



Государственное бюджетное образовательное учреждение высшего образования
Московской области

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ

Колледж космического машиностроения и технологий

КУРСОВОЙ ПРОЕКТ

По МДК.01.02 «Прикладное программирование»

Тема: «Разработка парсера HTML на Python»

Выполнил студент

Пищулин И.А.

Группа П1-17

_____ (Подпись)

_____ (Дата сдачи работы)

Принял преподаватель

Гусятинер Л.Б.

_____ (Оценка)

_____ (Подпись)

Королёв 2020 г.

Оглавление

Введение.....	3
1. Теоретическая часть	4
1.1. Описание предметной области.....	4
1.2. Описание существующих разработок	8
1.2.1. ParserOK.	8
1.2.2. Datacol.....	9
2. Проектная часть	11
2.1. Диаграмма прецедентов.....	11
2.2. Выбор инструментов	11
2.3. Проектирование сценария	12
2.4. Диаграмма классов	13
2.5. Описание главного модуля.....	13
2.6. Описание спецификаций к модулям.....	16
2.7. Описание модулей	16
2.8. Описание тестовых наборов модулей	20
2.9. Описание применения средств отладки	21
2.10. Анализ оптимальности использования памяти и быстродействия ..	22
3. Эксплуатационная часть	23
3.1. Руководство оператора	23
3.1.1. Назначение программы.....	23
3.1.2. Условия выполнения программы	23
3.1.3. Выполнение программы	23
3.1.4. Сообщение оператору	25
3.2. To-Do лист	25
Заключение	26
Список литературы и интернет-источников	27

Введение

Целью данного курсового проекта является разработка парсера HTML. Эта тема является актуальной на данный момент, потому что в наше время web разработки выходят на один из первых планов в программировании. Данная курсовая работа позволит получить из HTML кода абстрактное дерево классов, которое в дальнейшем можно использовать в создании браузеров.

В первой части будет рассмотрена предметная область данной темы, а также несколько продуктов по данной теме.

Во второй части будут рассмотрены классы и методы, которые были разработаны, структура программной части и листинги ключевых частей программных модулей.

В третьей части будет рассмотрено руководство для пользователей, а также подробная инструкция по использованию программы.

В заключительной части будет приведен To-do лист с планами по доработке программы, а также сделаны общие выводы о получившемся проекте.

1. Теоретическая часть

1.1. Описание предметной области

Парсер — это программа, сервис или скрипт, который собирает данные с указанных веб-ресурсов, анализирует их и выдает в нужном формате.

С помощью парсеров можно делать много полезных задач:

- **Цены.** Актуальная задача для интернет-магазинов. Например, с помощью парсинга вы можете регулярно отслеживать цены конкурентов по тем товарам, которые продаются у вас. Или актуализировать цены на своем сайте в соответствии с ценами поставщика (если у него есть свой сайт).
- **Товарные позиции:** названия, артикулы, описания, характеристики и фото. Например, если у вашего поставщика есть сайт с каталогом, но нет выгрузки для вашего магазина, вы можете спарсить все нужные позиции, а не добавлять их вручную. Это экономит время.
- **Метаданные:** SEO-специалисты могут парсить содержимое тегов title, description и другие метаданные.
- **Анализ сайта.** Так можно быстро находить страницы с ошибкой 404, редиректы, неработающие ссылки и т. д.

Для справки. Есть еще серый парсинг. Сюда относится скачивание контента конкурентов или сайтов целиком. Или сбор контактных данных с агрегаторов и сервисов по типу Яндекс.Карт или 2Гис (для спам-рассылок и звонков). Но мы будем говорить только о белом парсинге, из-за которого у вас не будет проблем.

HTML (от англ. *HyperText Markup Language* — «язык гипертекстовой разметки») — стандартизированный язык разметки документов во Всемирной паутине. Большинство веб-страниц содержат описание разметки

на языке HTML (или XHTML). Язык HTML интерпретируется браузерами; полученный в результате интерпретации форматированный текст отображается на экране монитора компьютера или мобильного устройства.

Язык HTML до 5-й версии определялся как приложение SGML (стандартного обобщённого языка разметки по стандарту ISO 8879). Спецификации HTML5 формулируются в терминах DOM (объектной модели документа).

Язык XHTML является более строгим вариантом HTML, он следует синтаксису XML и является приложением языка XML в области разметки гипертекста.

Во всемирной паутине HTML-страницы, как правило, передаются браузерам от сервера по протоколам HTTP или HTTPS, в виде простого текста или с использованием шифрования.

Язык гипертекстовой разметки HTML был разработан британским учёным Тимом Бернерсом-Ли приблизительно в 1986—1991 годах в стенах ЦЕРНа в Женеве в Швейцарии. HTML создавался как язык для обмена научной и технической документацией, пригодный для использования людьми, не являющимися специалистами в области вёрстки. HTML успешно справлялся с проблемой сложности SGML путём определения небольшого набора структурных и семантических элементов — дескрипторов. Дескрипторы также часто называют «тегами». С помощью HTML можно легко создать относительно простой, но красиво оформленный документ. Помимо упрощения структуры документа, в HTML внесена поддержка гипертекста. Мультимедийные возможности были добавлены позже.

Первым общедоступным описанием HTML был документ «Теги HTML», впервые упомянутый в Интернете Тимом Бернерсом-Ли в конце 1991 года. В нём описываются 18 элементов, составляющих первоначальный, относительно простой дизайн HTML. За исключением тега

гиперссылки, на них сильно повлиял SGMLguid, внутренний формат документации, основанный на стандартном обобщенном языке разметки (SGML), в CERN. Одиннадцать из этих элементов всё ещё существуют в HTML 4.

Изначально язык HTML был задуман и создан как средство структурирования и форматирования документов без их привязки к средствам воспроизведения (отображения). В идеале, текст с разметкой HTML должен был без стилистических и структурных искажений воспроизводиться на оборудовании с различной технической оснащённостью (цветной экран современного компьютера, монохромный экран органайзера, ограниченный по размерам экран мобильного телефона или устройства и программы голосового воспроизведения текстов). Однако современное применение HTML очень далеко от его изначальной задачи. Например, тег `<table>` предназначен для создания в документах таблиц, но иногда используется и для оформления размещения элементов на странице. С течением времени основная идея платформонезависимости языка HTML была принесена в жертву современным потребностям в мультимедийном и графическом оформлении.

Абстрактное синтаксическое дерево (АСД) — в информатике конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены (помечены) с операторами языка программирования, а листья — с соответствующими операндами. Таким образом, листья являются пустыми операторами и представляют только переменные и константы.

Синтаксические деревья используются в парсерах для промежуточного представления программы между деревом разбора (конкретным синтаксическим деревом) и структурой данных, которая затем используется в качестве внутреннего представления в компиляторе или интерпретаторе

компьютерной программы для оптимизации и генерации кода. Возможные варианты подобных структур описываются абстрактным синтаксисом.

Абстрактное синтаксическое дерево отличается от дерева разбора тем, что в нём отсутствуют узлы и рёбра для тех синтаксических правил, которые не влияют на семантику программы. Классическим примером такого отсутствия являются группирующие скобки, так как в АСД группировка операндов явно задаётся структурой дерева.

Для языка, который описывается контекстно-свободной грамматикой, какими являются почти все языки программирования, создание абстрактного дерева в синтаксическом анализаторе является тривиальной задачей. Большинство правил в грамматике создают новую вершину, а символы в правиле становятся рёбрами. Правила, которые ничего не привносят в АСД (например, группирующие правила), просто заменяются в вершине одним из своих символов. Кроме того, анализатор может создать полное дерево разбора и затем пройти по нему, удаляя узлы и рёбра, которые не используются в абстрактном синтаксисе, для получения АСД.

