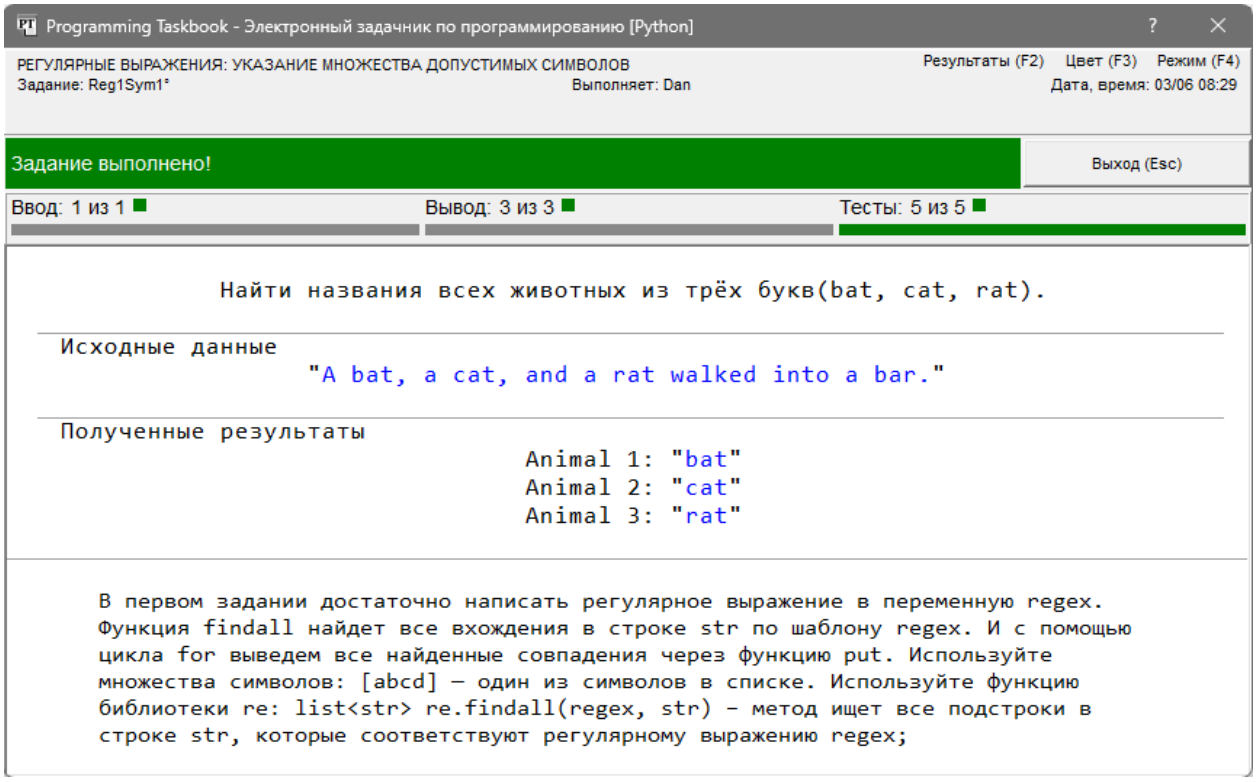

Рис. 3.3.2. Окно задачника Programming Taskbook

Заключение

В результате проделанной работы удалось выполнить все поставленные задачи.

Подготовлено подробное описание библиотек для работы с регулярными выражениями в языках C#, C++ и Python. Представлены примеры использования каждой библиотеки. Дана сравнительная характеристика возможностей библиотек для различных языков программирования.

Было написано 47 формулировок задач по регулярным выражениям, которые были разделены на 7 групп заданий, из которых 5 являются основными: Reg1Sym (множества символов), Reg2Esc (Esc-последовательности), Reg3Quanty (квантификаторы), Reg4Group (группирование и ссылки на группы), Reg5ZLen (директивы нулевой длины), а 2 дополнительными: Reg0Begin (общее описание библиотек) и Reg9Mix (сложные задачи с применением различных методов).

Каждая группа заданий имеет библиотеку dll, в которой реализованы задачи, охватывающие основные возможности регулярных выражений.

Кроме того, для каждой группы были подготовлены файлы дополнений с инструкциями по использованию данных возможностей для указанных языков программирования.

Приведены примеры решений задач для каждого задания на языках программирования C#, C++, Python.

Все разработанные материалы, включая формулировки заданий, файлы дополнения и примеры решений, доступны на GitHub по ссылке https://github.com/hr5h/training_system_regex.

Литература

1. Сайт с официальной документацией для Python 3.11.3. – URL: https://docs.python.org/3/library/re.html#match-objects (дата обращения 02.06.2024).
2. Сайт Викиучебник. – URL: https://ru.wikibooks.org/wiki/Регулярные_выражения (дата обращения 02.06.2024).
3. Сайт с официальной документацией для C++. – URL: https://learn.microsoft.com/ru-ru/cpp/standard-library/regex?view=msvc-150 (дата обращения 02.06.2024).
4. Сайт с документацией для C++. – URL: https://cplusplus.com/reference/regex/ (дата обращения 02.06.2024).
5. Сайт с документацией для C++. – URL: https://en.cppreference.com/w/cpp/regex (дата обращения 02.06.2024).
6. Сайт с переводной документацией для C++. – URL: https://runebook.dev/ru/docs/cpp/regex (дата обращения 02.06.2024).
7. Сайт с документацией для C++, созданной участниками Stack Overflow. – URL: https://sodocumentation.net/cplusplus/topic/1681/regular-expressions (дата обращения 02.06.2024).
8. Абрамян М. Э. Электронный задачник по программированию. – URL: http://ptaskbook.com (дата обращения 02.06.2024).
9. Сайт электронного задачника по программированию “Programming TaskBook”, использование файлов дополнений – URL: http://ptaskbook.com/ru/teacherpack/tmaker_add.php (дата обращения 02.06.2024).
10. Джеффри Фридл: Регулярные выражения. – СПб. : Питер, 2018.