МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Алгоритмы и структуры данных»

Тема: Рекурсивная обработка иерархических список

Студент гр. 9303 _____ Муратов Р.А.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург

2020

Цель работы.

Познакомиться с одной из часто используемых на практике нелинейных конструкций, иерархическим списком, способами её организации и рекурсивной обработки. Получить навыки решения задач обработки иерархических списков, как с использованием базовых функций их рекурсивной обработки, так и без использования рекурсии.

Задание.

Вариант 16

Пусть выражение (логическое, арифметическое, алгебраическое) представлено иерархическим списком. В выражение входят константы и переменные, которые являются атомами списка. Операции представляются в префиксной форме ((<операция> <аргументы>)), либо в постфиксной форме ((<аргументы> <операция>)). Аргументов может быть 1, 2 и более.

В индивидуальном задании указывается: тип выражения (возможно дополнительно - состав операций), вариант действия и форма записи.

Задание 16: логическое, проверка синтаксической корректности, добавить 4-ую операцию (которая может принимать 2 аргумента), префиксная форма

Основные теоретические положения.

В практических приложениях возникает необходимость работы с более сложными, чем линейные списки, нелинейными конструкциями. Рассмотрим одну из них, называемую иерархическим списком элементов базового типа El или S-выражением.

Определим соответствующий тип данных S_expr (El) рекурсивно, используя определение линейного списка (типа L_list):

< S_expr (El) > ::= < Atomic (El) > | < L_list (S_expr (El)) >,

< Atomic (E) > ::= < El >.

< L_list(El) > ::= < Null_list > | < Non_null_list(El) >

< Null_lis t> ::= Nil

< Non_null_list(El) > ::= < Pair(El) >

< Pair(El) > ::= (< Head_l(El) > . < Tail_l(El) >)

< Head_l(El) > ::= < El >

< Tail_l(El) > ::= < L_list(El) >

Традиционно иерархические списки представляют или графически или в виде скобочной записи. На рисунке 1 приведен пример графического изображения иерархического списка. Соответствующая этому изображению сокращенная скобочная запись - это (a (b c) d e).

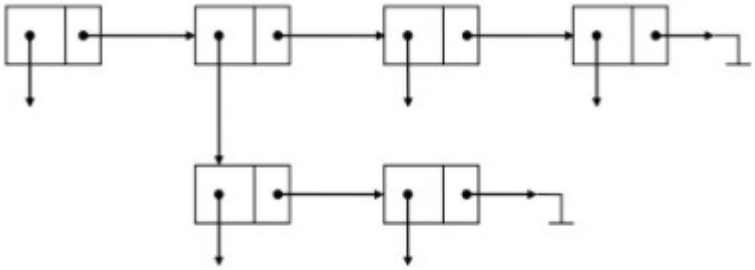

Рисунок 1 – Иерархический список

Выполнение работы.

`Type` – перечисление, которое задает три константы: `ARG`, `PAIR`, `OPERATION`, используемые в узлах иерархического списка для определения того, что находится в данном узле.

`SExpr` – структура, представляющая собой узел списка: поле `type` необходимо, чтобы определить тип узла, и два остальных поля необходимы для универсальности данной структуры. В зависимости от типа узла одной из полей является валидным, другое – нет.

Для данной структуры данных реализован определенный набор функции:

`LogicalExpr head(const LogicalExpr logExpr)` – данная функция принимает структуру данных (фактически указатель на узел) и возвращает её «голову», если `logExpr` указывает на `PAIR`, иначе программа завершает свою работу.

`LogicalExpr tail(const LogicalExpr logExpr)` – данная функция принимает структуру данных (фактически указатель на узел) и

возвращает её «хвост», если `logExpr` указывает на `PAIR`, иначе программа завершает свою работу.

`LogicalExpr construct(const LogicalExpr h, const LogicalExpr t)` – данная функция конструирует новый узел. Если `t` – атом, то программа завершает свою работу, иначе создается новый узел `PAIR`, в «голову» которого записывается `h`, а в «хвост» `t`.

`LogicalExpr makeArg(Base a)` и `LogicalExpr makeOper(Base op)` – эти две функции выделяют необходимую память и создают соответственно аргументы и операции.

`bool isArg(const LogicalExpr le); bool isOper(const LogicalExpr le); bool isPair(const LogicalExpr le); bool isNull(const LogicalExpr le)` – эти четыре функций проверяют, на что указывает `le`: на аргумент, на операцию, на пару или `le` – нулевой указатель.

`void destroy(LogicalExpr le)` – данная рекурсивная функция удаляет весь иерархический список.

`void print(const LogicalExpr logExpr, std::ostream& os); void printSeq(const LogicalExpr logExpr, std::ostream& os)` – эти две рекурсивные функции выводят иерархический список в скобочной записи. Первая – основная, вторая – вспомогательная.

