ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Факультет программной инженерии и компьютерной техники

Дисциплина «Низкоуровневое программирование»

Отчет к лабораторной работе №2

Выполнил:
Студент группы P33302
Владыкина Карина Кирилловна
Проверил:
Кореньков Юрий Дмитриевич

Санкт-Петербург
2023 год

Вариант 6: язык запросов Gremlin

Репозиторий:
https://github.com/1KarinaV/llp2

Задание:

Использовать средство синтаксического анализа по выбору, реализовать модуль для разбора некоторого достаточного подмножества языка запросов по выбору в соответствии
с вариантом формы данных. Должна быть обеспечена возможность описания команд создания, выборки, модификации и удаления элементов данных.

Порядок выполнения:

1. Изучить выбранное средство синтаксического анализа
    a. Средство должно поддерживать программный интерфейс совместимый с
    языком C
    b. Средство должно параметризоваться спецификацией, описывающий синтаксическую структуру разбираемого языка
    c. Средство может функционировать посредством кодогенерации и/или подключения необходимых для его работы дополнительных библиотек
    d. Средство может быть реализовано с нуля, в этом случае оно должно быть
    основано на обобщённом алгоритме, управляемом спецификацией
2. Изучить синтаксис языка запросов и записать спецификацию для средства синтаксического анализа
    a. При необходимости добавления новых конструкций в язык, добавить нужные синтаксические конструкции в спецификацию (например, сравнения в GraphQL)
    b. Язык запросов должен поддерживать следующие возможности:
        • Условия
            o На равенство и неравенство для чисел, строк и булевских значений
            o На строгие и нестрогие сравнения для чисел
            o Существование подстроки
        • Логическую комбинацию произвольного количества условий и булевских значений
        • В качестве любого аргумента условий могут выступать литеральные
        значения (константы) или ссылки на значения, ассоциированные с элементами данных (поля, атрибуты, свойства)
        • Разрешение отношений между элементами модели данных любых

условий над сопрягаемыми элементами данных
• Поддержка арифметических операций и конкатенации строк не обязательна

c. Разрешается разработать свой язык запросов с нуля, в этом случае необходимо показать отличие основных конструкций от остальных вариантов (за исключением типичных выражений типа инфиксных операторов сравнения)

3. Реализовать модуль, использующий средство синтаксического анализа для разбора
языка запросов

a. Программный интерфейс модуля должен принимать строку с текстом запроса и возвращать структуру, описывающую дерево разбора запроса или
сообщение о синтаксической ошибке

b. Результат работы модуля должен содержать иерархическое представление
условий и других выражений, логически представляющие собой иерархически организованные данные, даже если на уровне средства синтаксического анализа для их разбора было использовано линейное представление

4. Реализовать тестовую программу для демонстрации работоспособности созданного
модуля, принимающую на стандартный ввод текст запроса и выводящую на стандартный вывод результирующее дерево разбора или сообщение об ошибке

5. Результаты тестирования представить в виде отчёта, в который включить:

a. В части 3 привести описание структур данных, представляющих результат
разбора запроса

b. В части 4 описать, какая дополнительная обработка потребовалась для результата разбора, представляемого средством синтаксического анализа, чтобы сформировать результат работы созданного модуля

c. В части 5 привести примеры запросов для всех возможностей из п.2.b и
результирующий вывод тестовой программы, оценить использование разработанным модулем оперативной памяти

Структура проекта:

Лексический анализ (lexer.l) происходит с помощью flex. Символы превращаются в токены. Некоторые токены преобразуются в enum'ы или base_value_t:

```
typedef union {
   int integer_value;
   bool bool_value;
   char* string_value;
   float float_value;
} base_value_t;
```

Синтаксический анализатор (parser.y), написанный на bison, принимает токены и строит синтаксическое дерево с помощью вспомогательных структур данных, описанных в vector.h и описываемых типов, которые задаются через attribute.h, request.h, schema.h, select.h, value.h, statement.h.

attribute.h

```
typedef enum {
   INT,
   BOOL,
   FLOAT,
   STR,
   REFERENCE
} attr_type_t;

typedef struct {
   char *attr_name;
   attr_type_t type;
   char *schema_ref_name;
} attribute_declaration_t;

typedef struct {
   char *attr_name;
   attr_type_t type;
   base_value_t value;
} attr_value_t;
```

schema.h:

```
typedef struct {
   char *schema_name;
   vector *attribute_declarations;
} add_schema_t;

typedef struct {
   char *schema_name;
} delete_schema_t;

typedef struct {
   char* schema_name;
   vector *attribute_values;
} add_node_t;
```

select.h:

```
typedef enum {
   EQUAL,
   GREATER,
   GREATER_EQUAL,
   LESS,
   LESS_EQUAL,
   NOT_EQUAL,
   LIKE
} select_opt_t;

typedef struct {
```

```
   char *attr_name;
   select_opt_t option;
   attr_type_t type;
   base_value_t value;
} select_cond_t;
```

statement.h:

```
typedef enum {
   SELECT,
   OUT,
   DELETE
} action_t;

typedef struct {
   action_t type;
   union {
       vector *conditions;
       char *attr_name;
   };
} statement_t;
```

value.h:

```
typedef union {
   int integer_value;
   bool bool_value;
   char* string_value;
   float float_value;
} base_value_t;
```

