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: «Иерархические списки»

Студент гр. 7381 Лукашев Р.С.
Преподаватель Фирсов М.А.

Санкт-Петербург
2018

Задание.

26. Символьное дифференцирование алгебраического выражения, рассматриваемого как функция от одной из переменных. На входе выражение в виде иерархического списка и переменная, по которой следует дифференцировать. На выходе – производная исходного выражения. Набор операций (функций), которые могут входить в выражение: +, -, *, ^, exp(). Форма записи – префиксная.

Дополнительное требование к 1-ой работе: необходимо описать структуру данных, используемую для представления иерархического списка в данной работе, и привести графическую схему примера иерархического списка (см. приложение Е).

Пояснение задания.

На вход программе подаётся символьное значение переменной, по которой необходимо дифференцировать, а затем последовательность символов, являющихся скобочной записью иерархического списка, которую необходимо проверить на корректность относительно понятия “алгебраическое выражение”, а затем взять от этого выражения производную используя рекурсивный алгоритм. В выражение входят константы и переменные, которые являются атомами списка. Операции представляются в префиксной форме (`(<операция> <аргументы>)`). Аргументов может быть 1, 2 и более.

Описание алгоритма.

Пусть

```
<выражения>::= <выражение> | <выражение> <выражения>
<выражение>::= (<операция> <аргументы>) | <функция>(<аргумент>)
```

```
<аргумент>::=<выражение> | <элемент>
<аргументы>::=<аргумент> | <аргумент> <аргументы>
<элемент>::= <переменная> | <константа>.
<операция>::= + | - | * | ^
<функция>::= exp
<переменная>::= последовательность букв
<константа>::= положительные, отрицательные числа и ноль
```

Синтаксис операций и функций:

```
+: (+ <аргумент> <аргументы>)
-: (- <аргументы>)
*: (* <аргумент> <аргументы>)
^: (^ <аргумент> <аргумент>)
exp: exp (<аргумент>)
```

Сначала считывается переменная, по которой необходимо дифференцировать, затем поэлементно читается из входного потока выражение и записывается в виде иерархического списка. В случае, если в входном потоке нарушается синтаксис операций, функций или выражения, выводится сообщение о характере ошибки, и алгоритм завершает работу. После этого, в случае валидного ввода, берется производная от данного выражения и записывается в новый список по следующим правилам:

1) Встречена переменная: если она совпадает с переменной, по которой необходимо дифференцировать, то возвращается атомарный элемент списка со значением 1, иначе 0.
2) Встречена константа: возвращается элемент со значением 0.
3) Встречено + или -: возвращается новый список, в который записывается встреченная операция и последовательность производных её аргументов.

4) Встречено *: возвращается новый список, в который записывается сумма произведений аргументов операции, в каждом из произведений последовательно берется производная от следующего аргумента.
5) Встречена ^: возвращается список производной степенно-показательной функции: $(u^v)' = vu^{v-1}u' + u^v v' \ln(u)$.
6) Встречена exp: возвращается список равный производной аргумента функции на функцию от аргумента.

   После чего происходит вывод списка на экран и алгоритм завершает работу.

Описание функций и структур данных.

1. `struct hlist` – иерархический список, реализованный по следующему правилу:

   ```
   <hlist(el)>::=<atomic(el)> | <two_ptr(hlist(el))>
   <atomic(el)>::=el
   <two_ptr(hlist(el))>::=<null_list> | <non_null_list(el)>
   <null_list>::=nil
   <non_null_list(el)>::=<pair(el)>
   <pair(el)>::=(<head(el)> <tail(el)>)
   <head(el)>::= el
   <tail(el)>::= <two_ptr(el)>
   ```
2. `template <class base> bool isatomic(const hlist<base>* ptr)` – функция, возвращающая атомарность элемента списка ptr. Далее шаблон будет упускаться.
3. `bool empty(const hlist<base>* ptr)` – функция, возвращающая 1, если список ptr пуст (NULL pointer).

