13

Синтактичен анализ и намиране на стойност на израз. Търсене с връщане назад

13.1 Синтактичен анализ и намиране на стойност на израз

В много случаи синтаксисът на различни езикови конструкции има рекурсивна структура. За формалното описание на такива конструкции се използва широкоразпространеният език на Бекус-Наур. Например цяло число без знак може да се опише по следните два начина:

```
<цяло_без_знак> ::=  <цифра>|
                     <цяло_без_знак><цифра>
```

или

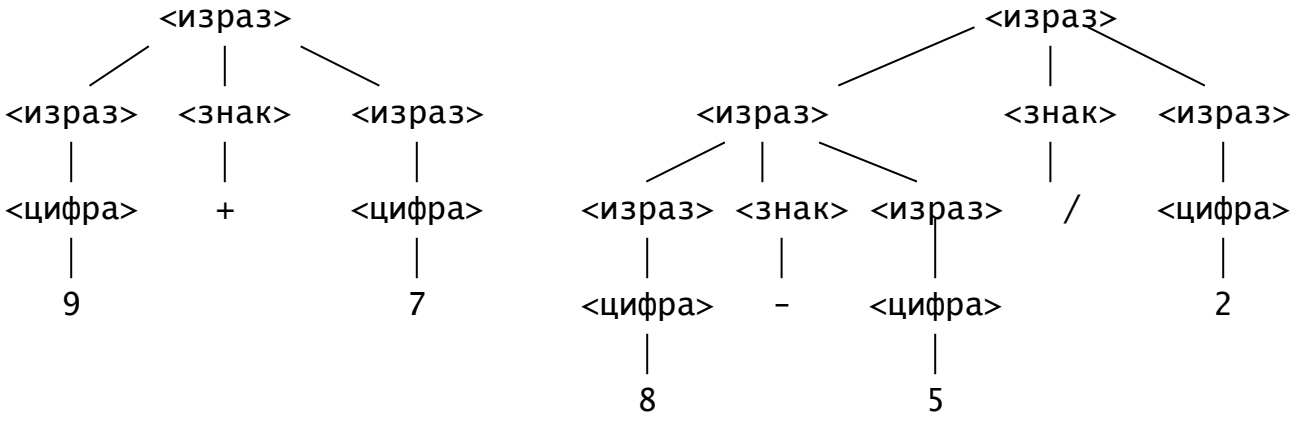
```
<цяло_без_знак> ::=  <цифра>|
                     <цифра><цяло_без_знак>
```

Да се направи синтактичен анализ на символен низ според някакви правила означава да се провери дали низът е получен според тези правила, да се определи видът на съставящите го части и връзките между тях. За целта се конструира т. нар. **дърво на синтактичния разбор.**

Например символните низове (9+7) и ((8-5)/2) са изрази според правилата:

```
<израз> ::= <цифра>|(<израз><знак><израз>)
<знак> ::= +|-|*|/;
<цифра> ::= 0|1|...|9
```

и съответните дървета на синтактичен разбор са:



Задачата на синтактичния анализ не се състои в рисуване на такива дървета, а в анализ на текста и ако е правилен, да се конструира някаква структура, която определя вида на съставящите низа части и връзките между тях. В горния случай следните структури определят дърветата на синтактичния разбор:

```
expr(sign(+), expr(digit(9)), expr(digit(7)))
expr(sign(/), expr(sign(-), expr(digit(8)), expr(digit(5))),
              expr(digit(2))).
```

В тази част няма да разгледаме в пълнота задачата за синтактичния анализ на изрази. Ще дискутираме само проверката дали даден символен низ е израз в смисъла на дадени правила. Тъй като дефинициите на тези правила обикновено са рекурсивни, най-естествените решения са рекурсивните.

Задача 111. Да се състави рекурсивна програма, която проверява дали низ, въведен от клавиатурата, *започва* с израз, определен от правилата:

```
<израз> ::= <цифра>|(<израз><знак><израз>)
<знак> ::= +|-|*|/;
<цифра> ::= 0|1|...|9.
```

Програма Zad111.cpp решава задачата. В нея са дефинирани булевите функции:

bool formula1(); - проверява дали въведеният низ започва с формула;

bool digit(char); - проверява дали символ е цифра;

bool sign(char); - проверява дали символ е знак, според указаното правило.

```
// Program Zad111.cpp
#include <iostream.h>
bool formula1();
bool digit(char);
bool sign(char);
int main()
{if (formula1()) cout << "yes \n";
 else cout << "no \n";
 return 0;
}
bool digit(char c)
{return c >= '0' && c <= '9';
}
bool sign(char c)
{return c == '+' || c == '-' || c == '*' || c == '/';
}
bool formula1()
{char c;
 cin >> c;
 if (c != '(') return digit(c);
 bool yes = formula1();
 if (!yes) return false;
 cin >> c;
 if (!sign(c)) return false;
 yes = formula1();
 cin >> c;
 return yes && c == ')';
}
```

Забележка: Ще отбележим още веднаж, че програмата отговаря положително, както за низа 8, така и за низовете 88 и 8a, както за низа (3+9), така и за низа (3+9)a-5.