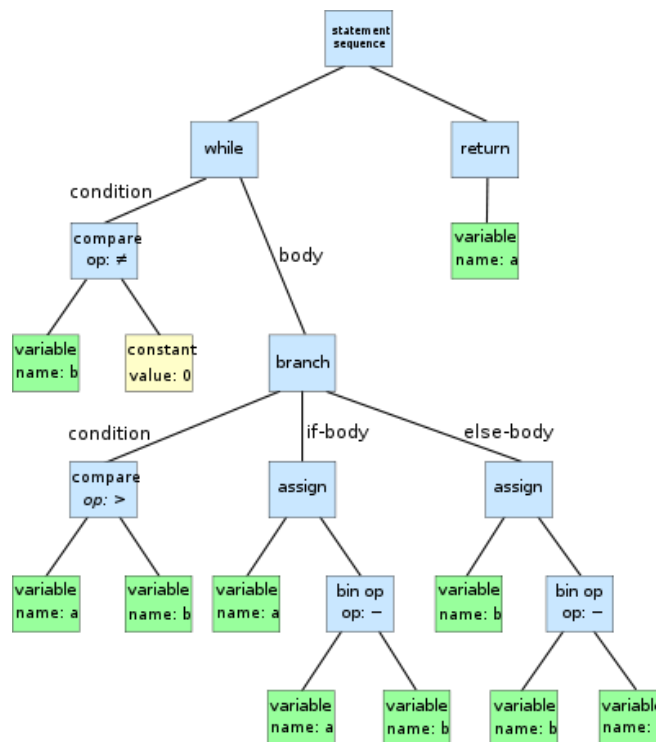


Рисунок 1. AST дерево.

1.2. Описание существующих разработок

Есть несколько вариантов:

1. Оптимальный — если в штате есть программист (а еще лучше — несколько программистов). Поставьте задачу, опишите требования и получите готовый инструмент, заточенный конкретно под ваши задачи. Инструмент можно будет донастраивать и улучшать при необходимости.
2. Воспользоваться готовыми облачными парсерами (есть как бесплатные, так и платные сервисы).
3. Desktopные парсеры — как правило, программы с мощным функционалом и возможностью гибкой настройки. Но почти все — платные.
4. Заказать разработку парсера «под себя» у компаний, специализирующихся на разработке (этот вариант явно не для желающих сэкономить).

Первый вариант подойдет далеко не всем, а последний вариант может оказаться слишком дорогим.

Что касается готовых решений, их достаточно много, и если вы раньше не сталкивались с парсингом, может быть сложно выбрать.

1.2.1. ParserOK.

Программа «Парсер сайтов» разработана для сбора, анализа, выборки, группировки, структуризации, трансформации данных с последующим выводом данных в таблицу Excel в форматах xls* и csv.

Парсер создан на VBA (Visual Basic for Applications) и представлен в виде надстройки для MS Excel, по сути это набор макросов, каждый набор отвечает за выполнение определенных функций при обработке.

Таким образом, для работы программы необходимы: файл надстройки Parser.xla и файл управления надстройкой Name.xlp (Name — имя файла).

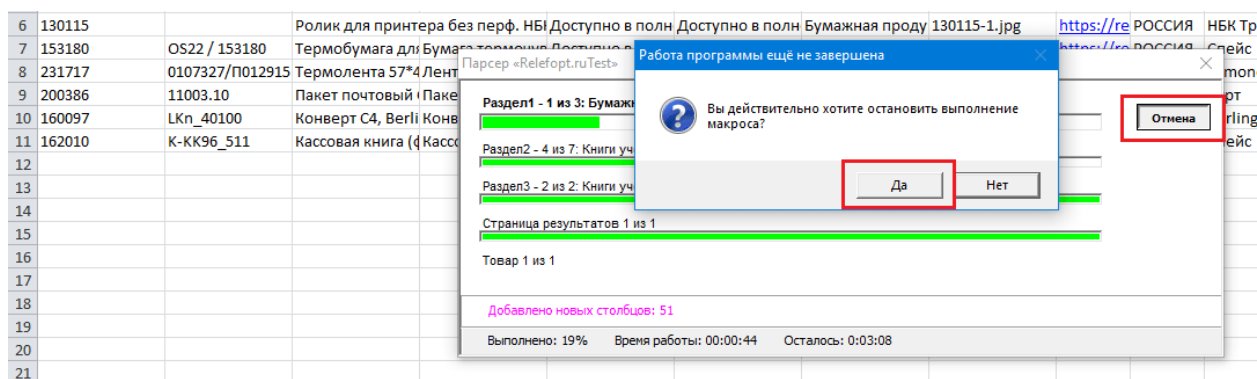


Рисунок 2. Работа с ParserOK

1.2.2. Datacol

Это универсальный парсер, который умеет собирать данные с любого сайта или файла, дополнительно их обработать и сохранять результат работы в файл, базу данных или сразу грузить на ваш сайт.

Перед запуском кампании задайте входные данные - название товаров, прайс лист или ссылки на категории сайта, из которых нужно спарсить данные.

После запуска кампании за ее работой можно следить в блоке “Статистика работы”. Если все настроено верно, первые результаты вскоре появятся внизу программы.

Собранные данные можно дополнительно обрабатывать: переводить, добавлять наценку к цене поставщика, уникализировать текст, переводить в другую валюту и многое другое. Это возможно благодаря плагинам, которые дополняют и расширяют функционал Datacol. Доступные плагины можно найти [здесь](#) и [здесь](#). Если нужного вам плагина нет, его разработку можно заказать у наших специалистов.

По умолчанию, результаты парсинга в базовых кампаниях автоматически сохраняются в EXCEL файл с таким же названием, что и кампания парсинга. Файл можно найти в папке “Мои документы”. При необходимости, в настройках кампании можно изменить как параметры сохранения, так их формат данных. Можно отправить данные на FTP, в базу данных MySQL, экспортировать в потоковом режиме или сохранить все в CSV файл нужной структуры.