4. `hlist<base>* head(const hlist<base>* ptr)` – функция-селектор, возвращающая голову списка ptr.
5. `hlist<base>* tail(const hlist<base>* ptr)` – функция-селектор, возвращающая хвост списка ptr.
6. `hlist<base>* cons(const hlist<base>* head, const hlist<base>* tail)` – функция-конструктор. Создает элемент списка с головой head и хвостом tail.
7. `hlist<base>* make_atom(const base el)` – функция-конструктор. Создает атомарный элемент со значением el.
8. `base get_atom(const hlist<base>* ptr)` – возвращает атом по адресу элемента ptr.
9. `void destroy(hlist<base>* ptr)` – функция-деструктор списка.
10. `hlist<base>* copy(const hlist<base>* ptr)` – создает и возвращает копию списка, находящегося по адресу ptr.
11. `enum type { CONST, NUMBER, OPERATION, FUNCTION }` – перечисление типов элемента алгебраического выражения.
12. `struct alg_el` – элемент алгебраического выражения (содержит тип элемента и его символьное/целочисленное обозначение).
13. `alg_el* read_el(std::istream &input)` – возвращает элемент алгебраического выражения, считанный из потока input.
14. `void print_el(std::ostream &output, alg_el* el)` – печатает элемент el в поток output.
15. `alg_el* make_el(const char* str, type t)` – создает и возвращает элемент выражения с символьным обозначением str типа t.

16. `alg_el* make_el(int num, type t)` – создает и возвращает элемент выражения с целочисленным значением num типа t.
17. `void read_expr(lisp& d, int ldepth, alg_el* op, std::istream& in, std::ostream out)` – основная функция считывания списка d из входного потока in. ldepth – текущая глубина считываемого линейного списка аргументов, op – операция (функция), с которой ассоциируется текущий список, out – поток вывода промежуточной информации и ошибок. Функция при считывании проводит синтаксический анализ входных данных.
18. `void read_seq(alg_el* prev, lisp& d, int ldepth, alg_el* op, std::istream& in, std::ostream out)` – вспомогательная функция считывания. Записывает в d элемент prev, если он существует, иначе вызывает read_expr.
19. `void print_expr(const lisp& d, std::ostream& out)` – основная функция вывода содержимого списка d в выходной поток out.
20. `void write_seq(const lisp& x, std::ostream out)` – вспомогательная функция вывода содержимого списка.
21. `enum err_code{NO_CLOSING, NO_OPENING, NO_VARIABLE, NOT_ENOUGH_ARGS, UNEXPECTED_SYM, EXPECTED_OP, TOO_MANY_ARGS, EXCESS_C}` – перечисление обрабатываемых ошибок ввода.
22. `void error_handler(err_code code, std::ostream& out)` – функция-обработчик ошибок.
23. `alg_el* diff` – элемент (переменная), по которому необходимо дифференцировать выражение.
24. `lisp derivative(const lisp& d)` – основная функция взятия производной. Берет производную функции, записанной в списке d,

и возвращает новый список – производную. Поэлементно анализирует список d, и записывает в новый список элементы, равные производной каждого отдельного выражения или аргумента. Т.е. если d является переменной, по которой необходимо дифференцировать, то в новый список записывается производная переменной, т.е. 1. Если элемент – любая другая переменная или константа, то записывается 0. Если элемент является узлом, то записывается производная узла в зависимости от его головной операции (функции) (либо просто производная головы, если она не атомарна).

25. `lisp derivPM(const lisp& c)` – функция, возвращающая производную функции в списке c для операций плюс и минус.
26. `lisp derivMul(const lisp& c, int depth)` – функция, возвращающая производную функции в списке c для операции умножения. depth – вспомогательная переменная, обозначающая то, какой по счету множитель списка c необходимо дифференцировать.
27. `lisp derivPow(const lisp& c, int step)` – функция, возвращающая производную функции в списке c для операции возведения в степень. step – вспомогательная переменная, характеризующая шаг алгоритма построения списка – производной степени.
28. `lisp derivExp(const lisp& c)` – функция, возвращающая производную функции в списке c для функции экспоненты.
29. `void displayFileContents(std::ifstream* in)` – функция, выводящая на консоль содержимое файла in.

Тестирование.

В первую очередь считывается переменная, по которой необходимо дифференцировать. В случае, если считан элемент другого вида, выводится сообщение о ошибке (тест 1). Далее считывается исходное алгебраическое выражение. Если в какой-то момент нарушается синтаксис, выводится сообщение о ошибке (тесты 2, 3, 4, 5). Иначе берется производная выражения и выводится на экран (тесты 6, 7).

Таблица 1 – Тестирование программы.