Задачата за правилността на символен низ относно дадени правила ще решаваме по аналогичен начин, но чрез индекс ще следим до коя позиция е разпозната търсената форма и отговорът ще е положителен само ако низът е изчерпен.

Задача 112. Да се състави програма, която определя дали символен низ е израз в смисъла на следните правила:

```
<израз> ::= <израз>+<терм>|
            <израз>-<терм>|
            <терм>;
<терм> ::= <терм>*<цифра>|
           <терм>/<цифра>|
           <цифра>;
<цифра> ::= 0|1| ... |9.
```

Във функцията main се въвежда символен низ, в който се "изтриват" интервалите. Четенето на отделните символи от низа се осъществява чрез символната функция getchar:

```
char getchar()
{i++;
 if (i == len) return ' ';
 else return s[i];
}
```

която използва следните глобални променливи:

```
int i,           // индекс на текущия символ
    len;         // дължина на символния низ s
char s[100];     // символния низ, анализиран за израз
```

и връща сочения от индекса i символ, ако $0 \leq i < len$ и интервал, в противен случай.

Ще дефинираме следните рекурсивни функции:

bool expr() - реализира правилото

```
<израз> ::= <терм>|
            <израз>+<терм>|
            <израз>-<терм>;
```

bool term() - реализира правилото

```
<терм> ::= <цифра>|
```

```
            <терм>*<цифра>|
            <терм>/<цифра>;
```

Забелязваме, че тези дефиниции са ляво-рекурсивни, заради това че операциите +, -, * и / са лявоасоциативни. Дословното им реализаране ще доведе до зацикляне в неграничните случаи. Този проблем е известен като **проблем на лявата рекурсия**. Промяната на асоциативността на операциите, т.е.

```
<израз> ::= <терм>+<израз>|
            <терм>-<израз>|
            <терм>;
<терм> ::= <цифра>*<терм>|
           <цифра>/<терм>|
           <цифра>;
```

в случая решава проблема на лявата рекурсия. Програма Zad112.cpp дава едно решение на задачата.

```
// Program Zad112.cpp
#include <iostream.h>
#include <string.h>
char c;
int i, len;
char s[100];
char getchar()
{i++;
 if (i==len) return ' ';
 else return s[i];
}
bool expr();
bool term();
bool digit();
int main()
{cout << "Input an expression! ";
 char t[100];
 cin.getline(t, 100);
 len = strlen(t);
 i = -1;
 for (int j = 0; j <= len-1; j++)
```

```
   if (t[j] != ' ')
    {i++;
     s[i] = t[j];
    }
 len = i+1;
 i = -1;
 if (expr() && i+1 == len) cout << " yes \n";
 else cout << "no\n";
 return 0;
}
bool digit()
{c = getchar();
 return c >= '0' && c <= '9';
}
bool term()
{if (!digit()) return false;
 c = getchar();
 if (c != '*' && c != '/')
 {i--;
  return true;
 }
 return term();
}
bool expr()
{if (!term()) return false;
 c = getchar();
 if (c != '+' && c != '-')
 {i--;
  return true;
 }
 return expr();
}
```

Задача 113. Да се напише програма, която въвежда символен низ и ако той е правилен израз според правилата от предишната задача, пресмята стойността му.

В предишната задача проблема на лявата рекурсия решихме чрез промяна асоциативността на операциите. Това не оказа влияние на правилността на решението. Ако трябва обаче да се намери стойността на символен низ, представящ израз по новите правила, ще се получи грешен резултат (действията ще се извършват отдясно наляво).