`void read(LogicalExpr& logExpr, std::istream& is)`

`void readSExpr(Base prev, LogicalExpr& logExpr, std::istream& is); void readSeq(LogicalExpr& logExpr, std::istream& is)`

– эти три функции считывают иерархический список из вводного потока `is`. Первая – основная, вторая и третья – вспомогательные (они являются рекурсивными). Если вводная строка имеет неверный формат, например, количество открывающих скобок неравно количеству закрывающих скобок, тогда программа завершает свою работу (даже если, например, во входном файле остались несчитанные строки).

`bool check(LogicalExpr logExpr)` и `bool checkArgs(LogicalExpr logExpr, int8_t& counter, std::string opType)` – две рекурсивные функции для проверки корректности логического выражения. Первая – основная, вторая – вспомогательная. `counter` – счетчик аргументов для операции, с который на данный момент работает функция, `opType` – тип данной операции.

В функции main из входного файла, название которого ввел пользователь, считывается построчно каждое выражение и проверяется его корректность. Программа может прекратить свою работу и не дочитать все выражения из файла (см. функции `read, readSExpr, readSeq`).

Разработанный программный код см. в приложении А.

Результаты тестирования программы приведены в приложении Б.

Выводы.

Познакомился с одной из часто используемых на практике нелинейных конструкций, иерархическим списком, способами её организации и рекурсивной обработки. Получил навыки решения задач обработки иерархических списков, как с использованием базовых функций их рекурсивной обработки, так и без использования рекурсии.

Была реализована программа, считывающая из файла, название которого вводит пользователь, построчно каждое логическое выражение и с помощью функции `check` проверяющая ее корректность.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include <string>
#include <list>
#include <fstream>
#include <sstream>

namespace HList {
enum Type : uint8_t {
  ARG, PAIR, OPERATION
};

using Base = std::string;

struct SExpr;
struct Pair {
  SExpr* head;
  SExpr* tail;
};

struct SExpr {
  Type type;
  Pair pair;
  Base opOrArg;
};

using LogicalExpr = SExpr*;

LogicalExpr head(const LogicalExpr logExpr) {
  if (logExpr != nullptr) {
    if (!isArg(logExpr)) {
      return logExpr->pair.head;
    } else {
      std::cerr << "Error: head(Atom)\n";
      exit(1);
    }
  } else {
    std::cerr << "Error: head(nullptr)\n";
    exit(1);
  }
}

LogicalExpr tail(const LogicalExpr logExpr) {
  if (logExpr != nullptr) {
    if (!isArg(logExpr)) {
      return logExpr->pair.tail;
    } else {
      std::cerr << "Error: head(Atom)\n";
      exit(1);
    }
  } else {
    std::cerr << "Error: tail(nullptr)\n";
    exit(1);
  }
}
```

```
bool isArg(const LogicalExpr le) {
  return (!isNull(le)) && (le->type == ARG);
}

bool isNull(const LogicalExpr le) {
  return le == nullptr;
}

LogicalExpr construct(const LogicalExpr h, const LogicalExpr t) {
  if (isArg(t)) {
    std::cerr << "Error: construct(*, Atom)" << std::endl;
    exit(1);
  } else {
    LogicalExpr newHL = new SExpr;
    newHL->type = PAIR;
    newHL->pair.head = h;
    newHL->pair.tail = t;
    return newHL;
  }
}

LogicalExpr makeArg(Base a) {
  LogicalExpr newHL = new SExpr;
  newHL->type = ARG;
  newHL->opOrArg = a;
  return newHL;
}

void destroy(LogicalExpr le) {
  if (!isNull(le)) {
    if (le->type == PAIR) {
      destroy(head(le));
      destroy(tail(le));
    }
    delete le;
  }
}

void print(const LogicalExpr logExpr, std::ostream& os) {
  if (isNull(logExpr)) {
    os << "()";
  } else if (isArg(logExpr)) {
    os << logExpr->opOrArg;
  } else if (isOper(logExpr)) {
    os << logExpr->opOrArg;
//    os << "[" << hl->opOrArg << "]";
  } else {
    os << "(";
    printSeq(logExpr, os);
    os << ")";
  }
}

void printSeq(const LogicalExpr logExpr, std::ostream& os) {
  if (!isNull(logExpr)) {
    print(head(logExpr), os);
    printSeq(tail(logExpr), os);
  }
}
```

```
      void read(LogicalExpr& logExpr, std::istream& is) {
        Base x;
        is >> x;
        readSExpr(x, logExpr, is);
      }

      void readSExpr(Base prev, LogicalExpr& logExpr, std::istream& is) {
        if (prev == ")") {
          std::cerr << "Error: prev == \')\'" << std::endl;
          exit(1);
        } else if (prev != "(") {
           if (prev == "AND" || prev == "OR" || prev == "NOT" || prev ==
"XOR")
            logExpr = makeOper(prev);
          else
            logExpr = makeArg(prev);
        } else {
          readSeq(logExpr, is);
        }
      }

      void readSeq(LogicalExpr& logExpr, std::istream& is) {
        Base x;
        is >> x;
      //  std::string op;
        LogicalExpr le1, le2;
        if (is.eof()) {
          std::cerr << "Error: incorrect expression" << std::endl;
          exit(1);
        } else {
          if (x == ")") {
            logExpr = nullptr;
          }
          else {
            readSExpr(x, le1, is);
            readSeq(le2, is);
            logExpr = construct(le1, le2);
          }
        }
      }

      bool isCorrect(LogicalExpr logExpr) {
        if (isNull(logExpr)) {
          return false;
        } else if (isArg(logExpr)) {
          return true;
        } else
          if (head(logExpr)->type == ARG) {
            return true;
          } else {
            return false;
          }
      }

      bool check(LogicalExpr logExpr, std::ostream& os) {
        int8_t counter = 0;
        std::string opType;
        if (isPair(logExpr)) {
          LogicalExpr h = head(logExpr);
```