№ теста	**Исходные данные:**	**Результат:**
1	+ (* x y)	! - Expected constant as an argument
2	x (* x ^)	0.([ ] 0.* [ ] 1.x . [*] 2.^ . . [*] ! - Unexpected symbol
3	x (* x y	0.([ ] 0.* [ ] 1.x . [*] 2.y . . [*] ! - Expected closing bracket
4	x (^ x)	0.([ ] 0.^ [ ] 1.x . [^] 2.) . . [^] ! - Not enough arguments
5	x (* x (y) z)	0.([ ] 0.* [ ] 1.x . [*] 2.(. . [*] 0.y [ ] ! - Expected operation
6	x (* x y)	0.([ ] 0.* [ ] 1.x . [*] 2.y . . [*] 3.) . . . [*]

		derivative: (+ (* 1 y) (* x 0))
7	x exp(x)	0.e [ ] 0.([ ] 0.x [exp] 1.) . [exp] derivative: (* 1 exp(x))

Тест 7. Считывается x – переменная, по которой необходимо дифференцировать. Затем вызывается read_expr, считывается exp, алгоритм начинает считывать выражение для функции. Считывается ‘(‘, после чего вызывается read_seq, который в свою очередь вызывает read_expr, который считывает x, и вызывает read_seq, который возвращает атомарный элемент x. После чего вызывается read_expr, в котором считывается ‘)’, происходит проверка на корректность введенных в скобочках данных, ‘)’ возвращается в поток для дальнейшей проверки корректности exp( ), и возвращается NULL (пустой элемент списка). Затем происходит выход из вложенных функций, поочередно вызывается cons, в результате получается список (exp (x)), считывается ‘)’, который мы вернули в поток, и заключается корректность введенных данных. Затем вызывается функция derivative. Поскольку в голове списка находится exp, возвращается результат функции derivExp(d), в которой создается список (* derivative(x) copy(d)). derivative(x) = 1 (переменная, по которой берется производная). В итоге возвращается список (* 1 exp(x)). Далее, происходит вывод этого списка и завершение работы программы.

Вывод.

В процессе выполнения лабораторной работы были получены знания и навыки по ООП, деревьям, иерархическим спискам, рекурсивным функциям и bash-скриптам. Работа написана на языке C++.

Приложение А. Содержимое файла alg_expr.h

```
#pragma once
#include <fstream>
#include <iostream>
#include "hlist.h"

	enum type { VARIABLE, NUMBER, OPERATION, FUNCTION };

	struct alg_el {

		type el_type;

		union {
			long long num;
			char* str;
		} el;
	};

	alg_el* read_el(std::istream &input);

	void print_el(std::ostream &output, alg_el* el);

	alg_el* make_el(const char* str, type t);
alg_el* make_el(int num, type t);
```

Приложение Б. Содержимое файла alg_expr.cpp

```
#include "stdafx.h"
#include <fstream>
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include "alg_expr.h"

	alg_el* read_el(std::istream &input) {
		if (input.eof()) return NULL;
		char* n = new char;
		int i = 0;
		alg_el* el = new alg_el;
		do {
			*(n + i) = input.get();
		} while ((*(n + i) == ' ') && input.eof());

		input.unget();
		while (input.good()) {
			*(n + i) = input.get();
			if (input.eof()) break;
			if (*(n + i) == ' ') break;
			if (*(n + i) == '\n') break;
```

```
                        if (*(n + i) == '(' || *(n + i) == ')') {
                              input.unget();
                              break;
                        }
                        i++;
                        n = (char*)realloc(n, (i + 1) * sizeof(char));
                  }
                  if ((i == 0) || ((*n == '(') || (*n == ')'))) return NULL;
                  *(n + i) = '\0';
                  if ((strcmp(n, "+")) && (strcmp(n, "-")))
                  if (i != 0) {
                        i--;
                        for (i; i >= 0; i--) {
                              if ((i == 0) && (*(n + i) == '-') || *(n + i) == '+')
continue;
                              if ((*(n + i) >= '0') && (*(n + i) <= '9')) continue;
                              else break;
                        }
                  }
                  if (i == -1) {
                        el->el_type = NUMBER;
                        el->el.num = atoll(n);
                  }
                  else {
                        if (!strcmp(n, "exp")) el->el_type = FUNCTION;
                        else
                              if  (!(strcmp(n,  "+"))  ||  !(strcmp(n,  "-"))  ||
!(strcmp(n, "*")) || !(strcmp(n, "^"))) {
                                    el->el_type = OPERATION;
                                    //el->el.str = n;
                              }
                              else {
                                    for (i = 0; i < strlen(n); i++)
                                          if (!isalpha(*(n + i))) return NULL;
                                    el->el_type = VARIABLE;
                                    //el->el.str = n;
                              }
                        el->el.str = n;
                        }
                        return el;
            }

            void print_el(std::ostream &output, alg_el* el) {
                  if (el) {
                        switch (el->el_type) {
                        case NUMBER: output << el->el.num; break;
                        case OPERATION: output << el->el.str; break;
                        case VARIABLE: output << el->el.str; break;
                        case FUNCTION: output << el->el.str; break;
```

```
                }
            }
        }

        alg_el* make_el(const char* str, type t) {
            alg_el* res = new alg_el();
            res->el.str = (char*)malloc(4 * sizeof(char));
            res->el_type = t;
            strcpy(res->el.str, str);
            return res;
        }

        alg_el* make_el(int num, type t) {
            alg_el* res = new alg_el();
            res->el_type = t;
            res->el.num = num;
            return res;
}
```