В случая с лявата рекурсия ще се справим като запишем правилата по следния нерекурсивен начин:

<израз> ::= <терм>{<знак1><терм>}$_{опц}$

<терм> ::= <цифра>{<знак2><цифра>}$_{опц}$

<знак1> ::= +|-

<знак2> ::= *|/

Програма Zad113.cpp решава задачата.

```
// Program Zad113.cpp
#include <iostream.h>
#include <string.h>
char c;
int i, len;
char s[100];
char getchar()
{i++;
 if (i == len) return ' ';
 else return s[i];
}
bool expr(double&);
bool expr1(double, char, double&);
bool term(double&);
bool term1(double, char, double&);
bool digit(double& x)
{c = getchar();
 x = (int)c-48;
 return c >= '0' && c <= '9';
}
int main()
{cout << "Input an expression! " ;
 char t[100];
```

```
 cin.getline(t, 100);
 len = strlen(t);
 i = -1;
 for (int j = 0; j <= len-1; j++)
   if (t[j] != ' ')
    {i++;
     s[i] = t[j];
    }
 len = i+1;
 i = -1;
 double m;
 if (expr(m) && i+1 == len) cout << m << endl;
 else cout << "no \n";
 return 0;
}
bool term(double& res)
{if (!digit(res)) return false;
 c = getchar();
 if (c == ' '|| c == '+' || c == '-') {i--; return true;}
 if (c != '*' && c != '/') return false;
 return term1(res, c, res);
}
bool term1(double x, char ch, double& res)
{if (!digit(res)) return false;
 switch (ch)
 {case '*': res = x*res; break;
  case '/': res = x/res;
 }
 c = getchar();
 if (c == ' ' || c == '+' || c == '-'){i--; return true;}
 if (c != '*' && c != '/') return false;
 return term1(res, c, res);
}
bool expr(double& res)
{if (!term(res)) return false;
 c = getchar();
```

```
 if (c == ' ') {i--; return true;}
 if (c != '+' && c != '-') return false;
 return expr1(res, c, res);
}
bool expr1(double x, char ch, double& res)
{if (!term(res)) return false;
 switch (ch)
 {case '+': res = x + res; break;
  case '-': res = x - res;
 }
 c=getchar();
 if (c == ' '){i--; return true;}
 if (c != '+' && c != '-') return false;
 return expr1(res, c, res);
}
```

13. 2 Търсене с връщане назад

При редица задачи се поставят въпроси като “Колко начина за ... съществуват?” или “Има ли начин за ...? или възниква необходимостта от намиране на всички възможни решения, т.е. да се изчерпят всички възможни варианти за решаване на дадена задача. Широкоизползван общ метод за организация на такива търсения е **методът търсене с връщане назад** (backtracking). При него всяко от решенията (вариантите) се строи стъпка по стъпка. Частта от едно решение, построена до даден момент се нарича **частично решение**.

Методът се състои в следното:

- Конструира се едно частично решение.
- Проверява се дали частичното решение е общо (търсеното). Ако е така, решението се запомня или изважда и процесът на търсене или завършва, или продължава по същата схема, докато бъдат генерирани всички възможни решения.
- В противен случай се прави опит текущото частично решение да се продължи (според условието на задачата).

- Ако на някоя стъпка се окаже невъзможно ново разширяване, извършва се връщане назад към предишното частично решение и се прави нов опит то да се разшири (продължи) по друг, различен от предишния начин.

Търсенето с връщане назад се осъществява като се използва механизмът на рекурсията.

Ще го илюстрираме чрез няколко примера.

Задача 114. (Пътища в лабиринт) Квадратна мрежа има n реда и n стълба. Всяко квадратче е или проходимо, или е непроходимо. От проходимо квадратче може да се премине във всяко от четирите му съседни като се прекоси общата им страна. В непроходимо квадратче може да се влезе, но от него не може да се излезе. Да се напише програма, която намира всички пътища от квадратчето в най-горния ляв ъгъл до квадратчето в най-долния десен ъгъл. Пътищата (ако съществуват) да се маркират със '*'.

Мрежата ще представим чрез квадратна матрица M от символи. Квадратче (i, j) е запълнено със символа 1, ако е непроходимо и с 0 – ако е проходимо.

Програма Zad114.cpp решава задачата. В нея функцията init реализаира въвеждане на мрежата. При това се осигурява квадратче (0, 0) да е проходимо. Това не е ограничение, тъй като ако то е непроходимо, задачата няма смисъл. Функцията writelab извежда мрежата с маркирания със * път, ако съществува или съобщава, че път не съществува. Съществената част от програмата е функцията void labyrinth(int i, int j). Тя осъществява търсенето на всички пътища от квадратче (i, j) до квадратче (n-1, n-1). /Предполагаме, че сме намерили частично решение – път от квадратче (0, 0) до квадратче (i, j)/. В главната функция main обръщението към нея се осъществява с фактически параметри 0 и 0. В labyrinth е реализиран следният алгоритъм:

- Ако квадратче (i, j) съвпада с (n-1, n-1) е намерен един път от квадратче (i, j) до квадратче (n-1, n-1), извежда се и или се завършва изпълнението, или се продължава търсенето на други пътища;

- Ако горното не е вярно, а квадратче (i, j) е извън мрежата или е непроходимо, labyrinth завършва изпълнението си;
- Ако квадратче (i, j) е вътре в мрежата и е проходимо, частичното решение се продължава като квадратче (i, j) се включва в пътя /маркира се със символа */ и продължава търсенето на всички възможни пътища от някое от четирите ограждащи (i,j) чрез стена квадратчета до квадратче (n-1, n-1), т.е.

```
labyrinth(i+1, j);
labyrinth(i, j+1);
labyrinth(i-1, j);
labyrinth(i, j-1);
```

Завършването на изпълнението на тези рекурсивни функции означава, че са невъзможни нови разширения, т.е. не съществуват други пътища от квадратче (i, j) до квадратче (n-1, n-1). Осъществява се връщане назад към предишното частично решение, като се отказваме квадратче (i, j) да принадлежи на пътя чрез възстановяване проходимостта му.

```
// Program Zad114.cpp
#include <iostream.h>
char m[20][20];
int n;
bool way = false;

void init()
{int i, j;
 do
 {cout << "n= ";
  cin >> n;
 } while(n < 1 || n > 20);
 do
 {cout << "labyrinth:\n";
  for (i = 0; i <= n-1; i++)
  for (j = 0; j <= n-1; j++)
    cin >> m[i][j];
```

```
 } while (m[0][0] != '0');
}
void writelab()
{int k, l;
 cout << endl;
 for (k = 0; k <= n-1; k++)
  {for (l = 0; l <= n-1; l++)
     cout << m[k][l] << " ";
   cout << endl;
  }
}
void labyrinth(int i, int j)
{if (i == n-1 && j == n-1)
 {m[i][j] = '*';
  way = true;
  writelab();
 }
 else
  if (i >= 0 && i <= n-1 && j >= 0 && j <= n-1)
   if(m[i][j] == '0')
   {m[i][j] = '*';
    labyrinth(i+1, j);
    labyrinth(i, j+1);
    labyrinth(i-1, j);
    labyrinth(i, j-1);
    m[i][j] = '0';
   }
}
int main()
{init();
 labyrinth(0, 0);
 if (!way) cout << "no \n";
 return 0;
}
```

Задача 115. Квадратна мрежа има n реда и n стълба. Всяко квадратче е или проходимо, или е непроходимо. От проходимо квадратче може да се премине във всяко от четирите му съседни като се прекоси общата им страна. В непроходимо квадратче може да се влезе, но от него не може да се излезе. Да се напише програма, която намира най-кратък път от квадратчето в най-горния ляв ъгъл до квадратчето в най-долния десен ъгъл. Пътят (ако съществува) да се маркира със '*'.

Програма Zad115.cpp решава задачата. Чрез алгоритъм, аналогичен на този от предишната задача, се генерират всички възможни пътища от квадратче (0, 0) до квадратче (n-1, n-1). От тях се избира един с най-малка дължина (брой квадратчета, включени в него). Освен масива M, програмата поддържа и двумерен масив P, в който пази мрежата с текущия най-кратък път. Тези структури са описани като глобални променливи. Като глобални са дефинирани и n – размерност на мрежата, bmin – дължина на текущия минимален път, b – дължина на текущо конструирания път и way – булева променлива, индицираща съществуването на път от квадратче (0, 0) до квадратче (n-1, n-1).

```
// Program Zad115.cpp
#include <iostream.h>
char p[20][20];
char m[20][20];
int n, bmin = 0, b = 0;
bool way = false;
void init()
{int i, j;
 do
 {cout << "n= ";
  cin >> n;
 } while (n < 1 || n > 20);
 do
 {cout << "labyrinth: \n";
  for (i = 0; i <= n-1; i++)
```

```
  for (j = 0; j <= n-1; j++)
    cin >> m[i][j];
 } while (m[0][0] != '0');
}
void opt()
{int k, l;
 if (b < bmin || bmin == 0)
 {bmin = b;
  for (k = 0; k <= n-1; k++)
     for (l = 0; l <= n-1; l++)
        p[k][l] = m[k][l];
 }
}
void writelab()
{int k, l;
 if (!way) cout << "no way \n";
 else
 {cout << "yes \n";
  for (k = 0; k <= n-1; k++)
  {for (l = 0; l <= n-1; l++)
     cout << p[k][l] << " ";
   cout << endl;
  }
 }
}
void labyrinth(int i, int j)
{if (i == n-1 && j == n-1) {way = true; m[i][j]='*'; opt();}
 else
   if (i >= 0 && i <= n-1 && j >= 0 && j <= n-1)
   if (m[i][j] == '0')
   {m[i][j] = '*';
     b++;
     labyrinth(i+1, j);
     labyrinth(i, j+1);
     labyrinth(i-1, j);
     labyrinth(i, j-1);
```

```
      m[i][j] = '0';
      b--;
    }
  }
  int main()
  {init();
   labyrinth(0, 0);
   writelab();
   return 0;
  }
```

