**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра МО ЭВМ**

**ОТЧЕТ**

**по лабораторной работе № 2**

**по дисциплине «Программирование»**

**Тема: Рекурсивная обработка иерархических списков**

|  |  |  |
| --- | --- | --- |
| Студент гр. 6383 |  | Никитин К.В. |
| Преподаватель |  | Шолохова О.М. |

Санкт-Петербург

2017

**Цель работы.**

Познакомиться с одной из часто используемых на практике нелинейных конструкций, 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.

**Постановка задачи.**

Задание:

Проверить иерархический список на наличие в нем заданного элемента (атома) x.

**Основные теоретические положения.**

1. Рекурсивное определение иерархического списка

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*) >

## 2. Представление иерархического списка

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.



***a***



***e***



***d***



***c***



***b***



Рис. 1. Пример представления иерархического списка в виде двумерного рисунка

Переход от полной скобочной записи, соответствующей определению иерархического списка, к сокращенной производится путем отбрасывания конструкции . Nill и удаления необходимое число раз пары скобок вместе с предшествующей открывающей скобке точкой.

|  |  |
| --- | --- |
| Полная запись | Сокращенная запись |
| ***a***  *Nil*  (***a* .** (***b* .**(***c* .***Nil*)))  (***a* .**((***b* .** (***c* .***Nil*)) **.**(***d* .**(***e* .***Nil*)))) | ***a***  (  )  (***a******b******c***)  (***a*** (***b******c***) ***d******e***) |

Рис. 2. Примеры перехода от полной к сокращенной скобочной записи иерархических списков

Согласно приведенному определению иерархического списка, структура непустого иерархического списка － это элемент размеченного объединения множества атомов и множества пар «голова-хвост».

## 3. Рекурсивная структура иерархического списка на языке С++

1. typedef char base; // базовый тип элементов (атомов)
2. struct s\_expr;
3. struct two\_ptr {
4. s\_expr \*hd;
5. s\_expr \*tl;
6. } ; //end two\_ptr;
7. struct s\_expr {
8. bool tag; // true: atom, false: pair
9. union {
10. base atom;
11. two\_ptr pair;
12. } node; //end union node
13. }; //end s\_expr
14. typedef s\_expr \*list;

Поясняющие эту структуру иллюстрации представлены на рисунке 3.

two\_ptr:

s\_expr:

hd

tl

tag

node:

atom

pair

Рис. 3. Представление рекурсивной структуры списка

## 2.4. Функциональная спецификация и реализация иерархического списка

Функциональная спецификация иерархического списка включает:

* функции － селекторы Head и Tail , выделяющие соответственно «голову» и «хвост» списка
* функции － конструкторы: Cons, создающая точечную пару (новый список из «головы» и «хвоста»), и Make\_Atom, создающая атомарное S-выражение.
* предикаты Is Null, проверяющий список на отсутствие в нем элементов, и Atom, проверяющий, является ли список атомом.

Необходимо включить в список базовых функций и функцию Destroy, позволяющую уничтожить созданный список, т.е. освободить псмять от ставших ненужными списочных структур.

При реализации иерархического списка, рассматриваемого нами, как абстрактный тип данных (АТД), на языке С++, поместим прототипы всех перечисленных выше базовых функций в отдельный файл с расширением .h. list head (const list s);

list **tail** (const list s);

list **cons** (const list h, const list t);

list **make\_atom** (const base x);

bool **isAtom** (const list s);

bool **isNull** (const list s);

void **destroy** (list s);

Файл реализации этих функций имеет расширение .cpp.

list **head** (const list s)

{// PreCondition: not null (s)

if (s != NULL)

if (!isAtom(s)) return s->node.pair.hd;

else { cerr << "Error: Head(atom) \n"; exit(1); }

else { cerr << "Error: Head(nil) \n";

exit(1);

}

}

Если «голова» списка не атом, то функция **head** возвращает список, на который указывает голова пары, т.е. подсписок, находящийся на следующем уровне иерархии. Если же «голова» списка － атом, то выводится сообщение об ошибке и функция прекращает работу.

list **tail** (const list s)

{// PreCondition: not null (s)

if (s != NULL)

if (!isAtom(s)) return s->node.pair.tl;

else { cerr << "Error: Tail(atom) \n"; exit(1); }

else { cerr << "Error: Tail(nil) \n";

exit(1);

}

}

bool **isAtom** (const list s)

{ if(s == NULL) return false;

else return (s -> tag);

}

//.......................................

bool **isNull** (const list s)

{ return s==NULL;

}

Предикат **isAtom** возвращает значение tag, которое равно True, если элемент － атом, и значение False, если － «голова-хвост». В случае пустого списка значение предиката False.