Приложение B. Содержимое файла hlist.h

```
#pragma once
#include <fstream>
#include <iostream>

namespace h_list {

template <class el>
        struct two_ptr {
            el* head;
            el* tail;
        };

template <class base>
        struct hlist {
            bool tag;
            union {
                base atom;
                two_ptr<hlist> list;
            } node;
        };

template <class base>
        bool atomic(const hlist<base>* ptr);

template <class base>
        bool empty(const hlist<base>* ptr);

template <class base>
        hlist<base>* head(const hlist<base>* ptr);
```

```
	template <class base>
		hlist<base>* tail(const hlist<base>* ptr);

	template <class base>
		hlist<base>* cons(const hlist<base>* head, const hlist<base>* tail);

	template <class base>
		hlist<base>* make_atom(const base el);

	template <class base>
		base get_atom(const hlist<base>* ptr);

	template <class base>
		void destroy(hlist<base>* ptr);

	template <class base>
		hlist<base>* copy(const hlist<base>* ptr);

	#include "hlist_impl.h"

}
```

Приложение Г. Содержимое файла hlist_impl.h

```
#pragma once
template <class base>
	bool atomic(const hlist<base>* ptr) {
		return ptr && ptr->tag;
	}

template <class base>
	bool empty(const hlist<base>* ptr) {
		return !ptr;
	}

template <class base>
	hlist<base>* head(const hlist<base>* ptr) {
		if (!ptr) return NULL;
		if (!atomic(ptr)) return ptr->node.list.head;
		else return NULL;
	}

template <class base>
	hlist<base>* tail(const hlist<base>* ptr) {
		if (!atomic(ptr)) return ptr->node.list.tail;
		else return NULL;
	}

template <class base>
```

```
	hlist<base>* cons(hlist<base>* head, hlist<base>* tail) {
		hlist<base>* new_node = new hlist<base>;
		if (atomic(tail) || !new_node) return NULL;
		new_node->tag = false;
		new_node->node.list.head = head;
		new_node->node.list.tail = tail;
		return new_node;
	}

template <class base>
	hlist<base>* make_atom(const base el) {
		hlist<base>* new_atom = new hlist<base>;
		if (!new_atom) return NULL;
		new_atom->tag = true;
		new_atom->node.atom = el;
		return new_atom;
	}

template <class base>
	base get_atom(const hlist<base>* ptr) {
		if (atomic(ptr)) return ptr->node.atom;
		else {
			std::cerr << "get_atom(!atomic)";
			std::cin.get();
		}
	}

template <class base>
void destroy(hlist<base>* ptr) {
	if (ptr) {
		if (!atomic(ptr)) {
			destroy(head(ptr));
			destroy(tail(ptr));
		}
		delete ptr;
	}
}

template <class base>
hlist<base>* copy(const hlist<base>* ptr) {
	if (empty(ptr)) return NULL;
	hlist<base>* res = new hlist<base>;
	if (!res) return NULL;
	if (ptr->tag == true) {
		res = make_atom(get_atom(ptr));
	}
	else {
		res->tag = false;
		res->node.list.head = copy(head(ptr));
```

```
            res->node.list.tail = copy(tail(ptr));
      }
      return res;
}
```