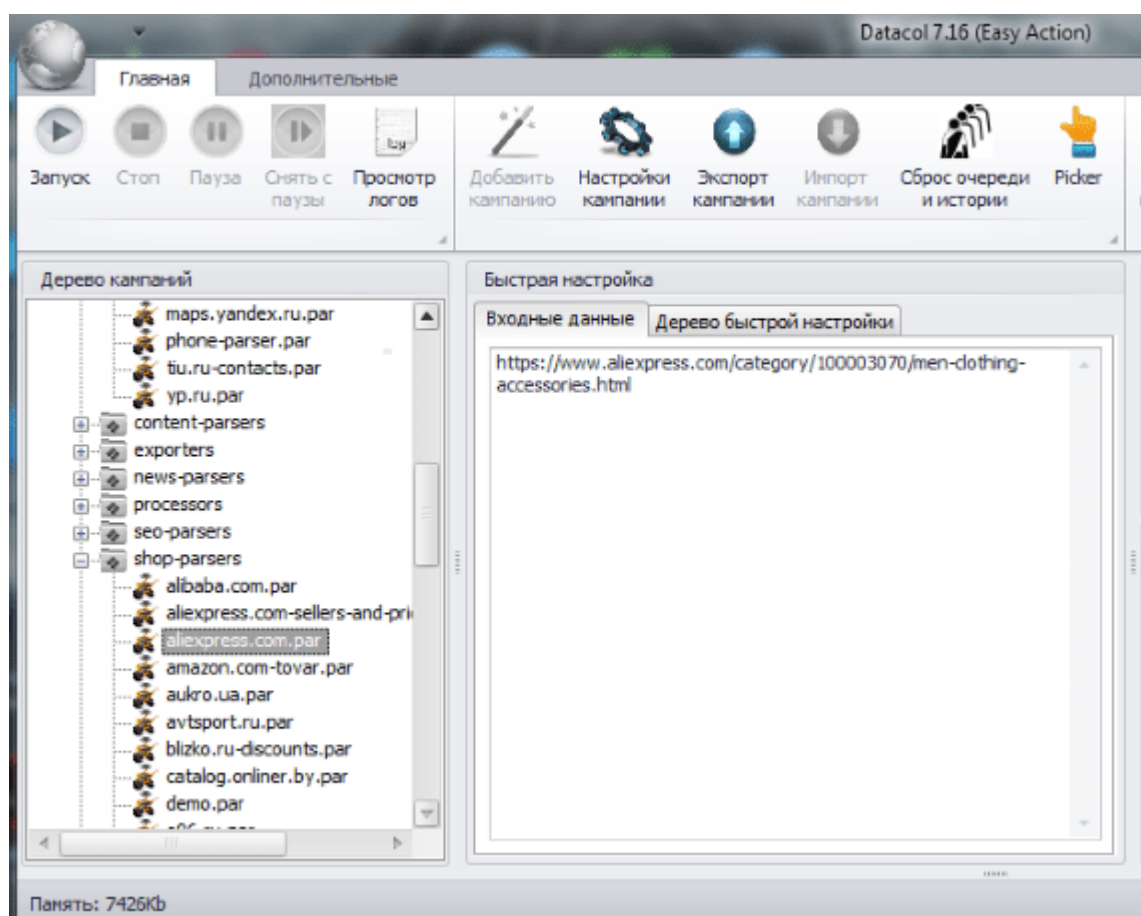


Рисунок 3. Работа в программе Datacol

2. Проектная часть

2.1. Диаграмма прецедентов

В этом разделе представлена диаграмма прецедентов. На диаграмме (Рисунок 4) показаны все возможные функциональные отношения.

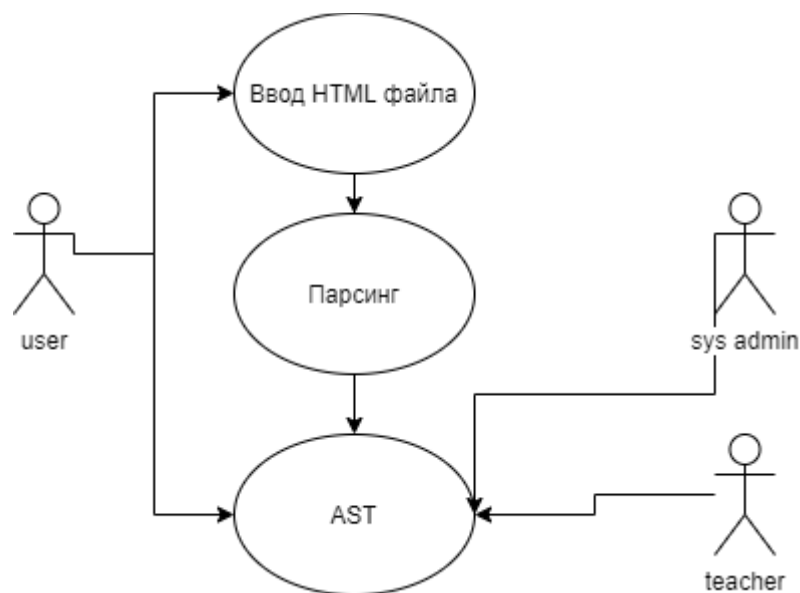


Рисунок 4. Диаграмма прецедентов работы парсера

2.2. Выбор инструментов

При выборе инструментов я исходил из некоторых критериев. Важность критерия выбрана из: низкая, ниже средней, средняя, ниже высокой, высокая.

Таблица 1.

Важность критериев выбора.

Критерий	Участие в корпоративном проекте	Простота сопровождения	Наличие библиотек	Наличие документации на русском языке	Скорость разработки
Важность критерия	Ниже средней	Средняя	Высокая	Ниже высокой	Ниже высокой

Исходя из этих критериев, были сравнены 3 языка программирования от 0 до 10 баллов за критерий.

Таблица 2.

Выбор языка по критериям.

Критерий/Язык программирования	C++	Python	Object Pascal
Участие в корпоративном проекте	5	10	1
Простота сопровождения	2	10	2
Наличие библиотек	5	10	3
Наличие документации на русском языке	7	10	7
Скорость разработки	1	10	3
Итого баллов	20	50	16

По результатам сравнения был выбран язык программирования Python.

2.3. Проектирование сценария

В данном разделе приведен сценарий использования программы пользователем (Рисунок 5).

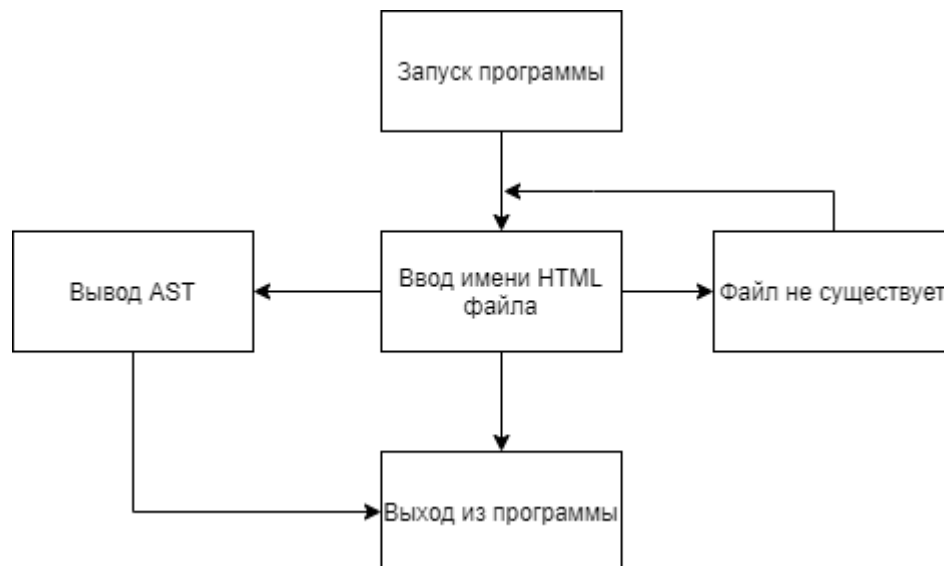


Рисунок 5. Сценарий использования

Пользователь после запуска программы вводит название HTML файла или может выйти из программы.

При выборе выхода программа заканчивает свою работу, при вводе имени начинается проверка на существование файла, если файл не существует, то предлагается ввести имя еще раз, если существует, то выводится сгенерированное AST дерево HTML файла и выходит из программы.

2.4. Диаграмма классов

В данном разделе представлены все классы, использующиеся в проекте, (Рисунок 6).

