**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МОЭВМ**

отчет

**по лабораторной работе №1**

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

Тема: Рекурсии

|  |  |  |
| --- | --- | --- |
| Студент гр. 6382 |  | Вайгачёв А.О. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2016

**Цель работы.**

Oзнакомиться с основными понятиями и приёмами рекурсивного программирования, получить навыки программирования рекурсивных процедур и функций.

**Задание № 21.**

Построить синтаксический анализатор для понятия скобки.

скобки::=квадратные | круглые

квадратные:: = [ [ квадратные ] ( круглые ) ] | B

круглые::=( ( круглые ) [ квадратные ] ) | А

**Основные теоретические положения.**

**Рекурсивные определения и вычисления**

Рассмотрим пример, который присутствует, по-видимому, во всех учебниках по программированию. Функция *факториал* натурального аргумента *n* обозначается как *n*! и определяется соотношением

*n*!=123...(*n* – 1)*n* . (2.1)

Удобно доопределить 0!=1 и считать, что *n* – целое неотрицательное число.

Некоторым недостатком определения (2.1) является наличие в нём многоточия «...», передающего речевой оборот «и так далее» и имеющего интуитивно понятный читателю смысл. Можно дать точное, так называемое *рекурсивное* определение функции *n*!, лишенное этого недостатка, т. е. не апеллирующее к нашей интуиции. *Определим:*

а) 0! = 1 , (2.2)

б) *n*! *=* (*n-1*)!*n* при *n* >0.

Соотношения (2.2) можно рассматривать как свойства ранее определенной функции, а можно (как в данном случае) использовать их для определения этой функции.

Далее для функции *n*! будем использовать привычное «функциональное» (префиксное) обозначение fact (*n*) , указывая имя функции и за ним в скобках аргумент. Тогда (2.2) можно записать в виде

*если n = 0;*

*fact (n 1) n, если n> 0;*

или в другой форме записи

*fact (n)* ***if*** *n = 0* ***then*** *1* ***else*** *fact (n 1) n,* (2.4)

где использовано условное выражение **if** *b* **then** *e*1 **else** *e*2, означающее, что в том месте, где оно записано, следует читать *e*1, если выполняется условие *b*, и следует читать *e*2, если условие *b* не выполняется.

Функция, определяемая таким образом, *единственна.* Действительно, пусть есть две функции, например: fact1 (*n*) и fact2 (*n*), удовлетворяющие соотношениям (2.2) или их эквивалентам (2.3), (2.4). Рассмотрим разность dfact (*n*) = fact1 (*n*) fact2 (*n*). Очевидно, что, во-первых, в силу соотношения «а» из (2.2) имеем dfact (0) = 0, а, во-вторых, для функции dfact (*n*) также справедливо соотношение «б». Действительно,

dfact (*n*) = fact1 (*n*) fact2 (*n*) = fact1 (*n-*1) *n* fact2 (*n-*1) *n* =

= (fact1 (*n* -1) fact2 (*n -* 1)) *n* = dfact (*n* -1) *n*.

По индукции легко доказывается, что из соотношений dfact (0) = 0 и dfact (*n*) = dfact (*n*  1)  *n* следует, что dfact (*n*) = 0 для любого *n*> 0*.*

Как конструктивно могут быть использованы эти определения (как они «работают»)? Например, вычислим fact(4):

fact(4) = (fact(3)4) = ((fact(2)3)4) = (((fact(1)2)3)4) = ((((fact(0)1)2)3)4) =

= ((((11)2)3)4) = (((12)3)4) = ((23)4) = (64) = 24.

Здесь каждое новое использование рекурсивного определения заключается в скобки, а затем, когда ссылки на новые значения функции исчерпаны, последовательно каждое произведение двух сомножителей, заключенное в скобки, заменяется на результат умножения. Очевидно, что такое рекурсивное определение может рассматриваться как рецепт вычисления функции.