Приложение Д. Содержимое файла lab2.cpp

```
      #include "stdafx.h"
      #include <iostream>
      #include <fstream>
      #include <string.h>
      #include <stdlib.h>
      #include "hlist.h"
      #include "alg_expr.h"

      typedef h_list::hlist<alg_el>* lisp;

      std::ofstream outf("output.txt", std::ofstream::out);
      enum err_code {NO_FILE, NO_CLOSING, NO_OPENING, NO_VARIABLE, NOT_ENOUGH_ARGS,\
                           UNEXPECTED_SYM, EXPECTED_OP, TOO_MANY_ARGS, EXCESS_C};

      lisp lst = NULL;
      alg_el* diff = NULL;

      //flag for definition of second call for error_handler
      bool fin = 0;

      void read_expr(lisp& d, int ldepth, alg_el* op, std::istream &in, std::ostream
&out);
      void read_seq(alg_el* prev, lisp& d, int ldepth, alg_el* op, std::istream &in,
std::ostream &out);
      void print_expr(const lisp& d, std::ostream& out);
      void error_handler(err_code code, std::ostream& out);
      void print_seq(const lisp& x, std::ostream& out);

      void read_expr(lisp& d, int ldepth, alg_el* op, std::istream &in, std::ostream
&out) {
            alg_el* el = NULL;
            lisp p1, p2;
            p1 = NULL;
            p2 = NULL;
            char c;
            do in >> c; while (c == ' ' );

            if (in.eof()) {
                  d = NULL;
                  return;
            }
```

```
            out << ldepth << "." << c;
            for (int i = 0; i < ldepth; i++) out << "\t.";
            out << " ["<< (op ? op->el.str : " ") << "]" << std::endl;

            if (c == '(') {
                  read_seq(NULL, p1, 0, NULL, in, out);

                  do in >> c; while (c == ' ');
                  if (c != ')') error_handler(NO_CLOSING, out);

                        if(!atomic(head(p1)) && !op)
                        error_handler(EXPECTED_OP, out);

                  read_expr(p2, ldepth + 1, op, in, out);

                  d = h_list::cons(p1, p2);
                  return;
            }
            else {

                  if (c == ')') {
                        in.unget();
                        if (op) {
                              if (!strcmp(op->el.str, "^")) {
                                    if (ldepth < 3) error_handler(NOT_ENOUGH_ARGS,
out); else
                                          if (ldepth > 3)
error_handler(TOO_MANY_ARGS, out);
                              }
                              else
                                    if (!strcmp(op->el.str, "exp")) {
                                          if (ldepth == 0)
error_handler(NOT_ENOUGH_ARGS, out);
                                          else
                                                if (ldepth > 1)
error_handler(TOO_MANY_ARGS, out);
                                    }
                                    else
                                          if (!strcmp(op->el.str, "-")) {
                                                if (ldepth < 2)
error_handler(NOT_ENOUGH_ARGS, out);
                                          }
                                          else
                                                if (ldepth < 3)
error_handler(NOT_ENOUGH_ARGS, out);
                        }
                        else
                              error_handler(EXPECTED_OP, out);
```

```
                        d = NULL;

                  }
                  else {

                        in.unget();
                        el = read_el(in);

                        if (!el) error_handler(NO_VARIABLE, out);

                        if (ldepth == 0) {
                              if (!(op && op->el_type == FUNCTION))
                                          if (el->el_type != OPERATION && el->el_type
!= FUNCTION) error_handler(EXPECTED_OP, out);
                        }

                        if (el->el_type == OPERATION) {
                              op = el;
                              if (op && op->el_type == FUNCTION && ldepth == 1);
                              else
                                    if (ldepth != 0) error_handler(UNEXPECTED_SYM,
out);
                        }
                        else {

                              if (el->el_type == FUNCTION) {
                                    do in >> c; while (c == ' ');
                                    if (c != '(') error_handler(NO_OPENING, out);

                                    out << ldepth << "." << c;
                                    for (int i = 0; i < ldepth; i++) out << "\t.";
                                    out << " [" << (op ? op->el.str : " ") << "]" <<
std::endl;


                                    lisp p3 = h_list::make_atom(*el);
                                    read_seq(NULL, p1, 0, el, in, out);
                                    lisp p4;

                                    if             (!atomic(head(p1))             ||
(h_list::get_atom(head(p1)).el_type == VARIABLE \
                                                                        ||
h_list::get_atom(head(p1)).el_type == NUMBER))
                                    {
                                          p4 = h_list::cons(p3, p1);
                                    }
                                    else {
                                          lisp nil = NULL;
```

```
                                        p4   =   h_list::cons(p3,   h_list::cons(p1,
nil));
                                  }