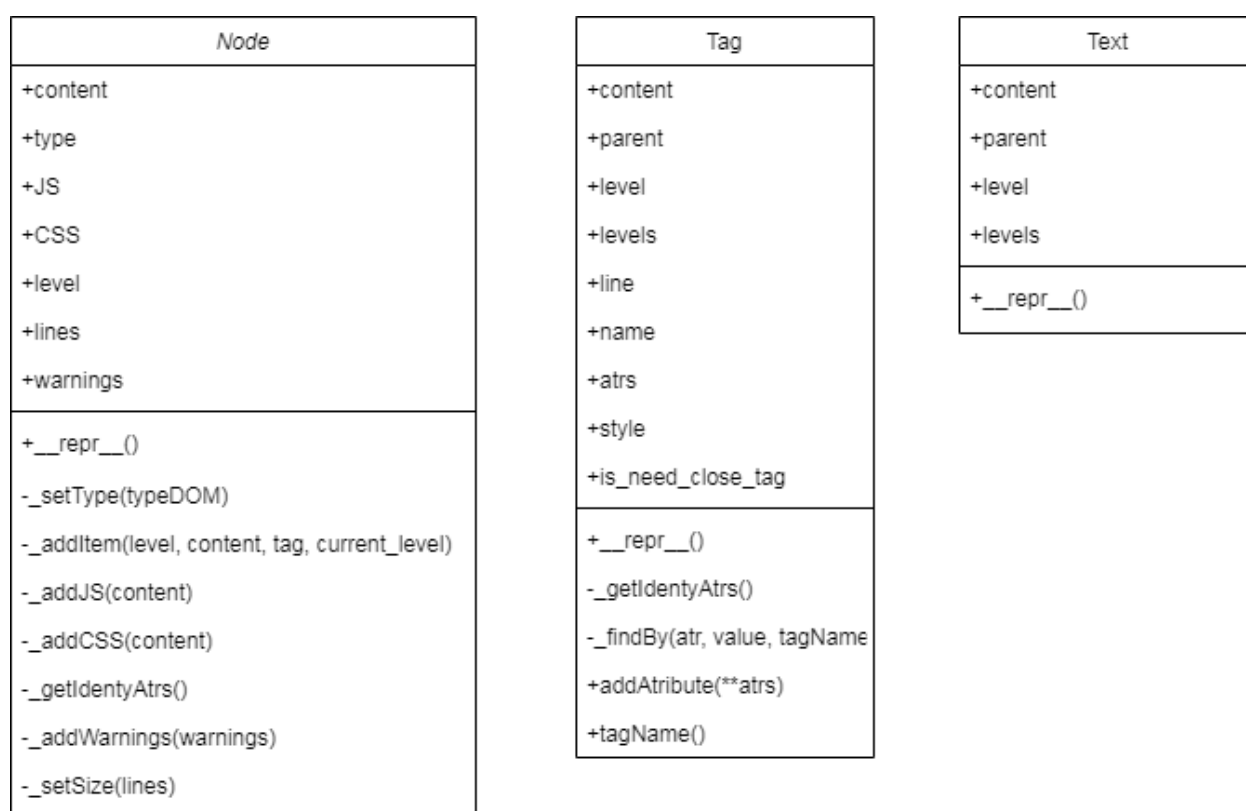


Рисунок 6. Диаграмма классов для проекта

2.5. Описание главного модуля

В главный модуль подключается файл parserHTML, в котором прописаны все связи остальных файлов и функции. Далее в главном модуле находится функция получения AST дерева из имени файла, получение имени файла, проверка ошибок и вывод его.

Листинг 1. Функция получения AST дерева.

```
def getDocument():
    filename = input("\nName of HTML file ('q' - exit): ") # получаем имя
    файла

    if filename == 'q': # выходим при q
        sys.exit()

    while not os.path.isfile(filename) or not 'html' in filename.split('.')
    or len(filename.split('.')) < 2: # пока ошибки, просим еще раз ввести имя
    файла
        if filename == 'q':
            sys.exit()
        if not 'html' in filename.split('.') or len(filename.split('.')) < 2:
            filename = input("\n\033[41m{}\033[40m\n".format("File '" +
            filename + "' is not HTML file.") + "\nPlease enter correct name of HTML file
            ('q' - exit): ")
        else:
            filename = input("\n\033[41m{}\033[40m\n".format("File '" +
            filename + "' is not exist.") + "\nPlease enter correct name of HTML file ('q'
            - exit): ")

    document = parserHTML(filename) #парсируем файл

    if document: # если успешно
        (document, (len_err, err)) = document # то получаем дерево и ошибки
        print("\n\033[42m{}\033[40m\n".format("Parsed completed!"))
        # if len_err: # если есть ошибки, то выводим их
        print("\033[30m\033[43m{}\033[37m\033[40m\n".format(
            f"{len_err} warning in {' '.join(' < ' + e + ' > on line ' +
            str(l) for (e, l) in err)}")
        )
        return document # возвращаем дерево
    else:
        return None
```

Данный метод получает имя файла, открывает его, читает из него текст, передает в лексер и получает из него токены, их же в свою очередь передает в парсер, который возвращает AST дерево.

Листинг 2. Главная функция.

```
if __name__ == "__main__":
    print("HTML Parser v.2.3.1.610 (released 13.05.2020). Created by
    OVOSKOP.")
    print('Type "help" for more information.')

    document = getDocument()
    while not document:
        document = getDocument() # получаем дерево
    print('\n')
    print(document) # выводим его на экран
```

Блок-схема лексера

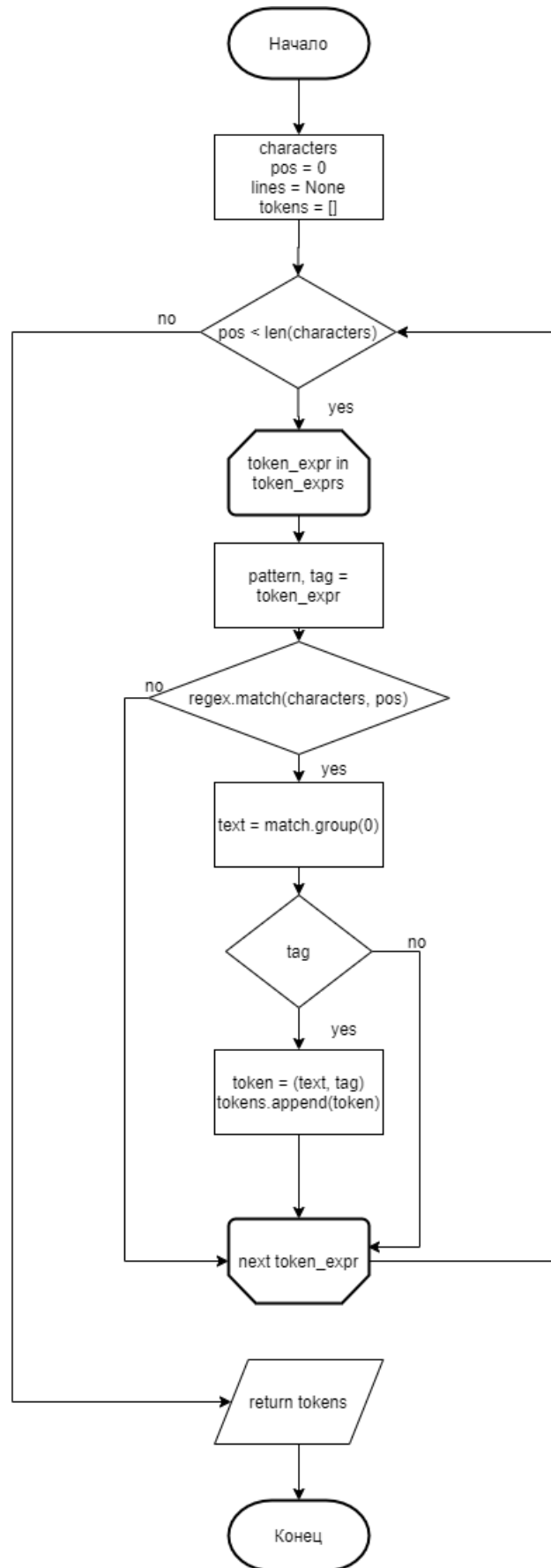


Рисунок 7. Блок схема лексера

2.6. Описание спецификаций к модулям

Разработаны 5 модулей, не включая главный:

- Классы и методы самого парсера
- Функция парсера
- Конфигурации лексера
- Функция лексера
- Подключение лексера и парсера в одну функцию

Структура подключения модулей изображена на Рисунке 8.

