Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Факультет программной инженерии и компьютерной техники

Дисциплина: Low-level Programming

Отчёт по лабораторной работе №2

Вариант 9

Выполнил: Нгуен Нгок Дык

Студент группы: P33302

Преподаватель: Кореньков Юрий Дмитриевич

Санкт-Петербург

2023 г

1. Objective:

Implement a module to parse some sufficient subset of the query language. It shall be possible to describe commands for creating, fetching, modifying, and deleting data elements.

Variant 9: Gremblin Query Language - https://docs.janusgraph.org/getting-started/gremlin/

2. Requirement

1. Examine the chosen parser
    a. The tool must support a programming interface compatible with the C language
    b. The facility must be parameterized by a specification that describes the syntactic structure of the parsed language
    c. The tool can function through code generation and / or connection of additional libraries necessary for its operation.
    d. A feature can be implemented from scratch, in which case it must be based on a generic algorithm driven by the specification
2. Learn the syntax of the query language and write the specification for the parser
    a. If you need to add new constructs to the language, add the necessary syntactic constructs to the specification (for example, comparisons in GraphQL)
    b. The query language must support the following features:
        - Conditions
            - Equality and inequality for numbers, strings, and booleans values
            - Strict and non-strict comparisons for numbers
            - Substring existence
        - A logical combination of an arbitrary number of conditions and boolean values
        - Any condition argument can be literal values (constants) or references to values associated with data elements (fields, attributes, properties)
        - Resolving relations between elements of the data model of any conditions on conjugated data elements
        - Support for arithmetic operations and string concatenation is optional
    c. It is allowed to develop your own query language from scratch, in this case it is necessary to show the difference between the main structures and other options (with the exception of typical expressions such as infix comparison operators)
3. Implement a module that uses a parser to parse the query language
    a. The module's API must accept a string with the text of the request and return a structure describing the parse tree of the request or a syntax error message.
    b. The output of the module must contain a hierarchical representation of conditions and other expressions that logically represent hierarchically organized data, even if a linear representation was used at the parser level to parse them.
4. Implement a test program to demonstrate the functionality of the created module, accepting the query text on standard input and outputting the resulting parse tree or an error message to standard output

5. Present the test results in the form of a report, which includes:
    a. In part 3, provide a description of the data structures that represent the result of query parsing
    b. In part 4, describe what additional processing was required on the result of the parse provided by the parser to form the result of the work of the generated module
    c. In part 5, give examples of queries for all the possibilities from clause 2.b and the resulting output of the test program, evaluate the use of the developed RAM module

3. Implementation

Request types:

- **File creation:**
  create(“filename.txt”);
- **Opening an existing file:**
  open(“filename.txt”);
- **Close the file:**
  close();
- **Adding a schema:**
  addSchema(“schema_name”, “attr_name1”, <attr_type1>, “attr_name2”, <attr_type2>);
- **Deleting a schema:**
  deleteSchema(“schema_name”);
- **Adding an entry:**

addVertex(“schema_name”, “attr_name”, <attr_value>, …);

- **Getting all elements of the schema**
  V(“schema_name)
- **Getting schema elements corresponding to a set of attribute value conditions.**
  V(“schema_name”).has(“attr_name”,<select_option> (select_value), …);
- **Getting the records associated with the key scheme by the edge:**
  V(“schema_name”).out(“edge_name”);
- **Deleting circuit elements:**
  V(“schema_name”).delete

The following item selection conditions are supported:

- **Equality:**
  eq(<attr_value>)
- **strictly more:**
  gt(<attr_value>)
- **more or equal:**
  gte(<attr_value>)
- **strictly less**
  lt(<attr_value>)
- **less or equal**
  lte(<attr_value)
- **inequality**
  neq(<attr_value>)
- **inclusion of a substring**
  like(<substr>)

Project structure:

Lexical analysis (lexer.l) happens with the help of flex. Symbols are converted into tokens. Some tokens are converted to enums or base_value_t:

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

The parser (parser.y), written in bison, accepts tokens and builds a parse tree using auxiliary data structures described in vector.h and described types that are specified through 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;
```

At the end of parsing, it all boils down to the **query_tree** structure, and from which a syntax tree is taken, through which, by descending the tree, a JSON format DOM is created according to the given variant.

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

typedef struct {
    char *filename;
} file_work_t;

typedef struct {
    query_type type;
    char* schema_name;
    union {
        file_work_t file_work;
        add_schema_t add_schema;
        delete_schema_t delete_schema;
        add_node_t add_node;
        vector *statements;
    };
} query_tree;
```

```
query_tree get_query_tree();
```

Console printing is used to output the results using the JSON_C_TO_STRING_SPACED , JSON_C_TO_STRING_PRETTY flags, and a transfer_data.json file is also created, which serves as a data storage format between the client and the server.

4. Demonstration of the functionality of the implemented module.

After composition, the contents of the structure are printed to the console output in JSON format and to the data.json file.

As part of the implementation of the tree, to provide an arbitrary number of conditions for selecting attributes, the number of attributes when creating a schema, as well as an arbitrary number of operations like "out" and "has", the vector structure is used. As a result, an array is allocated to store the elements.

```
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"
      }
    ]
  }
}
```

5. Conclusion

Tools such as Flex and Bison were used in this lab. The grammar of the Gremlin query language was considered, on the basis of which the specification for lexical and parsing utilities was written.

The specification allows you to make requests for working with a data file (opening, creating, deleting), working with schemas (adding/deleting schemas), adding data elements with attribute values, as well as searching for data elements by specified conditions for the values of the attributes specified in the request with the possibility of displaying information about the data elements connected by edges and deleting enumerated nodes.

A module was implemented that is capable of parsing a request and compiling it based on an abstract syntax tree. The resulting tree can be output in JSON format.