list **cons** (const list h, const list t)

// PreCondition: not isAtom (t)

{list p;

if (isAtom(t)) { cerr << "Error: cons(\*, atom) \n"; exit(1);}

else { p = new s\_expr;

if ( p == NULL) {cerr << "Memory ...\n"; exit(1); } else {

p->tag = false;

p->node.pair.hd = h;

p->node.pair.tl = t;

return p;

} } }

Функция **Сons** － конструктор. При создании нового S-выражения требуется выделение памяти. Если памяти нет, то p == NULL и это приводит к выводу соответствующего сообщения об ошибке. Если «хвост» － не атом, то для его присоединения к «голове» требуется создать новый узел (элемент), головная ссылка которого будет ссылкой на «голову» этого «хвоста», а хвостовая часть элемента (tag.hd.tl) － ссылкой на его «хвост»

list **make\_atom** (const base x)

{ list s;

s = new s\_expr;

s -> tag = true;

s->node.atom = x;

return s;

}

Создается узел типа (tag, x).

void **destroy** (list s)

{

if ( s != NULL) {

if (!isAtom(s)) {

destroy ( head (s));

destroy ( tail(s));

}

**delete** s;

};

}

Функция **delete** удаляет текущий элемент списка.

Вся работа с иерархическими списками осуществляется только с помощью описанных выше базовых функций.

Над иерархическими списками могут выполняться, например, следующие операции:

* добавление нового элемента (списка или атома),
* удаление всех вхождений заданного элемента (атома),
* проверка иерархического списка на наличие в нем заданного элемента (атома),
* замена всех вхождений одного элемента на другой,
* подсчет числа атомов в списке или числа одинаковых атомов,
* проверка идентичности 2-х иерархических списков,
* обращение списка
* вычисление глубины (числа уровней вложения)

Следует помнить, что функция Cons не формирует копий исходных S-выражений, т.о. программист должен сам заботиться о копировании списочных структур, избегая побочного эффекта от включения одних списков в другие в качестве фрагментов. Т.о. в дополнение к базовым функциям в состав модуля реализации иерархического списка необходимо включить функцию копирования списка Copy\_List.

list **copy\_list** (const list x)

{ if (isNull(x)) return NULL;

else if (isAtom(x))

return make\_atom (x->node.atom);

else return cons (copy\_list (head (x)),

copy\_list (tail(x)));

} //end copy-list

Эта функция, однако, использует уже описанные выше базовые функции **Cons, Head, Tail, Make\_Atom**, а также предикаты **IsNull** и **IsAtom**.

Кроме того, какую бы задачу мы ни решали, нам потребуются процедуры ввода/вывода иерархического списка. Эти процедуры должны быть написаны применительно к конкретной форме представления списков. Поскольку иерархические списки чаще всего представляются сокращенной скобочной записью, то можно предложить следующую процедуру вывода иерархического списка:

// функции вывода:

void write\_list (const list x); // основная

void write\_seq (const list x);

void write\_list (const list x)

{//пустой список выводится как ()

if (isNull(x)) cout << " ()";

else if (isAtom(x)) cout << ' ' << x->node.atom;

else { //непустой список

cout << " (" ;

write\_seq(x);

cout << " )";

}

} // end write\_list

void write\_seq (const list x)

{

/\*выводит последовательность элементов списка без обрамляющих его скобок \*/

if (!isNull(x)) {

write\_list(head (x));

write\_seq(tail (x));

}

}

Здесь процедура вывода списка с обрамляющими его скобками － write\_list, а без обрамляющих скобок － write\_seq. Процедура write\_list использует внутри себя вызов процедуры write\_seq.

Для ввода иерархического списка, представленного сокращенной скобочной записью, при условии ввода этой записи с клавиатуры можно предложить процедуру read\_list. Эта процедура использует внутри себя обращение к процедуре read\_s\_expr, а она, в свою очередь, обращение к read\_seq.

// функции ввода:

void read\_list ( list& y); // основная

void read\_s\_expr (base prev, list& y);

void read\_seq ( list& y);

// ввод списка с консоли

void read\_list ( list& y)

{ base x;

do cin >> x; while (x==' ');

read\_s\_expr ( x, y);

} //end read\_list

void read\_s\_expr (base prev, list& y)

{ //prev － ранее прочитанный символ

if ( prev == ')' ) {cerr << " ! List.Error 1 " << endl; exit(1); }

else if ( prev != '(' ) y = make\_atom (prev);

else read\_seq (y);

} //end read\_s\_expr

void read\_seq ( list& y)