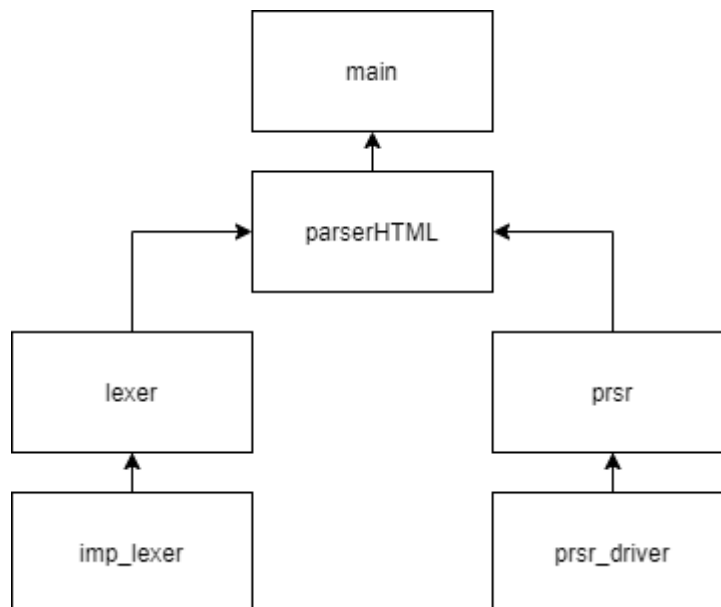


Рисунок 8 .Структура подключения модулей

2.7.Описание модулей

Модуль 1. Классы и методы самого парсера

Прописываются классы Node, Tag, Text. Реализуются поля и методы классов.

Листинг 1. Создание класса Text

```

# Text
class Text:
    def __init__(self, text): # конструктор
        self.content = text # контент тега
  
```



```

        self.parent = None # родительский тег
        self.level = 0 # уровень вложенности тега
        self.levels = []
# список последний ли тег на своем уровне вложенности (для вывода дерева)

    def __repr__(self): # строковое представление объекта
        return self.content

```

Модуль 2. Функция парсера

Прописывается основная функция парсера.

Листинг 2. Функция парсера

```

def parser(tokens):
    buff = []
    level = 0
    err = []
    lines = 1
    doc = Node() # создаем объект дерева
    for token in tokens: # извлекаем по одному токенеу из списка
        # в зависимости от токена, создаем нужный объект и добавляем в дерево
        if token[1] == 'OPEN_TAG':
            tag = Tag(token[0], lines) # создание объекта тега
            doc._addItem(level, tag) # добавление в дерево
            buff.append(tag) # добавление в список тега, чтобы отслеживать
# закрывающиеся
            level += 1 # добавление уровня вложенности
        elif token[1] == 'CLOSE_TAG':
            if buff[-1].tagName() == token[0][0][0]: # проверка закрывающихся
# тегов
                buff.pop()
                level -= 1 # убавление уровня вложенности
            else:
# если предыдущий тег не был закрыт, то ищем ближайший подходящий тег к
# текущему тегу
                while buff[-1].tagName() != token[0][0][0]:
                    err.append((buff[-1].name + buff[-1]._getIdentityAttrs(),
buff[-1].line)) # из неподходящих делаем список ошибок
                    buff.pop()
                    level -= 1

                buff.pop()
                level -= 1
        else:
            if token[1] == 'TAG':
                doc._addItem(level, Tag(token[0], lines,
is_need_close_tag=False)) # создаем элемент тега без закрывающегося тега
            elif token[1] == 'SCRIPT':
                doc._addJS(token[0]) # добавляем элемент скрипта
                doc._addItem(level, Tag(['script'], lines))
            elif token[1] == 'STYLE': # добавляем элемент стилей

```

```

        doc._addCSS(token[0])
        doc._addItem(level, Tag(['style'], lines))
    elif token[1] == 'TYPE': # добавление типа файла
        doc._setType(token[0])
    elif token[1] == 'LINE': # счетчик строк
        lines += 1
    else:
        doc._addItem(level, Text(token[0])) # добавление контента тега

doc._setSize(lines) # добавление размера файла в строках
for item in buff:
    # создание списка ошибок
    err.append((item.name + item._getIdentityAtrs(), item.line))
doc._addWarnings(err) # добавление списка ошибок в объект дерева
return (doc, (len(err), err)) # возвращаем дерево

```

Модуль 3. Конфигурации лексера

В данном модуле прописываются все паттерны и конфиги для лексера.

Листинг 3. Список токенов для лексера

```

# список токенов в виде регулярных выражений
token_exprs = [
    (r'[\t]+' , None),
    (r'\n' , LINE),
    (r'<!--' , COMMENT),
    (r'<script( |\n)*([-\\w;: ]+(=")?[>]*(")?)*(( )?/)?>' ,
SCRIPT),
    (r'<style( |\n)*([-\\w;: ]+(=")?[>]*(")?)*(( )?/)?>' , STYLE),
    (r'<\/[\\w-]*>' , CLOSE_TAG),
    (r'<!DOCTYPE [ \\w.:\\/-\\"]+>' , TYPE),
    (r'(([^<>\\s])|( )|( )|( )|( ))+' ,
CONTENT),

    (r'<((area)|(base)|(br)|(col(?:g))|(command)|(embed)|(hr)|(img)|(input)|(keyg
en)|(link)|(meta)|' +
        r'(param)|(source)|(track)|(wbr)) ( |\n)*([-\\w;:
]+(=")?[>]*(")?)*(( )?/)?>' , TAG),
    (r'<([\\w-]+) ( |\n)*([-\\w;: ]+(=")?[>]*(")?)*(( )?/)?>' ,
OPEN_TAG),
]

```

Модуль 4. Функция лексера

В данном модуле реализована функция лексера, которая создает из текста HTML многомерный массив, который в последствии превращается в AST

Листинг 4. Функция лексирования тегов

```
# функция лексинга тегов
def lexTag(characters, token_exprs):
    pos = 0
    lines = 0
    tokens = []
    while pos < len(characters): # пока позиция меньше длины текста
        match = None
        # извлекаем по одному паттерны и ищем соответствия в коде сайта
        for token_expr in token_exprs:
            pattern, tag = token_expr
            regex = re.compile(pattern)
            match = regex.match(characters, pos)
            if match:
                text = match.group(0)
                if tag:
                    if tag == 'LINE':
                        lines += 1
                    else:
                        # если находим, то добавляем в список
                        token = (text, tag)
                        tokens.append(token)
                        break
            if not match: # если нет, то выводим ошибку
                print("\n\033[41m{}\033[40m".format('Invalid tag: "%s" ' %
(characters[pos:pos+10:])) + 'in pos: %s' % str(pos))
                return None
        else:
            pos = match.end(0)
    return (tokens, lines)
```

Модуль 5. Подключение лексера и парсера в одну функцию

В данном модуле все функции соединяются в одну, которая получает текст HTML, а возвращает AST

Листинг 5. Функция полного парсинга HTML и отслеживание ошибок

```
def parserHTML(filename):
    characters = None
    if filename: # если имя файла корректное, то
        for enc in encoding:
            try:
                # пытаемся открыть файл в разных кодировках
                file = open(filename, encoding=enc)
                characters = file.read()
                file.close()
            # если ошибка, игнорируем ее
            except (UnicodeDecodeError, LookupError):
```

```

        pass
    else: # иначе выходим из цикла
        break
document = ""
if characters: # если файл удалось прочитать, то
    tokens = imp_lex(characters) #лекслируем файл
    if tokens: # если успешно, то
        document = parser(tokens) #парсируем файл
if document != "": # если парсинг успешный, возвращает AST
    return document
print("\n\033[41m{}\033[40m\n".format("Parsed not completed! Invalid
File!")) # иначе выводим сообщение
return None # и возвращает None

```

2.8. Описание тестовых наборов модулей

В каждом модуле требуется вводить определенные значения. Там, где нужно ввести имя файла, идет проверка на существование такого файла, после идет проверка на правильность кодировки, далее на валидность HTML файла лексером, а после парсером.