В конце синтаксического анализа все сводится к структуре request_t, и из которой берется синтаксическое дерево, посредством которого при помощи спуска по дереву создается DOM формата JSON согласно выданному варианту .

```
typedef enum {
    UNDEFINED,
    OPEN,
    CREATE,
    CLOSE,
    ADD_SCHEMA,
    DELETE_SCHEMA,
    ADD_NODE,
    SELECT_REQ
} request_type_t;

typedef struct {
   char *filename;
} file_work_struct_t;

typedef struct {
   request_type_t type;
   char* schema_name;
   union {
```

```
        file_work_struct_t file_work;
        add_schema_t add_schema;
        delete_schema_t delete_schema;
        add_node_t add_node;
        vector *statements;
    };
} request_t;
```

Для вывода результатов используется консольная печать, использующая флаги JSON_C_TO_STRING_SPACED , JSON_C_TO_STRING_PRETTY, и также создается файл transfer_data.json, который служит форматом системы хранения данных между клиентом и сервером.

Синтаксис языка запроса Gremlin:

Типы запросов:

- создание файла:

```
create("filename.txt");
```

- открытие существующего файла:

```
open("filename.txt");
```

- закрытие файла:

```
close();
```

- добавление схемы:

```
addSchema("schema_name", "attr_name1", <attr_type1>, "attr_name2",
<attr_type2>);
```

- удаление схемы:

```
deleteSchema("schema_name");
```

- добавление записи:

```
addVertex("schema_name", "attr_name", <attr_value>, ...);
```

- получение всех элементов схемы:

```
V("schema_name");
```

- получение элементов схемы, соответствующих набору условий значений атрибутов:

```
V("schema_name").has("attr_name", <select_option>(select_value), ...);
```

- получение записей, ассоциированных с ключевой схемой по ребру:

```
V("schema_name").out("edge_name");
```

- удаление элементов схемы:

```
V("schema_name").delete
```

Поддерживаются следующие условия выборки элементов данных:

- равенство:

```
eq(<attr_value>)
```

- строго больше:

```
gt(<attr_value>)
```

- больше или равно:

```
gte(<attr_value>)
```

- строго меньше:

```
lt(<attr_value>)
```

- меньше или равно:

```
lte(<attr_value>)
```

- неравенство:

```
neq(<attr_value>)
```

- включение подстроки:

```
like(<substr>)
```

Демонстрация работоспособности реализованного модуля.

После составление содержимое структуры печатается на консольный вывод в формате JSON и в файл transfer_data.json.
В рамках реализации дерева для обеспечения произвольного кол-ва условий выбора атрибутов, кол-ва атрибутов при создании схемы, а также произвольного кол-ва операций типа "out" и "has", используется структура vector. Вследствие чего для хранения элементов выделяется массив.

```
addVertex("val", "first", 123, "second", 31.2);
{
  "node": [
    {
      "name": "first",
      "value": 123
    },
    {
      "name": "second",
      "value": 31.200000762939453
    }
  ]
}
```

```
V("test").has("first", eq(123), "second", gte(21.2)).has("third", lte(2)).delete();
{
  "selection": {
    "statements": [
      [
        {
          "attr_name": "first",
          "option": "=",
          "value": 123
        },
        {
          "attr_name": "second",
          "option": ">=",
          "value": 21.200000762939453
        }
      ],
      [
        {
          "attr_name": "third",
          "option": "<=",
          "value": 2
        }
      ],
      {
        "option": "delete"
      }
    ]
  }
}
```

Выводы:

В ходе выполнения данной лабораторной работы были задействованы такие инструменты,
как Flex и Bison. Была рассмотрена грамматика языка запросов Gremlin, на основе которой
была записана спецификация для утилит лексического и синтаксического анализа. Спецификация позволяет составлять запросы на работу с файлом данных (открытие, создание, удаление), работу со схемами (добавление/удаление схем), на добавление элементов данных с указанием значений атрибутов, а также на поиск элементов данных по заданным условиям значения указанных в запросе атрибутов с возможностью вывода информации о соединенных ребрами элементов данных и удаления перечисляемых узлов.

Был реализован модуль, способный производить синтаксический разбор запроса и составлять его на основе абстрактное синтаксическое дерево. Полученное дерево может быть выведено в формате JSON.