{ base x;

list p1, p2;

if (!(cin >> x)) {cerr << " ! List.Error 2 " << endl; exit(1);}

else {

while ( x==' ' ) cin >> x;

if ( x == ')' ) y = NULL;

else { read\_s\_expr ( x, p1);

read\_seq ( p2);

y = cons (p1, p2);

}

}

} //end read\_seq

Язык С++ поддерживает парадигмы процедурного, модульного и объектно-ориентированного программирования. Используя модульное программирование, получим проект, содержащий 3 файла:

* Заголовочный файл “l\_intrfc.h”
* Файл реализации “l\_intrfc.cpp”
* Файл с клиентской программой “main.cpp”, которая использует структуры данных и функции, объявленные в интерфейсе (“l\_intrfc.h”)

**Спецификация программы.**

*Назначение программы*.

Программа предназначена для определения, есть ли в иерархическом списке определённый элемент.

*Описание программы*.

Программа написана на языке С++ с использованием компилятора g++ 5.4.0 Входными данными для программы является иерархический список вводимый пользователем с клавиатуры и символ, который будем искать в данном спике. Выходными данными является информация, есть ли данный элемент в списке или нет.

*Пример диалога с пользователем*.

Enter an element you want to search

t

Enter a list

(a b c (d (e f (h ))))

Input list:

(a b c (d (e f (h ))))

t is existing

*Реализация*.

Реализацию функций

list head (const lisp s);

list tail (const lisp s);

list cons (const lisp h, const lisp t);

list make\_atom (const base x);

bool isAtom (const lisp s);

bool isNull (const lisp s);

void destroy (lisp s);

base getAtom (const lisp s);

void read\_list ( lisp& y);

void read\_s\_expr (base prev, lisp &y);

void read\_seq ( lisp& y);

void write\_list (const lisp x);

void write\_seq (const lisp x);

list copy\_list (const lisp x);

а так же структуры s\_expr и two\_ptr смотри в «Основные теоретические положения»

Функции:

bool found(const base &x, const lisp l);

Определение, входит ли элемент х в состав списка l.

Входные параметры:

Элемент х, который будем искать в l.

Выходные параметры:

отсутствуют

Возвращаемое значение:

True, если х входит в состав l; false – если не входит.

Предусловие:

l – не пустой список.

Постусловие:

Если найден атом со значением х, возвращается true;

Если по прошествии всего списка нужный атом не найден, возвращается false

Побочные действия:

Пробегает по всему списку: сначала смотрит голову элемента (если это не атом – просматривает голову головы), если атом не найден, возвращается на шаг назад и просматривает хвост аналогичным образом. Повторяет действие пока не найдёт нужный элемент, или не останется непроверенных хвотов.

**Тестирование.**

Список:

(a b c (d (e f (h ) g (k l) m) n ) o )

|  |  |
| --- | --- |
| **Искомый элемент** | **Результат** |
| q | Отсутствует |
| w | Отсутствует |
| e | Присутствует |
| a | Присутствует |
| b | Присутствует |
| x | Отсутствует |

**Выводы.**

В ходе выполнения лабораторной работы по рекурсивной обработке рекурсивных списков мы познакомились с одной из часто используемых на практике нелинейных конструкций, способами её организации и рекурсивной обработки, получили навыки решения задач обработки иерархических списков, с использованием базовых функций их рекурсивной обработки.

Реализовать решение задачи нерекурсивно не получилось ввиду того, что сам по себе нерекурсивный обход иерархического списка во много раз труднее реализуем, нежели его рекурсивный аналог.

**Приложение А. Исходный код.**

l\_intrfc.h

namespace h\_list

{

typedef char base; // базовый тип элементов (атомов)

struct s\_expr;

struct two\_ptr

{

s\_expr \*hd;

s\_expr \*tl;

} ; //end two\_ptr;

struct s\_expr {

bool tag; // true: atom, false: pair

union

{

base atom;

two\_ptr pair;

} node; //end union node

}; //end s\_expr

typedef s\_expr \*lisp;

// функции

void print\_s\_expr( lisp s );

// базовые функции:

lisp head (const lisp s);

lisp tail (const lisp s);

lisp cons (const lisp h, const lisp t);

lisp make\_atom (const base x);

bool isAtom (const lisp s);

bool isNull (const lisp s);

void destroy (lisp s);

base getAtom (const lisp s);

// функции ввода:

void read\_lisp (lisp& y); // основная

void read\_s\_expr (base prev, lisp& y);

void read\_seq ( lisp& y);

// функции вывода:

void write\_lisp (const lisp x); // основная

void write\_seq (const lisp x);

lisp copy\_lisp (const lisp x);

}

l\_intrfc.cpp

#include "l\_intrfc.h"

