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 2](#_Toc145016776)

[Выполнение 3](#_Toc145016777)

[Выводы по работе 5](#_Toc145016778)

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

1. Средство должно поддерживать программный интерфейс совместимый с языком С
2. Средство должно параметризоваться спецификацией, описывающий синтаксическую структуру разбираемого языка
3. Средство может функционировать посредством кодогенерации и/или подключения необходимых для его работы дополнительных библиотек
4. 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

1. При необходимости добавления новых конструкций в язык, добавить нужные синтаксические конструкции в спецификацию (например, сравнения в GraphQL)
2. 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

* 1. Программный интерфейс модуля должен принимать строку с текстом запроса и возвращать структуру, описывающую дерево разбора запроса или сообщение о синтаксической ошибке
  2. 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** | **Комбинации** |
|  |  |
| **Строки** | **Числа** |
|  |  |

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.