Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский университет ИТМО»
Факультет программной инженерии и компьютерной техники

Лабораторная работа №2 по дисциплине

«**Низкоуровневое программирование**»

Выполнил:
Миху Вадим Дмитриевич
Факультет “ПИиКТ”
Группа: P33301

Преподаватель:
Кореньков Юрий Дмитриевич
Вариант 5

Санкт-Петербург, 2023 г.

Оглавление

Задание

Вариант №5

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
   e. При необходимости добавления новых конструкций в язык, добавить нужные синтаксические конструкции в спецификацию (например, сравнения в GraphQL)
   f. Язык запросов должен поддерживать возможность описания следующих конструкций: порождение нового элемента данных, выборка, обновление и удаление существующих элементов данных по условию
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.

Выполнение

В ходе работы была реализована программа, исходный код которой был опубликован на GitHub

https://github.com/filberol/lab2llp

Был написан Cmake файл, позволяющий запускать программу на платформах Windows и *nix.

Согласно варианту, программа строит Ast дерево для языка запросов AQL. Файл сборки содержит только одну цель сборки, которая читает из файла тестовый запрос и формирует его представление.

1. Для работы анализатора использованы утилиты Flex и Bison, которые позволяют установить синтаксис языка. Flex используется для создания выражений и токенов, которые используются для парсинга и построения дерева.
2. Эти средства поддерживают совместимость с языком C, генерируя синтаксические анализаторы исходя из файла синтаксиса соответствующего формата.

Спецификация языка

Операции

```
"FOR" { return T_FOR; }
"FILTER" { return T_FILTER; }
"RETURN" { return T_RETURN; }

"{" { return T_OBJECT_OPEN; }
"}" { return T_OBJECT_CLOSE; }

"IN" { return T_IN; }
"REMOVE" { return T_REMOVE; }
"INSERT" { return T_INSERT; }
"UPDATE" { return T_UPDATE; }

"TRUE" { return T_TRUE; }
"FALSE" { return T_FALSE; }
```

Комбинации

```
"==" { return T_EQ; }
"!=" { return T_NE; }
">=" { return T_GE; }
">" { return T_GT; }
"<=" { return T_LE; }
"<" { return T_LT; }
"," { return T_COMMA; }
"(" { return T_OPEN; }
")" { return T_CLOSE; }
":" { return T_COLON; }
"!" { return T_NOT; }
"&&" { return T_AND; }
"||" { return T_OR; }
```

Строки

```
(\"[^\"]*\") {
  /* quoted string */
  yylval.s_val = yytext;
  printf("%s", yytext);
  return T_QUOTED_STRING;
}

(($?|_+)[a-zA-Z]+[_a-zA-Z0-9]*) {
  /* unquoted string */
  yylval.s_val = yytext;
  return T_STRING;
}
```

Числа

```
(0|[1-9][0-9]*) {
  /* a numeric integer value, base 10 (decimal) */
  yylval.node = create_int_node(atoi(yytext));
  return T_INTEGER;
}

((0|[1-9][0-9]*)(\.[0-9]+)?|\.[0-9]+)([eE][\-\+]?[0-9]+)? {
  /* a numeric double value, base 10 (decimal) */
  double value = atof(yytext);
  yylval.node = create_double_node(value);
  return T_DOUBLE;
}
```

Этот набор используется Flex для разбиения на токены и создания примитивных нод из значений