Тест 1. Ввод неправильного имени

Если файл не существует, то вызовется ошибка (рис. 8), и потребуются заново писать имя файла.

```

Name of HTML file ('q' - exit): 35.html
File '35.html' is not exist.

```

Рисунок 9. Ошибка при вводе неверного пути к файлу

Тест 2. Ввод файла в неправильной кодировке

Если файл в неправильной кодировке, то вызовется ошибка, что файл не корректный

```

HTML Parser v.2.3.1.610 (released 13.05.2020). Created by OVOSKOP.
Type "help" for more information.
Name of HTML file ('q' - exit): 4.html
Parsed not completed! Invalid File!

```

Рисунок 10. Ошибка при вводе файла в неправильной кодировке

Тест 3. Ввод файла с ошибкой

При вводе файла с ошибкой, парсер выдаст предупреждение об ошибке, в каком теге и на какой строке

```
HTML Parser v.2.3.1.610 (released 13.05.2020). created by OVOSKOP.
Type "help" for more information.

Name of HTML file ('q' - exit): 3.html

Parsed completed!

1 warning in < body > on line 5
```

Рисунок 11. Ввод файла с ошибкой

2.9. Описание применения средств отладки

В ходе написания курсового проекта при попытке запустить программу были получены ошибки:

```
E:\Игорь(все нужное)\учеба\2019-2020\прикладное программирование\c++\курсовой проект>python main.py
Traceback (most recent call last):
  File "main.py", line 21, in <module>
    from parserHTML import *
  File "E:\Игорь(все нужное)\учеба\2019-2020\прикладное программирование\c++\курсовой проект\parserHTML.py", line 9
    def parserHTML(filename)
                                ^
SyntaxError: invalid syntax
```

Рисунок 12. Ошибки

При проверке кода были исправлены найденные ошибки, в результате при запуске программы ошибок не было:

```
E:\Игорь(все нужное)\учеба\2019-2020\прикладное программирование\c++\курсовой проект>python main.py
HTML Parser v.2.3.1.610 (released 13.05.2020). created by OVOSKOP.
Type "help" for more information.

Name of HTML file ('q' - exit):
```

Рисунок 13. Сообщение об ошибках

2.10. Анализ оптимальности использования памяти и быстродействия

Сложность нерекурсивного алгоритма равна $O(n)$, соответственно, сложность алгоритма добавления тега в AST равна $O(n)$, что при очень больших кодах сайта заметно тормозит работу программы.

3. Эксплуатационная часть

3.1.Руководство оператора

3.1.1. Назначение программы

Разработанное приложение производит парсинг HTML кода и выводит его в виде АСД. Благодаря АСД воспринимать HTML код становится намного проще. В будущем этот парсер можно использовать в более крупных разработках, например, браузеров.

3.1.2. Условия выполнения программы

Операционная система: Windows 98/Vista/XP/7/8/10.

Процессор (CPU): Любой.

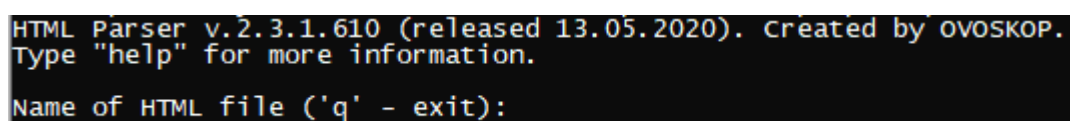
Оперативная память (RAM): 512 МБ.

Видеоадаптер: Любой.

Свободное место на жёстком диске: ~10 Мб.

3.1.3. Выполнение программы

При запуске программы в консоли с помощью интерпретатора python (main.py) будет предложено ввести имя файла:



```
HTML Parser v.2.3.1.610 (released 13.05.2020). created by OVOSKOP.  
Type "help" for more information.  
Name of HTML file ('q' - exit):
```

Рисунок 14. Запуск программы

Нужно будет указать имя HTML файла в формате {filename.html}. Если необходимо закрыть программу на этом шаге, то наберите "q".

При успешном выполнении программы вы увидите надпись:

```
Name of HTML file ('q' - exit): html/html.html
Parsed completed!
```

Рисунок 15. Успешный парсинг

При наличии ошибок в коде, вы увидите надпись с количеством ошибок и списком тегов.

```
Name of HTML file ('q' - exit): html/html.html
Parsed completed!
1 warning in <div#qwe> on line 7
```

Рисунок 16. Вывод списка ошибок

При успешном парсинге данных, в консоль будет выведено АСД.

```
html
|_ head
|   |_ link
|   |_ body
|       |_ div#qwe
|           |_ a.bad
|               |_ hello1
|               |_ div#ast
|                   |_ AST
|                   |_ sCOOL
|               |_ div#i.i.asd
|                   |_ hello2
|               |_ input
|               |_ div.i
|                   |_ hello3
```

Рисунок 17. Вывод АСД

3.1.4. Сообщение оператору

Программа main.py выдаст сообщение об ошибке, показанное на рис. 18

```
HTML Parser v.2.3.1.610 (released 13.05.2020). Created by OVOSKOP.  
Type "help" for more information.  
Name of HTML file ('q' - exit): 3.html  
Illegal character: " " in pos: 53  
Parsed not completed! Invalid File!
```

Рисунок 18. Ошибка

ПРИЧИНА. В коде HTML присутствует символ, не прописанный в темплейтах лексера.

ДЕЙСТВИЯ ПРОГРАММЫ. Программа не парсит данный сайт.

ДЕЙСТВИЯ ОПЕРАТОРА. Добавить соответствующий символ в темплейты лексера.

3.2. To-Do лист

- Создать интерактивную консольную программу для работы с HTML и модифицированием его

Заключение

В результате выполнения курсовой работы был разработан парсер HTML. Данная программа имеет потенциал для будущих разработок и ее можно будет использовать для более крупных проектов. Разработанная программа выполнена в соответствии с требованиями технического задания.

Список литературы и интернет-источников

1. Что такое Datacol? [1.2.2]

<http://web-data-extractor.net/datacol-s-chego-nachat/>

2. Парсер сайтов [1.2.1]

<https://parserok.ru/>

3. 30+ парсеров для сбора данных с любого сайта [1]

<https://habr.com/ru/company/click/blog/494020/#:~:text=%D0%9F%D0%B0%D1%80%D1%81%D0%B5%D1%80%20%E2%80%94%D1%8D%D1%82%D0%BE%20%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0%2C%20%D1%81%D0%B5%D1%80%D0%B2%D0%B8%D1%81%20%D0%B8%D0%BB%D0%B8,%D0%A6%D0%B5%D0%BD%D1%8B.>

4. Абстрактное синтаксическое дерево [1]

https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D0%BE%D0%B5_%D1%81%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE

5. HTML [1]

<https://ru.wikipedia.org/wiki/HTML>

Приложение 1. Код главного модуля.

```

## HTML_PARSER V.2.3.1.610 (Count Lines, Fixed bugs, Correct output of
warnings)
##
## DEVELOPER: OVOSKOP
##
## COPYRIGHT. ALL RIGHT RESERVED.
##
## CLASSES:
##
## TAG - Object of Tag
## Node - Document Object Model or Node of Tags
##
##

import sys
import os.path
import re
import inspect
import ctypes

from parserHTML import *

# подключаем ANSI в консоль
kernel32 = ctypes.windll.kernel32
kernel32.SetConsoleMode(kernel32.GetStdHandle(-11), 7)

def getDocument():
    # получаем имя файла
    filename = input("\nName of HTML file ('q' - exit): ")

    if filename == 'q': # выходим при q
        sys.exit()

    # пока ошибки, просим еще раз ввести имя файла
    while not os.path.isfile(filename) or not 'html' in filename.split('.')
    or len(filename.split('.')) < 2:
        if filename == 'q':
            sys.exit()
        if not 'html' in filename.split('.') or len(filename.split('.')) < 2:
            filename = input("\n\033[41m{}\033[40m\n".format("File '" +
filename + "' is not HTML file.") + "\nPlease enter correct name of HTML file
('q' - exit): ")
        else:
            filename = input("\n\033[41m{}\033[40m\n".format("File '" +
filename + "' is not exist.") + "\nPlease enter correct name of HTML file ('q'
- exit): ")

    document = parserHTML(filename) #парсируем файл

    if document: # если успешно
        (document, (len_err, err)) = document # то получаем дерево и ошибки
        print("\n\033[42m{}\033[40m\n".format("Parsed completed!"))

