**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МОЭВМ**

отчет

**по лабораторной работе №2**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: Рекурсивная обработка иерархических списков**

|  |  |  |
| --- | --- | --- |
| **Студент гр. 7381** |  | **Машина Ю. Д.** |
| **Преподаватель** |  | **Фирсов М. А.** |

**Санкт-Петербург**

**2018**

**Цель работы.**

Ознакомиться с основными методами рекурсивной обработки иерархических списков.

**Задание.**

Решить следующие задачи с использованием базовых функций рекурсивной обработки списков:

*Вариант 9.*

Подсчитать число атомов в иерархическом списке; сформировать линейный список атомов, соответствующий порядку подсчёта.

**Основные теоретические положения.**

Рассмотрим нелинейную конструкцию, 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)) >,

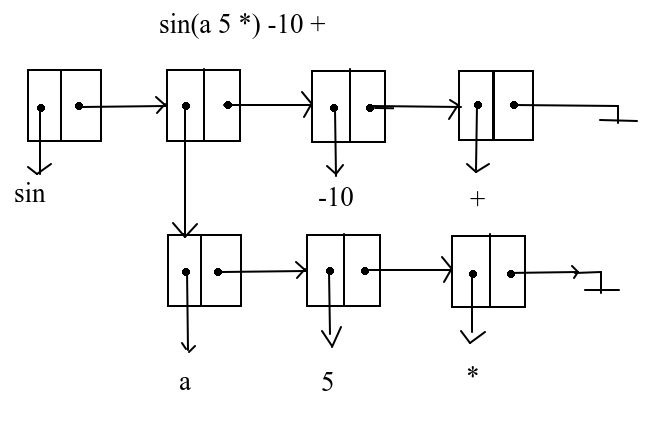
< Atomic (El) > ::= < El >

Иерархический список согласно определению представляет собой или элемент базового типа El, называемый в этом случае атомом (атомарным S-выражением), или линейный список из S-выражений. Приведенное определение задает структуру непустого иерархического списка как элемента размеченного объединения множества атомов и множества пар «голова»–«хвост» и порождает различные формы представления в зависимости от принятой формы представления линейного списка. Традиционно иерархические списки представляют или графически, используя для изображения структуры списка двухмерный рисунок, или в виде одномерной скобочной записи.

Пример: **(a (b c) d e)**

**Представление иерархических списков в программе.**

Каждый элемент иерархического списка является структурой *struct s\_expr,* содержащей поле *tag* булевого типа, где *true* соответствует атому, а *false* иерархическому списку.В зависимости от значения этого поля можно обратиться непосредственно к атому или к голове/хвосту иерархического списка. Атом представляется структурой *struct Base,* содержащей так же поле *tag*, которое позволяет различать тип элементов алгебраического выражения. Значение 0 соответствует численной константе, 1 — однобуквенной переменной или знаку операции, 2 — тригонометрической функции синуса или косинуса. Каждое значение позволяет обращаться к соответствующим полям: *num, ch, trig*, которые представлены объединением.

*Рис 1. Схематическое представление иерархического списка, содержащего алгебраическое выражение*

**Описание алгоритма.**

В программе осуществляется рекурсивный обход иерархического списка. Для каждого «уровня» существует свой стек для хранения численных констант и последующего вычисления алгебраического выражения. Поскольку сам иерархический список представляет собой пару *голова-хвост,* то очевидно, что переход на уровень ниже осуществляется из *головы.* Поскольку необходимо просмотреть все такие пары, то обобщенно можно сказать, что функция обхода вызывает сама себя два раза для каждого иерархического списка (очевидно, что с некоторыми условиями). В процессе обхода обрабатываются атомы выражения. Константы игнорируются, числа кладутся на стек, знак операции является признаком окончания текущего «уровня» и логически предшествует вызову функции упрощения.

Отметим, что программа ожидает на вход *корректное* алгебраическое выражение, посему проверка на отсутствие знака операции в конце списка или пустоту содержимого тригонометрических функций отсутствует. В программе так же используется структура стек, реализованная на базе массива.