Далее приведены выражения языка

```
queryStart: query {};

query: optional_statement_block_statements final_statement {};

final_statement:
    return_statement {}
  | remove_statement {}
  | insert_statement {}
  | update_statement {}
  ;

optional_statement_block_statements:
    /* empty */ {
    }
  | optional_statement_block_statements statement_block_statement {
    }
  ;

statement_block_statement:
    for_statement {}
  | filter_statement {}
  | remove_statement {}
  | insert_statement {}
  | update_statement {}
  ;

for_output_variable:
    variable_name {
          struct AstNode* node = create_string_node($1);
          $$ = node;
        }
    ;

for_statement:
    T_FOR for_output_variable T_IN expression {
        struct AstNode* variableNameNode = (struct AstNode*)($2);
        struct AstNode* variableNode = create_variable_node(variableNameNode->value._string);
        add_variable(variableNameNode->value._string, variableNode);
        add_operation(get_current_scope(), create_for_node(variableNode, $4));
    }
  ;

filter_statement:
    T_FILTER expression {
      struct AstNode* filterNode = create_filter_node($2);
      add_operation(get_current_scope(), filterNode);
    }
  ;

return_statement:
    T_RETURN expression {
      struct AstNode* node = create_return_node($2);
      add_operation(get_current_scope(), node);
    }
  ;

in_collection:
    T_IN in_collection_name {
      $$ = $2;
    }
   ;

remove_statement:
    T_REMOVE expression in_collection  {
      struct AstNode* node = create_remove_node($2, $3);
      add_operation(get_current_scope(), node);
    }
  ;
```

```
insert_statement:
    T_INSERT expression in_collection {
      struct AstNode* node = create_insert_node($2, $3);
      add_operation(get_current_scope(), node);
    }
  ;

update_parameters:
    expression in_collection {
      struct AstNode* node = create_update_node($1, $2);
      add_operation(get_current_scope(), node);
    }
  ;

update_statement:
    T_UPDATE update_parameters {
    }
  ;


object:
    T_OBJECT_OPEN {
      struct AstNode* node = create_node_object();
      push_common(node);
    } optional_object_elements T_OBJECT_CLOSE {
      $$ = (struct AstNode*)(pop_common());
    }
  ;

optional_object_elements:
    /* empty */ {
    }
  | object_elements_list {
    }
  | object_elements_list T_COMMA {
    }
  ;

object_elements_list:
    object_element {
    }
  | object_elements_list T_COMMA object_element {
    }
  ;

object_element:
    T_STRING {
      struct AstNode* variable = get_variable($1);
      struct AstNode* node = create_reference_node(variable);
      push_object_element($1, node);
      $$ = node;
    }
  | object_element_name T_COLON expression {
      push_object_element($1, $3);
    }
  ;


object_element_name:
    T_STRING {
      $$ = copy_string($1);
    }
  | T_QUOTED_STRING {
      $$ = copy_quoted_string($1);
    }

operator_binary:
    expression T_OR expression {
      $$ = create_binary_op_node(NODE_TYPE_OPERATOR_BINARY_OR, $1, $3);
```

```
    }
  | expression T_AND expression {
      $$ = create_binary_op_node(NODE_TYPE_OPERATOR_BINARY_AND, $1, $3);
    }
  | expression T_EQ expression {
      $$ = create_binary_op_node(NODE_TYPE_OPERATOR_BINARY_EQ, $1, $3);
    }
  | expression T_NE expression {
      $$ = create_binary_op_node(NODE_TYPE_OPERATOR_BINARY_NE, $1, $3);
    }
  | expression T_LT expression {
      $$ = create_binary_op_node(NODE_TYPE_OPERATOR_BINARY_LT, $1, $3);
    }
  | expression T_GT expression {
      $$ = create_binary_op_node(NODE_TYPE_OPERATOR_BINARY_GT, $1, $3);
    }
  | expression T_LE expression {
      $$ = create_binary_op_node(NODE_TYPE_OPERATOR_BINARY_LE, $1, $3);
    }
  | expression T_GE expression {
      $$ = create_binary_op_node(NODE_TYPE_OPERATOR_BINARY_GE, $1, $3);
    }
  | expression T_IN expression {
      $$ = create_binary_op_node(NODE_TYPE_OPERATOR_BINARY_IN, $1, $3);
    }
  ;

expression:
 operator_binary {
      $$ = $1;
    }
 | value_literal {
      $$ = $1;
    }
  | reference {
      $$ = $1;
    }
  ;

reference:
    T_STRING {
      struct AstNode* node = NULL;
      void* variable = NULL;

      variable = get_variable($1);
      if (variable == NULL) {
          node = create_data_source_node($1);
      } else {
        node = variable;
      }
      $$ = node;
    }
    | reference '.' T_STRING {
        struct AstNode* ref = (struct AstNode*)($1);
        struct AstNode* access = create_attribute_access_node($3);
        add_operation(ref, access);
        $$ = ref;
    }
    | T_OPEN expression T_CLOSE {
        $$ = $2;
    }
    | object {
      $$ = $1;
    }
  ;

numeric_value:
    T_INTEGER {
      $$ = $1;
    }
```

```
  | T_DOUBLE {
      $$ = $1;
    }
  ;

value_literal:
    T_QUOTED_STRING {
      $$ = create_string_node($1);
    }
  | numeric_value {
      $$ = $1;
    }
  | T_NULL {
      $$ = create_null_value_node();
    }
  | T_TRUE {
      $$ = create_bool_node(true);
    }
  | T_FALSE {
      $$ = create_bool_node(false);
    }
  ;

in_collection_name:
    T_STRING {
      $$ = create_data_source_node($1);
    }
  ;

variable_name:
    T_STRING {
      $$ = copy_string($1);
    }
  ;
```