Приведём примеры других известных функций, которые также могут быть определены рекурсивно. *Наибольший общий делитель (****g****reatest* ***c****ommon* ***d****iviser)* натуральных *a* и *b*:

gcd(*a*,*b*) **if** *a* = *b* **then** *a* **else** **if** *a* >*b* **then** gcd(*a b*,*b*) **else** gcd(*a*,*b* *a*),

или при *a* > *b* 0

gcd(*a*,*b*) **if** *b* = 0 **then** *a* **else** gcd(*b*,*a* **mod** *b*).

*Степенная функция f*(*a,n*)*=an* (основание степени *a* например, вещественное число, а показатель степени *n* целое неотрицательное число):

*f*(*a,n*) **if** *n* = 0 **then** 1 **else** *f*(*a,n* 1) *a ,* (2.5)

или другой вариант

*f*(*a,n*) **if** *n* = 0 **then** 1 **else** (*f* 2(*a, n* **div** 2))*f*(*a, n* **mod** 2) . (2.6)

Eщё один вариант можно получить из (2.6) с учётом того, что *n* **mod** 2 может принимать только значения 0 или 1, а *f*(*a*,0) = 1 и *f*(*a,*1) = *a*. Таким образом получаем

*f*(*a,n*) **if** *n* =0 **then** 1 **else**

**if** Even(*n*) **then** (*f* 2(*a, n* **div** 2)) **else** (*f* 2(*a, n* **div** 2)) *a,* (2.7)

где Even(*n*) = **not** Odd(*n*). Заметим, что соотношения (2.5) и (2.7) определяют одну и ту же функцию, но вычисления значения функции при заданном аргументе, порождаемые этими соотношениями, различны.

(Источник: учебное пособие)

**Рекурсивные функции и процедуры**

Рекурсивное *определение* можно превратить в *описание* рекурсивной процедуры или функции. Например:

**function** fact(n: Nat0): Nat;

**begin**

**if** n=0 **then** fact:=1 **else** fact:=fact(n–1)\*n

**end** {fact}

В нашем примере в теле функции fact в операторе присваивания **fact:=fact(n–1)\*n** находится вызов этой же функции, а именно **fact(n–1)**. Как происходит выполнение таких функций и процедур? Рассмотрим следующую наглядную *модель* исполнения рекурсивных алгоритмов вычислительной машиной на примере функции **fact**. Пусть в программе имеется вызов описанной ранее рекурсивной функции, например **z:=fact(k)** с фактическим параметром (аргументом) **k>0**. Будем считать, что этот вызов запускает первый (стартовый) *экземпляр* функции fact. Каждый очередной рекурсивный вызов этой же функции (в данном примере это **fact(k–1), fact(k–2)** и т. д.) приводит к запуску нового экземпляра функции. При этом предыдущий экземпляр *откладывается,* и так происходит до тех пор, пока не будет вызван последний (*терминальный*) экземпляр функции, т. е. экземпляр, не содержащий рекурсивного вызова (в данном примере fact(0)). Этот терминальный экземпляр порождает значение функции (**fact(0)=1**) и завершает свою работу. Затем *восстанавливается* предыдущий (отложенный последним) экземпляр. Он, в свою очередь, порождает новое значение функции и завершает работу. Затем восстанавливается предыдущий экземпляр и т. д., до тех пор, пока не будет восстановлен и завершен экземпляр, соответствующий стартовому запуску (и тем самым закончится процесс вычисления **fact(k)**). Очевидно, память машины, используемая для хранения отложенных экземпляров, должна быть устроена таким образом, чтобы обеспечить восстановление первым того экземпляра, который был отложен последним.

Такой способ организации и функционирования памяти известен как стек (Stack), или магазин (по аналогии с магазином огнестрельного оружия), или очередь типа LIFO, т. е. Last In First Out (последним пришёл – первым ушёл)

(Источник: Учебное пособие)

**Взаимно-рекурсивные функции и процедуры.**

Пусть требуется построить *синтаксический анализатор* понятия *скобки:*

*cкобки::=квадратные |* *круглые*

*квадратные::=*[*круглые круглые*] *| +*

*круглые::=*(*квадратные* *квадратные*) *|*

В этом рекурсивном определении последовательности символов, называемой *cкобки*, присутствуют две взаимно-рекурсивные части: *квадратные* определяются через *круглые*, и наоборот, *круглые* через *квадратные.* В простейшем случае *квадратные* есть символ «+», а *круглые* есть символ «»*.* Другие примеры последовательностей, порождаемых этим рекурсивным определением:

‘[– –]’, ‘(++)’, ‘[(++)([–(++)][– –])]’, ‘(+[(++)([–(++)][(+[– –])–])])’.