**Описание функций.**

*Функция* ***int main()***

В главной функции производится считывание иерархического списка, после чего вызывается функция ***func***, подсчитывающая количество атомов этого списка, изучающая иерархический список и осуществляющая передвижение по нему.

*Функция* ***int func(lisp L, int count)***

***Рекурсивная функция***

*Аргументы:*

Текущий элемент списка *L,* счетчик *count.*

*Возвращаемое значение:*

Значение счетчика *count*.

*Описание:*

Производится изучение текущего элемента списка.

Если он **не** атом, то:

Если элемент в голове **не** атом, то к счетчику *count* прибавляется результат рекурсивно вызываемой функции ***func*** с указателем на голову *L->node.pair.hd* и счетчиком *count* в качестве аргументов, сложенный с результатом рекурсивно вызываемой функции ***func*** с указателем на хвост *L->node.pair.tl* и счетчиком *count* в качестве аргументов***.***

**Тестирование программы.**

Файлы с тестовыми данными, находящиеся в директории **Tests** и имеющие названия вида **Testn.txt (1 ≤ n ≤ 6)**, проверяют функционал и работоспособность написанной программы. Корректность работы программы протестирована как на пустом списке, так и на списках различной глубины и содержания.

|  |  |  |  |
| --- | --- | --- | --- |
|  | № | Входные данные | Результат работы программы |
|  | 1 | (a(b)(c)) | 3 |
|  | 2 | (ac(b)) | 3 |
|  | 3 | (a(b(c))) | 1 |
|  | 4 | (q(z)e) | 3 |
|  | 5 | (acd(b)(q(z))) | 6 |
| 6 | | () | Empty list. |

**Выводы.**

В ходе выполнения лабораторной работы получены знания об иерархических списках, а также закреплены умения и навыки с их обработкой.

Приложение а

Исходный код программы

#include <stdio.h>

#include "L\_func.h"

#include <ctype.h>

#include <stdlib.h>

#include <string.h>

using namespace h\_list;

typedef struct list{

char atom;

struct list\* next;

}list;

void push(list\*\* tail\_ref, list\*\* head, char new\_data)

{

list\* new\_node = (list\*) malloc(sizeof(list));

new\_node->atom = new\_data;

new\_node->next = NULL;

if ((\*tail\_ref)==NULL){

(\*tail\_ref) = new\_node;

(\*head) = (\*tail\_ref);

return;

}

(\*tail\_ref)->next = new\_node;

(\*tail\_ref)= (\*tail\_ref)->next;

}

int count\_atoms(lisp L, list\*\* head, list\*\* tl, int count, int\* offsetindex){

count = 0;

if (isNull(L))

return 0;

printf("\nProcessing: ");

write\_lisp(L);

if(isAtom(L)){

count++;

push(tl, head, getAtom(L));

printf("\n%\*s'%c'!\n", (\*offsetindex)\*4, "Found atom: ", (\*tl)->atom);

(\*offsetindex)++;

return count;

}

printf("\n%\*s\n",(\*offsetindex)\*4,"Didn't find an atom");

(\*offsetindex)++;

count+=count\_atoms(L->node.pair.hd, head, tl, count, offsetindex)+count\_atoms(L->node.pair.tl, head, tl, count, offsetindex);

return count;

}

int main(){

lisp L;

read\_lisp(L);

if(isNull(L)){

printf("Empty list.\n");

return 0;

}

int count = 0, offsetindex=0;

list\* l = NULL;

list\* head = l;

count = count\_atoms(L, &head, &l, count, &offsetindex);

char s[3]="s ";

if (count==1) {

s[0]=' '; s[1]='\0';

}

printf("\n%d atom%sfound:\n",count, s);

int n = 1;

while (head!=NULL){

printf("%d. atom %c\n", n, head->atom);

head=head->next;

n++;

}

destroy(L);

putchar('\n');

return 0;

}