```
          if (isOper(h) && (h->opOrArg == "AND" || h->opOrArg == "OR" ||
h->opOrArg == "XOR" || h->opOrArg == "NOT")) {
            counter = 0;
            opType = h->opOrArg;
            return checkArgs(tail(logExpr), counter, opType, os);
          } else {
              os << "Invalid expression: There is missed one or more
operations in the expression" << std::endl;;
            return false;
          }
        } else {
          os << "Invalid expression: There is no arguments or operations
in the expression" << std::endl;
          return false;
        }
      }

      bool checkArgs(LogicalExpr logExpr, int8_t& counter, std::string
opType, std::ostream& os) {
        int8_t oldVal = 0;
        if (isPair(logExpr)) {
          LogicalExpr h = head(logExpr);
          if (isPair(h)) { // Операция
            oldVal = counter + 1; // Учитываем аргумент, являющийся рез-
том операции
            bool correctnessNextOp = check(h, os);
            counter = oldVal;
            return correctnessNextOp && checkArgs(tail(logExpr), counter,
opType, os);
          } else if (h == nullptr) { // Пустые скобки
              os << "Invalid expression: There is an empty brackets" <<
std::endl;
            return false;
          } else { // Аргументы
            return checkArgs(tail(logExpr), ++counter, opType, os);
          }
        } else { // Достигнут конец списка аргументов
          if (opType == "AND" || opType == "OR" || opType == "XOR")
            if (counter > 1)
              return true;
            else {
              os << "Invalid expression: There is too few arguments for "
<< opType << std::endl;
              return false;
            }
          else {
            if (counter > 0)
              return true;
            else {
               os << "Invalid expression: There is too few arguments for
NOT" << std::endl;
              return false;
          }
          }
        }
      }

      LogicalExpr makeOper(Base op) {
        LogicalExpr le = new SExpr;
        le->type = OPERATION;
```

```
  le->opOrArg = op;
  return le;
}


bool isOper(const LogicalExpr le) {
  return (!isNull(le)) && (le->type == OPERATION);
}

bool isPair(const LogicalExpr le) {
  return (!isNull(le)) && (le->type == PAIR);
}
}

int main() {
  HList::LogicalExpr logExpr = nullptr;

  std::string inFileName;
  std::cout << "Enter input file name: ";
  std::cin >> inFileName;

  std::fstream f(inFileName);
  if (!f.is_open()) {
    std::cerr << "Error: cannot open a file" << std::endl;
    exit(1);
  }

  std::string line;

  while (1) {
    std::getline(f, line);
    line.push_back('\n');
    if (f.eof()) {
      break;
    }
    std::stringstream lineStream(line);
    HList::read(logExpr, lineStream);
    HList::print(logExpr, std::cout);
    std::cout << '\n';
    if (check(logExpr)) {
      std::cout << "CORRECT\n";
    } else {
      std::cout << "INCORRECT\n";
    }
    destroy(logExpr);
    logExpr = nullptr;
  }

  f.close();

  return 0;
}
```

ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ

Таблица Б.1 - Примеры тестовых случаев

№ п/п	Входные данные	Выходные данные	Комментарии
1.	(AND a (OR -1 c) 0 e (NOT q (OR 0 0) w) (XOR t y u))	CORRECT	Корректное логическое выражение
2.	(AND a ( ) b)	Invalid expression: There is an empty brackets INCORRECT	Пустая пара скобок в выражении - ошибка
3.	(AND 1 2 (3 4))	Invalid expression: There is missed one or more operations in the expression INCORRECT	После открывающей скобки нет операции - ошибка
4.	(AND (NOT) b)	Invalid expression: There is too few arguments for NOT	Мало аргументов (меньше 1) для операции NOT - ошибка
5.	(AND a)	Invalid expression: There is too few arguments for AND INCORRECT	Мало аргументов для оператора AND (меньше 1) - ошибка
6.	( )	Invalid expression: There is no arguments or operations in the expression INCORRECT	Нет ни аргументов, ни операции - ошибка
7.	((AND a b))	Invalid expression: There is missed one or more operations in the expression	После открывающей скобки нет операции - ошибка

		INCORRECT	
8.	(AND (NOT) b)	Invalid expression: There is too few arguments for NOT	Мало аргументов (меньше 1) для операции NOT
9.	(AND	Error: incorrect expression	Некорректное выражение