Синтаксическим анализатором назовём программу, которая определяет, является ли заданная (входная) последовательность символов *скобками* или нет. В случае ответа «нет» сообщается место и причина ошибки.

*Реализуем основную часть этой программы как булевскую функцию Bracket, которая вызывает две другие (парные) булевские функции Round и Square, определяющие, является ли текущая подпоследовательность частью круглые или квадратные соответственно. Каждая из функций Round и Square в свою очередь вызывает парную к себе (Square и Round соответственно).*

Пусть входная последовательность читается из файла F, а результат и вспомогательные сообщения выводятся в файл G. Оба эти файла будут глобальными для функций Bracket, Round и Square. Вспомогательные сообщения квалифицируют ошибки в записи последовательности *скобки* в том случае, когда результат функции Bracket есть False. Для формирования этих сообщений будет использована процедура Error.

Функции Round и Square реализованы так, что они читают очередной символ входной последовательности и далее действуют в прямом соответствии с рекурсивными определениями частей *круглые* и *квадратные* соответственно. При этом в функции Bracket приходится читать первый символ входной последовательности дважды. Можно было бы избежать этого, используя «заглядывание вперёд», однако такая реализация менее прозрачна.

**Program** SyntaxAnalysisOfBracket;

{Bracket = скобки, Round = кругл, Square = квадр }

{ скобки ::= квадр | кругл }

{ квадр ::= + | [кругл кругл] }

{ кругл ::= – | (квадр квадр) }

**var** F,G : Text;

b: Boolean;

**procedure** Error (k: Word);

**begin**

WriteLn(G);

**case** k **of**

1:{Bracket} WriteLn(G,'! - Лишние символы во входной строке.');

2:{Bracket} WriteLn(G,'! - Недопустимый начальный символ.');

3:{Square} WriteLn(G,'! - Отсутствует "]".');

4:{Square} WriteLn(G,'! - Отсутствует "+" или "[".');

5:{Square} WriteLn(G,'! - Очередной квадр - пуст.');

6:{Round} WriteLn(G,'! - Отсутствует ")".');

7:{Round} WriteLn(G,'! - Отсутствует "–" или "(".');

8:{Round} WriteLn(G,'! - Очередной кругл - пуст.');

**else** {?} WriteLn(G,'! - ...');

**end**{case};

**end**{Error};

**function** Round : Boolean; Forward;

**function** Square : Boolean;

{ квадр ::= + | [кругл кругл] }

**var** s: Char;

k: Boolean;

**begin** Square := False;

**if not** Eof(F) **then**

**begin** Read(F,s); Write(G,s);

**if** s='+' **then** Square := True

**else if** s='[' **then**

**begin**

{ квадр ::= [кругл кругл] }

k := Round;

**if** **k then** k:={k **and** } Round

**else** {первый кругл ошибочен};

**if** k **then** {оба кругл правильны}

**if not** Eof(F) **then**

**begin**

Read(F,s); Write(G,s);

**if** (s=']') **then** Square:=True **else** Error(3)

**end**

**else** {нет ]} Error(3)

**else** {хотя бы один кругл ошибочен};

**end** {конец анализа [кругл кругл]}

**else** { не + и не [ } Error(4)

**end**

**else** {квадр - пуст} Error(5)

**end** {Square};

**function** Round : Boolean;

{ кругл ::= – | (квадр квадр) }

**var** s: Char;

k: Boolean;

**begin** Round := False;

**if not** Eof(F) **then**

**begin** Read(F,s); Write(G,s);

**if** s='–' **then** Round := True

**else if** s='(' **then**

**begin** { кругл ::= (квадр квадр) }

k := Square;