```

```

        # if len_err: # если есть ошибки, то выводим их
            print("\033[30m\033[43m{}\033[37m\033[40m\n".format(
                f"{len_err} warning in {' '.join(' < ' + e + ' > on line ' +
str(l) for (e, l) in err)}")
            )
            return document # возвращаем дерево
    else:
        return None

if __name__ == "__main__":

    print("HTML Parser v.2.3.1.610 (released 13.05.2020). Created by
OVOSKOP.")
    print('Type "help" for more information.')

    document = getDocument()
    while not document:
        document = getDocument() # получаем дерево
    print('\n')
    print(document) # выводим его на экран

```

Приложение 2. Код модуля parserHTML.

```

from imp_lexer import *
from prsr import *

# список доступных кодировок
encoding = [
    'utf-8',
    'windows-1251',
]

def parserHTML(filename):
    characters = None
    if filename: # если имя файла корректное, то
        for enc in encoding:
            # пытаемся открыть файл в разных кодировках
            try:
                file = open(filename, encoding=enc)
                characters = file.read()
                file.close()
            # если ошибка, игнорируем ее
            except (UnicodeDecodeError, LookupError):
                pass
            else: # иначе выходим из цикла
                break
    document = ""
    if characters: # если файл удалось прочитать, то
        tokens = imp_lex(characters) #лекслируем файл
        if tokens: # если успешно, то
            document = parser(tokens) #парсируем файл
    if document != "": # если парсинг успешный, возвращает AST
        return document
    print("\n\033[41m{}\033[40m\n".format("Parsed not completed! Invalid
File!")) # иначе выводим сообщение
    return None # и возвращает None

```

Приложение 3. Код модуля imp_lexer.

```

import lexer

# список всех элементов языка
TAG          = 'TAG'
OPEN_TAG     = 'OPEN_TAG'
CLOSE_TAG    = 'CLOSE_TAG'
TYPE         = 'TYPE'
CONTENT      = 'CONTENT'
SCRIPT       = 'SCRIPT'
STYLE        = 'STYLE'
COMMENT      = 'COMMENT'
LINE         = 'LINE'

TAG_NAME     = 'TAG_NAME'
ATTRIBUTE    = 'ATTRIBUTE'
ATTRIBUTE_NAME = 'ATTRIBUTE_NAME'
VALUE        = 'VALUE'

# список токенов в виде регулярных выражений
token_exprs = [
    (r'[ \t]+' , None),
    (r'\n' , LINE),
    (r'<!--' , COMMENT),
    (r'<script( |\\n)*([\\w;: ]+(=")?[>]*(")?)*(( )?/)?>' ,
SCRIPT),
    (r'<style( |\\n)*([\\w;: ]+(=")?[>]*(")?)*(( )?/)?>' , STYLE),
    (r'<[/[\\w-]*>' , CLOSE_TAG),
    (r'<!DOCTYPE [ \\w.:\\/-\\"]+>' , TYPE),
    (r'(([^<>\\s])|( )|( )|( )|( ))+' ,
CONTENT),

    (r'<((area)|(base)|(br)|(col(?:g))|(command)|(embed)|(hr)|(img)|(input)|(keyg
en)|(link)|(meta)|' +
        r'(param)|(source)|(track)|(wbr))( |\\n)*([\\w;:
]+(=")?[>]*(")?)*(( )?/)?>' , TAG),
    (r'<([\\w-]+)( |\\n)*([\\w;: ]+(=")?[>]*(")?)*(( )?/)?>' ,
OPEN_TAG),
]

# токены для тегов
token_tag = [
    [
        (r'[ \t<>/]+' , None),
        (r'\n' , LINE),
        (r'(?<=) [\\w-]+' , TAG_NAME),
        (r'(?<=) [/] [\\w-]+' , TAG_NAME),
        (r'["\\w;:-]+(="|\\')?[^\\'"]*(|\\')?>' , ATTRIBUTE),
    ],
]

# функция лексинга файла, передаем в нее все токены
def imp_lex(characters):
    return lexer.lex(characters, token_exprs, token_tag)

```

Приложение 4. Код модуля prsr.

```

from prsr_driver import *
import sys

# функция получает токены из лексера в виде ((text), tag)
def parser(tokens):
    buff = []
    level = 0
    err = []
    lines = 1
    doc = Node() # создаем объект дерева
    for token in tokens: # извлекаем по одному токенеу из списка
        # в зависимости от токена, создаем нужный объект и добавляем в дерево
        if token[1] == 'OPEN_TAG':
            tag = Tag(token[0], lines) # создание объекта тега
            doc._addItem(level, tag) # добавление в дерево
            # добавление в список тега, чтобы отслеживать закрывающиеся
            buff.append(tag)
            level += 1 # добавление уровня вложенности
        elif token[1] == 'CLOSE_TAG':
            # проверка закрывающихся тегов
            if buff[-1].tagName() == token[0][0][0]:
                buff.pop()
                level -= 1 # убавление уровня вложенности
            else:
                # если предыдущий тег не был закрыт, то ищем ближайший подходящий тег к
                # текущему тегу
                while buff[-1].tagName() != token[0][0][0]:
                    err.append((buff[-1].name + buff[-1]._getIdentyAtrs(),
buff[-1].line)) # из неподходящих делаем список ошибок
                    buff.pop()
                    level -= 1

                buff.pop()
                level -= 1
        else:
            if token[1] == 'TAG':
                doc._addItem(level, Tag(token[0], lines,
is_need_close_tag=False)) # создаем элемент тега без закрывающегося тега
            elif token[1] == 'SCRIPT':
                doc._addJS(token[0]) # добавляем элемент скрипта
                doc._addItem(level, Tag(['script'], lines))
            elif token[1] == 'STYLE': # добавляем элемент стилей
                doc._addCSS(token[0])
                doc._addItem(level, Tag(['style'], lines))
            elif token[1] == 'TYPE': # добавление типа файла
                doc._setType(token[0])
            elif token[1] == 'LINE': # счетчик строк
                lines += 1
            else:
                doc._addItem(level, Text(token[0])) # добавление контента тега

    doc._setSize(lines) # добавление размера файла в строках

```



```
for item in buff:
    # создание списка ошибок
    err.append((item.name + item._getIdentityAtrs(), item.line))
doc._addWarnings(err) # добавление списка ошибок в объект дерева
return (doc, (len(err), err)) # возвращаем дерево
```