#include <iostream>

#include <fstream>

#include <cstdlib>

//std::ifstream fin("input.txt");

using namespace std;

namespace h\_list

{

//....................................

lisp head (const lisp s)

{// PreCondition: not null (s)

if (s != NULL) if (!isAtom(s)) return s->node.pair.hd;

else { cerr << "Error: Head(atom) \n"; exit(1); }

else { cerr << "Error: Head(nil) \n";

exit(1);

}

}

//.......................................

bool isAtom (const lisp s)

{ if(s == NULL) return false;

else return (s -> tag);

}

//.......................................

bool isNull (const lisp s)

{ return s==NULL;

}

//.......................................

lisp tail (const lisp s)

{// PreCondition: not null (s)

if (s != NULL) if (!isAtom(s)) return s->node.pair.tl;

else { cerr << "Error: Tail(atom) \n"; exit(1); }

else { cerr << "Error: Tail(nil) \n";

exit(1);

}

}

//.......................................

lisp cons (const lisp h, const lisp t)

// PreCondition: not isAtom (t)

{lisp p;

if (isAtom(t)) { cerr << "Error: Cons(\*, atom)\n"; exit(1);}

else {

p = new s\_expr;

if ( p == NULL) {cerr << "Memory not enough\n"; exit(1); }

else {

p->tag = false;

p->node.pair.hd = h;

p->node.pair.tl = t;

return p;

}

}

}

//...........................

lisp make\_atom (const base x)

{ lisp s;

s = new s\_expr;

s -> tag = true;

s->node.atom = x;

return s;

}

//...........................

void destroy (lisp s)

{

if ( s != NULL) {

if (!isAtom(s)) {

destroy ( head (s));

destroy ( tail(s));

}

delete s;

// s = NULL;

};

}

//...........................

base getAtom (const lisp s)

{

if (!isAtom(s)) { cerr << "Error: getAtom(s) for !isAtom(s) \n"; exit(1);}

else return (s->node.atom);

}

//...........................

// ввод списка с консоли

void read\_lisp (lisp& y)

{ base x;

do cin >> x; while (x==' ');

read\_s\_expr ( x, y);

} //end read\_lisp

//...........................

void read\_s\_expr (base prev, lisp& y)

{ //prev － ранее прочитанный символ}

if ( prev == ')' ) {cerr << " ! List.Error 1 " << endl; exit(1); }

else if ( prev != '(' ) y = make\_atom (prev);

else read\_seq (y);

} //end read\_s\_expr

//...........................

void read\_seq ( lisp& y)

{ base x;

lisp p1, p2;

if (!(cin >> x)) {cerr << " ! List.Error 2 " << endl; exit(1);}

else {

while ( x==' ' ) cin >> x;

if ( x == ')' ) y = NULL;

else {

read\_s\_expr ( x, p1);

read\_seq ( p2);

y = cons (p1, p2);

}

}

} //end read\_seq

//...........................

// Процедура вывода списка с обрамляющими его скобками － write\_lisp,

// а без обрамляющих скобок － write\_seq

void write\_lisp (const lisp x)

{//пустой список выводится как ()

if (isNull(x)) cout << " ()";

else if (isAtom(x)) cout << ' ' << x->node.atom;

else { //непустой список}

cout << " (" ;

write\_seq(x);

cout << " )";

}

} // end write\_lisp

//...........................

void write\_seq (const lisp x)

{//выводит последовательность элементов списка без обрамляющих его скобок

if (!isNull(x)) {

write\_lisp(head (x));

write\_seq(tail (x));

}

}

//...........................

lisp copy\_lisp (const lisp x)

{ if (isNull(x)) return NULL;

else if (isAtom(x)) return make\_atom (x->node.atom);

else return cons (copy\_lisp (head (x)), copy\_lisp (tail(x)));

} //end copy-lisp

}

main.cpp

#include <iostream>

#include <fstream>

#include "l\_intrfc.h"

using namespace h\_list;

using namespace std;

bool found(const base &x, const lisp l);

int main()

{

lisp l;

base x;

cout << "Ener an element you want to search" << endl;

cin >> x;

cout << "Enter a list" << endl;

read\_lisp (l);

cout << "Input list:" << endl;

write\_lisp(l);

cout << endl;

if (found(x, l) )

cout << x <<" is existing\n";

else

cout << x << " is not existing \n";

}

bool found(const base &x, const lisp l)

{

if( isAtom(l) )

if (l->node.atom == x) return true;

else return false;

if( found(x, l->node.pair.hd) ) return true;

if ( !l->node.pair.tl ) return false;

if( found(x, l->node.pair.tl) ) return true;

return false;

}