**if** k **then** k:={k **and** }Square

**else** {первый квадр ошибочен};

**if** k **then** {оба квадр правильны}

**if not** Eof(F) **then**

**begin**

Read(F,s); Write(G,s);

**if** (s=')') **then** Round:=True **else** Error(6)

**end**

**else** {нет )} Error(6)

**else** {хотя бы один квадр ошибочен};

**end** {конец анализа (квадр квадр)}

**else** { не – и не ( } Error(7)

**end**

**else** {кругл – пуст} Error(8)

**end** {Round};

**function** Bracket : Boolean;

{ not Eof(F) }

**var** b: Boolean; c: Char;

**begin**

b:=False;

Read(F,c); Reset(F);

**if** (c='+') **or** (c='[') **then** b:=Square

**else if** (c='') **or** (c='(') **then** b:=Round

**else** {недопустимый начальный символ} Error(2);

Bracket := b **and** Eof(F);

**if** b **and not** Eof(F) **then** {лишние символы} Error(1);

**end** {Bracket};

**begin** { SyntaxAnalysisOfBracket }

Assign(F, 'Bracket.DAT'); Assign(G, 'BracketRes.DAT');

Reset (F); {Rewrite} Append(G);

WriteLn(G,'Анализатор скобок:');

**if not** Eof(F) **then**

**begin**

b := Bracket;

WriteLn(G);

**if** b **then** WriteLn(G,'ЭТО СКОБКИ!')

**else** WriteLn(G,'НЕТ, ЭТО НЕ СКОБКИ!');

**end**

**else** WriteLn(G,'Пусто!');

Close(G);

**end.**

Отметим взаимную симметричность текстов процедур Round и Square, что соответствует симметричности определения понятия *скобки* относительно частей *круглые* и *квадратные* соответственно.

(Источник: Учебное пособие)

***Требования и рекомендации* к выполнению задания:**

1. *Проанализировать полученное задание, выделив рекурсивно определяемые информационные объекты и (или) действия.* Задание можно выполнить, если создать две взаимно-рекурсивные функции (round & square). Эти функции необходимо для начала вызвать из функции brackets. Так как определение *квадратные* (квадратные:: = [ [ квадратные ] ( круглые ) ] | B) содержат определение и *квадратные,* и *круглые,* то тело функции round будет содержать вызов round и square (и наоборот для определения *круглые*)
2. *Разработать программу, использующую рекурсию*;( текст программы будет предоставлен в приложении А)
3. *Сопоставить рекурсивное решение с итеративным решением задачи*; Определим итеративное решение задачи. Пусть n — длина строки, данная на вход компилятору. В итеративном подходе i-ый символ должен проверятся с учетом окружения. К примеру возьмем кусок текста, который был дан на вход программе (“(( (( (( ((”). Можно предположить, что это начало для 4-ех определений *круглый.* В этом подходе нужно придумать сложную структуру из циклов и условий в циклах и условиях, в то время как это легко реализовать при помощи рекурсий. В этом случае программа вызовет рекурсию с глубиной 4 и при успешном выходе из нее будет обозначать, что данное выражение удовлетворяет заданию.
4. *Сделать вывод о целесообразности и эффективности рекурсивного решения данной задачи*; (В выводе).

**Решение задачи.**

*Описание работы программы и алгоритма*. Сперва для корректной работы программы мне нужно провести инициализацию переменных и объектов, с которыми мне предстоит работать ( функция init() ). После переходим в функцию brackets() , в которой идёт проверка текста на соответствие определению указанного в задании. Из нее мы переходим в рекурсию (round() & square()), где считываются символы и сравниваются исходя из предположений, что должно идти дальше. Эти рекурсивные алгоритмы имеют симметричное строение и разбиты на две части: первая часть («квадратные:: = [ [ квадратные ] ( круглые ) ]») состоит из if-ов, условия в которых заданны заранее или вычисляются в ходе вызова рекурсивной функции (может и той же самой), вторая проверяет условие квадратные:: = А/ круглые::=В;

*Описание функций и глобальных переменных;*

FILE\* f\_in = NULL; // глобальные переменные

FILE\* f\_out = NULL; //

void init(); // инициализация программы. Спрашивает пользователя откуда считывает данные. Использует глобальные переменные f\_in & f\_out;

int doublePrintf(const char \*s); // функция, которая выводит строку const char \*s в файл f\_out и на косоль.

int doublePrintf\_c(const char s); // схожая ф-я с int doublePrintf(const char \*s) только для символа.

int Error(int c); //Выводит печатает сообщение об ошибке и завершает работу программы

с =

1 - File not found.