                                  do in >> c; while (c == ' ');
                                  if (c != ')') error_handler(NO_CLOSING, out);

                                  read_expr(p2, ldepth + 1, op, in, out);
                                  d = h_list::cons(p4, p2);
                                  return;
                            }
                            else {
                                  if (!op) error_handler(EXCESS_C, out);
                            }
                      }

                      read_seq(el, p1, ldepth + 1, op, in, out);
                      read_expr(p2, ldepth + 1, op, in, out);
                      d = h_list::cons(p1, p2);
                }
          }
     }

     void error_handler(err_code code, std::ostream& out) {

          out << "\n! - ";
          switch (code) {
          case NO_FILE:
                out << "No input file.";
                std::cout << "No input file. Please, pass input file as an argument
to program \
     (or click and drag input file onto executable file).\n";
                break;
          case NO_CLOSING: out << "Expected closing bracket"; break;
          case NOT_ENOUGH_ARGS: out << "Not enough arguments"; break;
          case UNEXPECTED_SYM: out << "Unexpected symbol"; break;
          case EXPECTED_OP: out << "Expected operation"; break;
          case TOO_MANY_ARGS:out << "Too many arguments"; break;
          case NO_OPENING: out << "Expected opening bracket"; break;
          case EXCESS_C: out << "Excess characters"; break;
          case NO_VARIABLE: out << "Expected variable or constant as an argument";
break;
          }
          out << '\n';
          if (fin == 0) {
                fin = 1;
                error_handler(code, std::cout);
          }

          h_list::destroy(lst);
```

```
            #ifdef _WIN32
                  system("PAUSE");
            #endif
            exit(1);
      }

      void read_seq(alg_el* prev, lisp& d, int ldepth, alg_el* op, std::istream &in,
std::ostream &out) {
            if (prev) {
                  d = h_list::make_atom(*prev);
            }
            else read_expr(d, ldepth, op, in, out);
      }

      void print_expr(const lisp& d, std::ostream& out)
      {
            if (h_list::empty(d)) out << "()";
            else if (h_list::atomic(d)) {
                  print_el(out, &h_list::get_atom(d));
                  out << ' ';
            }
            else {
                  if (h_list::atomic(h_list::head(d)) && head(d)->node.atom.el_type
== FUNCTION) {
                        if (h_list::atomic(head(h_list::tail(d)))) {
                              print_el(out, &h_list::get_atom(head(d)));
                              print_expr(tail(d), out);
                        }
                        else {
                              print_el(out, &h_list::get_atom(head(d)));
                              print_expr(head(tail(d)), out);
                        }
                  }
                  else {
                        out << "( ";
                        print_seq(d, out);
                        out << ") ";
                  }
            }
      }

      void print_seq(const lisp& x, std::ostream& out)
      {
            if (!h_list::empty(x)) {
                  print_expr(head(x), out);
                  print_seq(tail(x), out);
            }
      }
```

```
	lisp derivative(const lisp& d);
	lisp derivPM(const lisp& c);
	lisp derivMul(const lisp& c, int depth);
	lisp derivPow(const lisp& c, int step);
	lisp derivExp(const lisp& c);

	lisp derivPM(const lisp& c) {
		//(a +- b +- c +- ...)' = a' +- b' +- c' +- ...
		if (h_list::empty(c)) return NULL;
		return	h_list::cons(	derivative(h_list::head(c)),	derivPM(
h_list::tail(c) ));
	}

	lisp derivMul(const lisp& c, int depth) {
		//(a*b*c*...)' = a'*b*c*... + a*b'*c*... + a*b*c'*... + ...
		lisp x;
		x = h_list::copy(c);
		lisp d;
		d = x;
		for (int i = 0; i < depth; i++) {
			d = tail(d);
		}
		if (h_list::empty(d)) return NULL;
		lisp j = NULL;
		j = derivative(h_list::head(d));
		*head(d) = *j;
		return h_list::cons(x, derivMul(c, depth + 1));
	}

	lisp derivPow(const lisp& c, int step) {
		//(u^v)' = v*u^(v-1)*u' + u^(v)*v'*ln(u)
		lisp nil = NULL;
		switch (step) {
		case 0: return h_list::cons(h_list::make_atom(*make_el("+", OPERATION)),
derivPow(c,1));
		case 1: return h_list::cons(derivPow(c, 2), h_list::cons(derivPow(c, 9),
nil));
		case 2: return h_list::cons(h_list::make_atom(*make_el("*", OPERATION)),
derivPow(c, 3));
		case	3:	return	h_list::cons(h_list::copy(head(tail(c))),
h_list::cons(derivPow(c, 4),\
				h_list::cons(derivative(head(c)),nil)));
		case 4: return h_list::cons(h_list::make_atom(*make_el("^", OPERATION)),
derivPow(c,5));
		case	5:	return	h_list::cons(h_list::copy(head(c)),
h_list::cons(derivPow(c, 6), nil));
		case 6: return h_list::cons(h_list::make_atom(*make_el("-", OPERATION)),
derivPow(c, 7));
```

```
            case 7: return h_list::cons(h_list::copy(head(tail(c))), derivPow(c,
8));
            case 8: return h_list::cons(h_list::make_atom(*make_el(1, NUMBER)),
nil);
            case 9: return h_list::cons(h_list::make_atom(*make_el("*", OPERATION)),
derivPow(c, 10));
            case 10: return h_list::cons(derivPow(c,11),
h_list::cons(derivative(head(tail(c))),derivPow(c,12)));
            case 11: return h_list::cons(h_list::make_atom(*make_el("^",
OPERATION)),\
                              h_list::cons(h_list::copy(head(c)),