После разбиения текста на токены и создания нод начинается парсинг синтаксического дерева. Для этого вызвается yyparse, который по сгенерированному синтаксическому парсеру стоит дерево. После, дерево рекурсивно выводится.

```
static void generateDotCode(struct AstNode *node, FILE *file, int id) {
    if (node == NULL) {
        return;
    }
    fprintf(file, "\t%d [ label=\"Type: %s Value: %s\" ];\n", id, typeString[node->type],
getValue(node));
    for (size_t i = 0; i < node->children.children_count; i++) {
        struct AstNode *childNode = node->children.data[i];
        max_id++;
        fprintf(file, "\t\t%d -> %d;\n", id, max_id);
        generateDotCode(childNode, file, max_id);
    }
}
```

Пример работы

```
FOR u IN users
  FOR p IN products
    INSERT { _from: u._id, _so: p._val } IN recommendations
```

```
digraph AstTree {
    0 [ label="Type: NODE_TYPE_ROOT Value: NULL" ];
       0 -> 1;
    1 [ label="Type: NODE_TYPE_FOR Value: NULL" ];
       1 -> 2;
    2 [ label="Type: NODE_TYPE_VARIABLE Value: u" ];
       1 -> 3;
    3 [ label="Type: NODE_TYPE_DATA_SOURCE Value: users" ];
       1 -> 4;
    4 [ label="Type: NODE_TYPE_FOR Value: NULL" ];
       4 -> 5;
    5 [ label="Type: NODE_TYPE_VARIABLE Value: p" ];
       4 -> 6;
    6 [ label="Type: NODE_TYPE_DATA_SOURCE Value: products" ];
       4 -> 7;
    7 [ label="Type: NODE_TYPE_INSERT Value: NULL" ];
       7 -> 8;
    8 [ label="Type: NODE_TYPE_OBJECT Value: NULL" ];
       8 -> 9;
    9 [ label="Type: NODE_TYPE_OBJECT_ELEMENT Value: _from:" ];
       9 -> 10;
    10 [ label="Type: NODE_TYPE_VARIABLE Value: u" ];
       10 -> 11;
    11 [ label="Type: NODE_TYPE_ATTRIBUTE_ACCESS Value: _id" ];
       8 -> 12;
    12 [ label="Type: NODE_TYPE_OBJECT_ELEMENT Value: _so:" ];
       12 -> 13;
    13 [ label="Type: NODE_TYPE_VARIABLE Value: p" ];
       13 -> 14;
    14 [ label="Type: NODE_TYPE_ATTRIBUTE_ACCESS Value: _val" ];
}
```

Выводы по работе

В ходе работы была реализована программа, позволяющая по определенным правилам создавать синтаксический анализатор и с помощью парсера строить синтаксическое дерево. Программа позволяет определять валидность входного текста.