2 - Not a right scobe.

3 - Expected '[' or ']'.

4 - Not a square scobe.

5 - Expected '(' or ')'.

6 - Not a round scobe.

Default - \*\*\*;

char GetNPost(); //возвращает код символа и одновремено печатает его

int brackets(); // функция, которая является синтаксическим анализатором

int square(); //взаимнореурсивная с ф-ей round(). Возвращает 1 если определено понятие «квадратные» и 0 если нет. Использует f\_in в качестве материала

int round(); //взаимнореурсивная с ф-ей square(). Возвращает 1 если определено понятие «круглые» и 0 если нет. Использует f\_in в качестве материала

void pretty\_show(); // процедура ,которая используется для форматированного вывода. Использует глобальную переменную deepth.

int deepth = 0; // ! - global var

int flag = 0; // ! - булевская переменная, которая принимает единицу если единственный символ это А или В в строке.

**Тестирование**

|  |  |  |
| --- | --- | --- |
| № | Входные данные из консоли и источников (f\_in.txt или f\_stdin.txt) | Выход (f\_out.txt) |
| 2 | A  asdsad  n  A | Do you want read from file? (y/n)  Enter y (yes) or n (no)  Enter y (yes) or n (no)  Analising...    A  It is scobe! |
| 3 | n  B | Do you want read from file? (y/n)  Analising...    B  It is scobe! |
| 4 | y | Do you want read from file? (y/n)  Analising...    A  It is scobe! |
| 5 | n  ((((( | Do you want read from file? (y/n)  Analising...    ((  ((  ((  <--  !- Not a right scobe. |
| 6 | N  ((A)[A]) | Do you want read from file? (y/n)  Analising...    ((  A  )[  A <--  !- Not a right scobe. |
| 7 | Y | Do you want read from file? (y/n)  Analising...  ((  ((  A  )[  B  ])  )[  [[  B  ](  A  )]  ])  It is scobe! |
| 8 | y | Do you want read from file? (y/n)  Analising...  ((  ((  A  )[  B  ])  )[  [[  B  ]x<--  !- Expected '(' or ')'. |

**Вывод:**

Oзнакомился с основными понятиями и приёмами рекурсивного программирования, получил навыки программирования рекурсивных процедур и функций.

**Приложение A**

#include <stdio.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

/\*

21. Построить синтаксический анализатор для понятия скобки.

скобки::=квадратные | круглые

квадратные:: = [ [ квадратные ] ( круглые ) ] | B

круглые::=( ( круглые ) [ квадратные ] ) | А

\*/

FILE\* f\_in = NULL; // global var

FILE\* f\_out = NULL; // global var

void init(); //

int doublePrintf(const char \*s);

int doublePrintf\_c(const char s);

int Error(int c);

char GetNPost();

int brackets();

int square(char s);

int round(char s);

void pretty\_look();

int deepth = 0; // ! - global var

int flag = 0;

void init()

{

f\_out = fopen("f\_out.txt","w");

if (f\_out == NULL)

{

exit(EXIT\_FAILURE);

}

char ans;

doublePrintf("Do you want read from file? (y/n)\n");

while(f\_in == NULL)

{

scanf("%s",&ans);

switch (ans) {

case 'y':

f\_in = fopen("f\_in.txt","r");

if(f\_in == NULL) {Error(1);}

break;

case 'n':

f\_in = fopen("f\_stdin.txt","w");

if(f\_in == NULL) {Error(1);}

char c;

scanf("%s",&c);

fprintf(f\_in,"%s",&c);

fclose(f\_in);

f\_in = fopen("f\_stdin.txt","r");

if(f\_in == NULL) {Error(1);}

break;

default:

doublePrintf("Enter y (yes) or n (no)\n");

break;