Приложение 5. Код модуля prsr_driver.

```
# ***** CLASSES *****

# Text
class Text:
    def __init__(self, text): # конструктор
        self.content = text # контент тега
        self.parent = None # родительский тег
        self.level = 0 # уровень вложенности тега
# список последний ли тег на своем уровне вложенности (для вывода дерева)
        self.levels = []

    def __repr__(self): # строковое представление объекта
        return self.content

# Tag
class Tag:
    def __init__(self, args, line, is_need_close_tag = True): # конструктор
        self.content = [] # контент тега
        self.parent = None # родительский тег
        self.level = 0 # уровень вложенности тега
        self.levels = [] # список последний ли тег на своем уровне
вложенности (для вывода дерева)
        self.line = line # номер строки тега
        self.name = args[0][0] # имя тега
        self.attrs = {} # словарь атрибутов
        self.style = {} # словарь стилей
        # требуется ли закрывающий тег
        self.is_need_close_tag = is_need_close_tag
        for atr in args[1:]: # добавление атрибутов
            if atr[1] == "ATTRIBUTE":
                atr_value = None
                if '=' in atr[0]:
                    atr_name = atr[0].split('=')[0].replace(' ', '')
                    atr_value = atr[0].split('=')[1]
                else:
                    atr_name = atr[0].replace(' ', '')
                if atr_value:
                    if atr_value[0] == '"' and atr_value[-1] == '"' or
atr_value[0] == "'" and atr_value[-1] == "'":
                        atr_value = atr_value[1:-1]
                    self.addAttribute(**{atr_name: (atr_value if atr_value else
""))})

    def __repr__(self): # строковое представление объекта
        tabs = ""
        if self.level >= 1:
            tabs = ''.join(['|' if self.levels[i] else ' ' for i in
range(self.level)])

        line = f"{self.name + self._getIdentyAttrs()}"
        for item in self.content:
            typeElem = str(type(item)).split("'")[1].split(".")
```

```

        typeElem = typeElem[1] if len(typeElem) > 1 else typeElem[0]
        item.level = self.level + 1
        if self.content.index(item) < len(self.content) - 1:
            item.levels = [*self.levels, 1]
        else:
            item.levels = [*self.levels, 0]
        line += f"\n{tabs}|_____" + '%r' % item
    self.level = 0
    self.levels = []
    return line

def _getIdentityAtrs(self): # функция превращения атрибутов в цельную
строку
    identity = ""
    if 'id' in self.atrs:
        for ids in self.atrs['id']:
            if len(ids) > 0:
                identity += "#" + ids
    if 'class' in self.atrs:
        for classes in self.atrs['class']:
            if len(classes) > 0:
                identity += "." + classes

    return identity

def _findBy(self, atr = None, value = None, tagName = None): # поиск
элемента по необходимым критериям
    elems = []
    for elem in self.content:
        typeElem = str(type(elem)).split("'")[1].split(".")
        typeElem = typeElem[1] if len(typeElem) > 1 else typeElem[0]
        # print(typeElem, value)
        if typeElem == "Tag":
            if not tagName:
                if atr in elem.atrs:
                    if value:
                        if elem.atrs[atr] == value or \
                            value in elem.atrs[atr]:
                            elems.append(elem)
                    else:
                        elems.append(elem)
            elems.extend(elem._findBy(atr, value))
        else:
            if elem.name == tagName:
                elems.append(elem)
            elems.extend(elem._findBy(tagName=tagName))
    return elems

def addAttribute(self, **atrs): # добавить атрибуты тегу
    for atr, value in atrs.items():
        if atr == 'style':
            pass
        elif atr == "class" or atr == "id":
            self.atrs[atr].extend(value.split(" ")) if (atr in self.atrs)
    else self.atrs.update({atr: value.split(" ")})

```

```

        else:
            self.attrs[atr] = value
        return True

    def tagName(self):
        return self.name

# класс дерева
class Node:
    def __init__(self, content = None):
        self.content = [] # контент дерева
        self.type = None # тип файла
        self.JS = [] # скрипты
        self.CSS = [] # стили
        self.level = -1 # уровень вложенности
# список последний ли тег на своем уровне вложенности (для вывода дерева)
        self.lines = 0
        self.warnings = None # список ошибок
        if content:
            self.content.append(content)

    def __repr__(self): # строковое представление объекта
        line = ""
        for item in self.content:
            line += f"{str(item)}\n"
        return line

    def _setType(self, typeDOM): # установка типа файла
        self.type = typeDOM.split(" ")[1].split(">")[0]

# добавление элемента в дерево
    def _addItem(self, level, content, tag = None, current_level = 0):
        if not tag:
            tag = self
        if level == current_level:
            typeElem = str(type(content)).split("'")[1].split(".")
            typeElem = typeElem[1] if len(typeElem) > 1 else typeElem[0]

            if typeElem == "Tag":
                content.parent = tag

            tag.content.append(content)
            return tag
        tag.content[len(tag.content) - 1] = self._addItem(level, content,
tag.content[len(tag.content) - 1], current_level + 1)
        return tag

    def _addJS(self, content): # добавление скриптов
        self.JS.append(content)

    def _addCSS(self, content): # добавление стилей
        self.CSS.append(content)

# функция превращения атрибутов в цельную строку
    def _getIdentyAttrs(self):

```

```
return ""

def _addWarnings(self, warnings): # добавление списка ошибок
    self.warnings = warnings

def _setSize(self, lines): # установка размера
    self.lines = lines
```

Приложение 6. Код модуля lexer.

```

import re

# функция лексинга файла
def lex(characters, token_exprs, sub_token_exprs=None):
    pos = 0
    lines = None
    tokens = []
    while pos < len(characters): # пока позиция меньше длина файла
        l = 0
        match = None
        # извлекаем из списка паттернов паттерны по одному и проверяет подходит ли
        for token_expr in token_exprs:
            pattern, tag = token_expr
            regex = re.compile(pattern)
            match = regex.match(characters, pos)
            if match: # если нашли соответствие, то
                text = match.group(0) # получает найденный текст
                if tag: # в зависимости от тега, разные действия
                    if (tag == "TAG" or tag == "CLOSE_TAG" or tag ==
"OPEN_TAG") and sub_token_exprs:
                        (text, l) = lexTag(text, sub_token_exprs[0])
                        # если теги, то лексируем их отдельно
                    # если скрипты и стили, то ищем закрывающий тег и получаем их контент
                    if tag == "SCRIPT" or tag == "STYLE":
                        start = match.end(0)
                        regex = re.compile(r'<\/(script|style)>')
                        match = regex.search(characters, match.end(0))
                        if match:
                            end = match.start(0)
                            text = characters[start:end:]
                            l = text.count('\n')
                    if tag == 'COMMENT': # игнорируем комменты
                        regex = re.compile(r'-->')
                        start = match.end(0)
                        match = regex.search(characters, start)
                        if match:
                            end = match.start(0)
                            text = characters[start:end:]
                            l = text.count('\n')
                            for i in range(l):
                                tokens.append((' \n', 'LINE'))
                                break
                            break
                        token = (text, tag)
                        tokens.append(token)
                        for i in range(l):
                            tokens.append((' \n', 'LINE'))
                            # добавляем все строки, которые находим в файле
                        break
                    if not match:
                        # если не находим, то выводим ошибку о незарегистрированном символе
                        print("\n\033[41m{}\033[40m\n".format('Illegal character: "%s" '
% (characters[pos]) + 'in pos: %s' % str(pos)))

```

```

        return None
    else:
        pos = match.end(0)
    return tokens # возвращаем токены

# функция лексинга тегов
def lexTag(characters, token_exprs):
    pos = 0
    lines = 0
    tokens = []
    while pos < len(characters): # пока позиция меньше длины текста
        match = None
        for token_expr in token_exprs:
            # извлекаем по одному паттерны и ищем соответствия
            pattern, tag = token_expr
            regex = re.compile(pattern)
            match = regex.match(characters, pos)
            if match:
                text = match.group(0)
                if tag:
                    if tag == 'LINE':
                        lines += 1
                    else:
                        token = (text, tag)
                        # если находим то добавляем в список
                        tokens.append(token)
                break
        if not match: # если нет, то выводим ошибку
            print("\n\033[41m{}\033[40m".format('Invalid tag: "%s" ' %
(characters[pos:pos+10:])) + 'in pos: %s' % str(pos)))
            return None
    else:
        pos = match.end(0)
    return (tokens, lines)

```