h_list::cons(h_list::copy(head(tail(c))), nil)));
            case 12: return h_list::cons(h_list::make_atom(*make_el("ln",
FUNCTION)),\
                                h_list::cons(h_list::copy(head(c)), nil));
            default: break;
            }
            return NULL;
      }

      lisp derivExp(const lisp& c){
            //(e^u)' = u'*e^u
            lisp nil = NULL;
            lisp ex = h_list::copy(c);
            return h_list::cons(h_list::make_atom(*make_el("*", OPERATION)),\
                     h_list::cons(derivative(head(tail(c))),       h_list::cons(ex,
nil)));
      }

      lisp derivative(const lisp& d) {
            if (h_list::empty(d)) return NULL;
            lisp res = NULL;
            lisp c = NULL;

            alg_el* nil = make_el(0,NUMBER);

            if (h_list::atomic(d)) {

                  if (h_list::get_atom(d).el_type == VARIABLE) {
                        if (!strcmp(h_list::get_atom(d).el.str , diff->el.str)) {
                              res = h_list::make_atom(*nil);
                              std::cout << "Met diff\n";
                              res->node.atom.el.num = 1;
                        }
                        else {
                              std::cout << "Met var\n";
                              res = h_list::make_atom(*nil);
                        }
                  } else
```

```
                if (h_list::get_atom(d).el_type == NUMBER) {
                      std::cout << "Met const\n";
                      res = h_list::make_atom(*nil);
                }
                return res;

          } else

                if (h_list::atomic(head(d))) {
                      if (!strcmp(h_list::get_atom(head(d)).el.str, "+") \
                            || !strcmp(h_list::get_atom(head(d)).el.str, "-")) {
                            std::cout << "Met +-\n";
                            c = tail(d);
                            lisp p1 = derivPM(c);
                            lisp                        atm                        =
h_list::make_atom(h_list::get_atom(head(d)));
                            res = h_list::cons(atm, p1);
                            std::cout << "Exiting +-\n";
                            return res;
                      }
                      if (!strcmp(h_list::get_atom(head(d)).el.str, "*")) {
                            std::cout << "Met *\n";
                            c = d;
                            lisp p1 = derivMul(c, 1);
                            lisp atm = h_list::make_atom(*make_el("+", OPERATION));
                            res = h_list::cons(atm, p1);
                            std::cout << "Exiting *\n";
                            return res;
                      }
                      if (!strcmp(h_list::get_atom(head(d)).el.str, "^")) {
                            std::cout << "Met ^\n";
                            res = derivPow(tail(d), 0);
                            std::cout << "Exiting ^\n";
                            return res;
                      }
                      if (!strcmp(h_list::get_atom(head(d)).el.str, "exp")) {
                            std::cout << "Met exp\n";
                            res = derivExp(d);
                            std::cout << "Exiting exp\n";
                            return res;
                      }
                }
                else
                      return derivative(head(d));
          return NULL;
     }

     void displayFileContents(std::ifstream* in) {
          std::cout << "File contents:\n";
```

```
            std::cout << "-begin-\n";
            char c;
            c = in->get();
            while (!in->eof()) {
                  std::cout << c;
                  c = in->get();
            }
            std::cout << "\n-end-\n";
      }

      int main(int argc, char* argv[]) {

            if (argc != 2) {
                  std::cout << "Please, enter the variable for which you want to take
the derivative.\n";
                  diff = read_el(std::cin);
                  if (!diff || diff->el_type != VARIABLE) error_handler(NO_VARIABLE,
outf);
                  std::cout << "Enter algebraic function (use ctrl-Z (ctrl-D for unix)
to determine end of input):\n";
                  read_expr(lst, 0, NULL, std::cin, outf);
                  if (!std::cin.eof()) error_handler(EXCESS_C, outf);
            }
            else {
                  std::ifstream in(argv[1], std::ifstream::in);
                  //std::ifstream in("test.txt", std::ifstream::in);
                  displayFileContents(&in);
                  in.clear();
                  in.seekg(0);
                  diff = read_el(in);

                  if (!diff) {
                        std::cout << "Please, enter the variable for which you want
to take the derivative.\n";
                        diff = read_el(std::cin);
                  }
                  if (diff->el_type != VARIABLE) error_handler(NO_VARIABLE, outf);
                  read_expr(lst, 0, NULL, in, outf);
                  if (!in.eof()) error_handler(EXCESS_C, outf);
            }

            lisp res = NULL;
            res = derivative(lst);
            outf << "derivative:\n";
            std::cout << "derivative:\n";
            print_expr(res, outf);
            print_expr(res, std::cout);

            std::cout << '\n';
```

```
#ifdef _WIN32
      system("PAUSE");
#endif
      h_list::destroy(lst);
      h_list::destroy(res);
    return 0;
}
```