}

}

}

int doublePrintf(const char\* s)

{

printf("%s",s);

fprintf(f\_out,"%s",s);

return 0;

}

int doublePrintf\_c(const char s)

{

printf("%c",s);

fprintf(f\_out,"%c",s);

return 0;

}

int Error(int c)

{

switch (c) {

case 1:

doublePrintf("\n!- File not found.");

exit(EXIT\_FAILURE);

break; // while loading from file

case 2:

doublePrintf(" <--\n!- Not a right scobe.");

exit(EXIT\_FAILURE);

break;

case 3:

doublePrintf("<--\n!- Expected '[' or ']'.");

exit(EXIT\_FAILURE);

break;

case 4:

doublePrintf("<--\n!- Not a square scobe.");

exit(EXIT\_FAILURE);

break;

case 5:

doublePrintf("<--\n!- Expected '(' or ')'.");

exit(EXIT\_FAILURE);

break;

case 6:

doublePrintf("<--\n!- Not a round scobe.");

exit(EXIT\_FAILURE);

break;

default: doublePrintf("\n!- \*\*\*\*."); exit(EXIT\_FAILURE); break;

}

}

char GetNPost()

{

char s;

fread(&s,1,1,f\_in);

doublePrintf\_c(s);

return s;

}

int brackets();

int square(char s);

int round(char s);

void pretty\_look();

void pretty\_look(int option) //if option equals 1, that means entering recursy, else exit. uses deepth as global var;

{

if (option) {++deepth ;} else {--deepth;};

doublePrintf("\n");

for (int i = 0; i < deepth; ++i)

{

doublePrintf(" ");

}

}

int brackets() //returns 1 {скобки::=квадратные | круглые} else 0

{

doublePrintf("Analising...\n ");

//pretty\_look(1);

char s = GetNPost();

if (round(s) || square(s))

{

doublePrintf("\nIt is scobe!");

return 1;

}

else

{

Error(777);

};

}

int square(char s) //returns 1 {квадратные:: = [ [ квадратные ] ( круглые ) ] | B} else 0

{

if (s == 'B') { pretty\_look(0); return 1;}

if (s == 'A' && !flag) { pretty\_look(0); return 1;}

flag = 1;

if(s == '[')

{

s = GetNPost();

if( s == '[')

{

pretty\_look(1);

s = GetNPost();

if (square(s) == 1)

{

s = GetNPost();

if (s == ']')

{

s = GetNPost();

if (s == '(')

{

pretty\_look(1);

s = GetNPost();

if(round(s) == 1)

{

s = GetNPost();

if (s == ')')

{

s = GetNPost();

if (s == ']')

{

pretty\_look(0);

return 1;

} else {Error(3);};

} else {Error(5);};

} else {Error(6);};

} else {Error(5);};

} else {Error(3);};

} else {Error(4);};

} else {Error(3);};

}

return 0;

}

int round(char s) //returns 1 {круглые::=( ( круглые ) [ квадратные ] ) | А} else 0

{

if (s == 'A') { pretty\_look(0); return 1;}

if (s == 'B' && !flag) { pretty\_look(0); return 1;}

flag = 1;

if(s == '(')

{

s = GetNPost();

if( s == '(')

{

pretty\_look(1);

s = GetNPost();

if (round(s) == 1)

{

s = GetNPost();

if (s == ')')

{

s = GetNPost();

if (s == '[')

{

pretty\_look(1);

s = GetNPost();

if(square(s) == 1)

{

s = GetNPost();

if (s == ']')

{

s = GetNPost();

if (s == ')')

{

pretty\_look(0);

return 1;

} else {Error(5);};

} else {Error(3);};

} else {Error(4);};

} else {Error(3);};

} else {Error(5);};

} else {Error(6);};

} else {Error(5);};

}

return 0;

}

int main()

{

init();

brackets();

}