Приложение Е. Графическая схема примера иерархического списка, используемого в данной работе

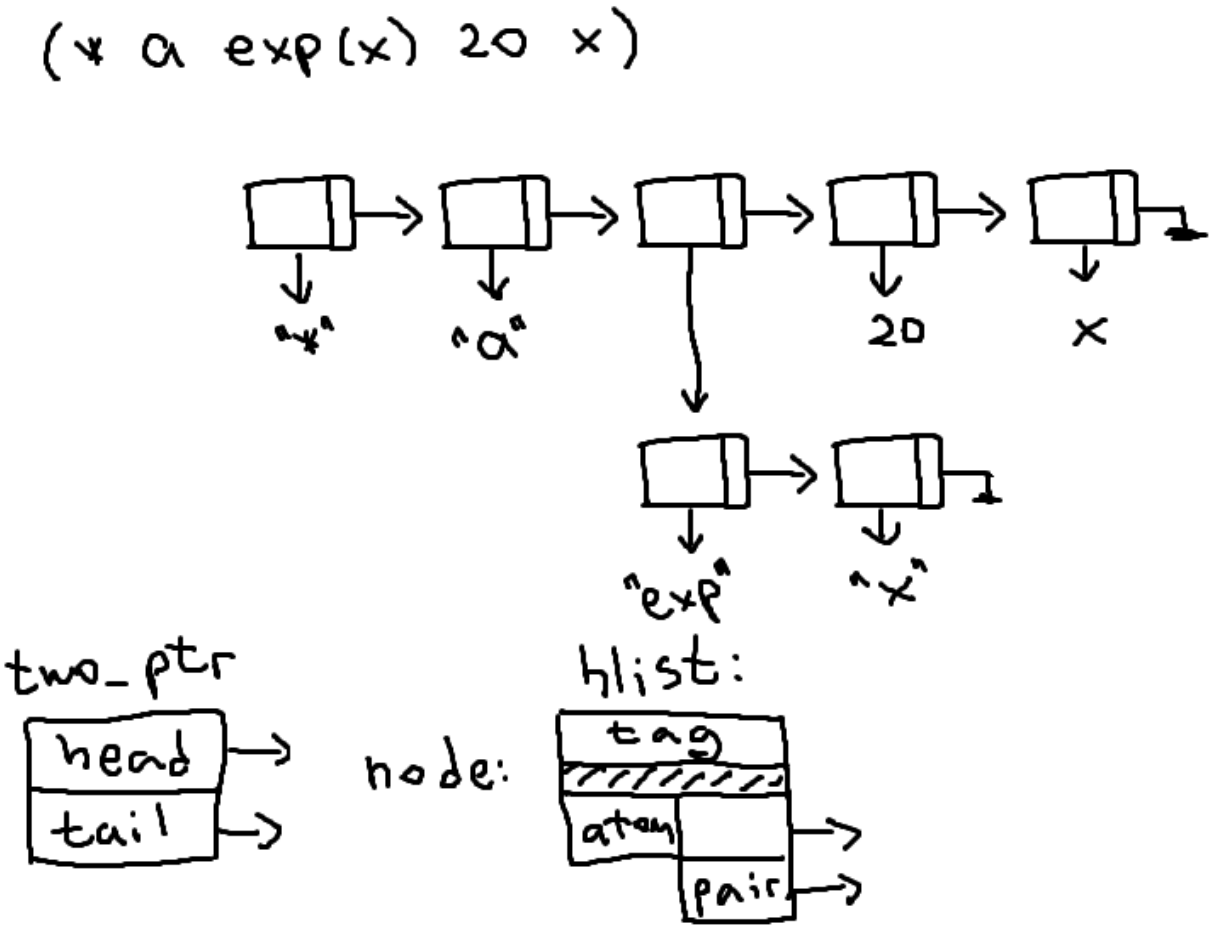

Рисунок 1 – Графическая схема примера иерархического списка.