Задача 116. Дадени са n града (n е естествено число, $1 \le n \le 10$) и целочислена матрица Anxn, така че a_{ij} е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \le i, j \le n-1$). Да се напише програма, която намира и извежда всички ациклични пътища между два произволно зададени града в случай, че път между тях съществува.

Програма Zad116.cpp решава задачата. Ацикличен път е път, състоящ се от различни градове. Процедурата foundallway намира всички ациклични пътища от град i до град j в случай, че път съществува. Всеки път се записва в глобалния едномерен масив int x[100], а дължината му - в глобалната променлива s. Глобалната булева променлива way получава стойност true, ако съществува път между двата зададени града. Инициализирана е с false.

```
// Program Zad116.cpp
#include <iostream.h>
int arr[10][10] = {0};
int n, s = -1;
int x[100];
bool way = false;
void writeway()
{for (int i = 0; i <= s; i++)
  cout << x[i] << " ";
 cout << endl;
```

```
}
bool member(int x, int n, int* a)
{if (n == 1) return a[0] == x;
 return x == a[0] || member(x, n-1, a+1);
}
void foundallway(int i, int j)
{if (i == j)
 {way = true;
  s++;
  x[s] = i;
  writeway();
 }
 else
 {s++;
  x[s] = i;
  for (int k = 0; k <= n-1; k++)
    if (arr[i][k] == 1 && !member(k, s+1, x))
    {arr[i][k] = 0; arr[k][i] = 0;
     foundallway(k, j);
     arr[i][k] = 1; arr[k][i] = 1;
     s--;
    }
   }
}
int main()
{do
 {cout << "n= ";
  cin >> n;
 } while (n < 1 || n > 10);
 for (i = 0; i <= n-2; i++)
   for (int j = i+1; j <= n-1; j++)
   {cout << "connection between " <<  i << " and "
         << j << " 0/1? ";
    cin >> arr[i][j];
    arr[j][i] = arr[i][j];
   }
```

```
 int j;
 do
 {cout << "start and final towns: ";
  cin >> i >> j;
 } while (i < 0 || i > 9 || j < 0 || j > 9);
 foundallway(i, j);
 if (!way) cout << "no \n";
 return 0;
}
```

Задача 117. Дадени са n града (n е естествено число, $1 \le n \le 10$) и целочислена матрица Anxn, така че a_{ij} е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \le i, j \le n-1$). Да се напише програма, която намира и извежда ацикличен път с минимална дължина между два произволно зададени града в случай, че път съществува.

Използван е подход аналогичен на този в Задача 115.

```
// Program Zad117.cpp
#include <iostream.h>
int arr[10][10] = {0};
int n, s = -1, smin = 0;
int x[100], xmin[100];
bool way = false;
void writeway()
{if (!way) cout << "no way \n";
 else
 {cout << "yes \ n";
  for (int i = 0; i <= smin - 1; i++)
   cout << xmin[i] << " ";
  cout << endl;
 }
}
void opt()
{if (s+1 < smin || smin == 0)
```

```
 {smin = s + 1;
  for (int i = 0; i <= smin-1; i++)
    xmin[i] = x[i];
 }
}
bool member(int x, int n, int* a)
{if (n == 1 ) return a[0] == x;
 return x == a[0] || member(x, n-1, a+1);
}
void foundminway(int i, int j)
{if (i == j)
 {way = true;
  s++; x[s] = i;
  opt();
 }
 else
 {s++; x[s] = i;
  for (int k = 0; k <= n-1; k++)
    if (arr[i][k] == 1 && !member(k, s+1, x))
    {arr[i][k] = 0; arr[k][i] = 0;
     foundminway(k,j);
     s--;
     arr[i][k] = 1; arr[k][i] = 1;
    }
 }
}
int main()
{do
 {cout << "n= ";
  cin >> n;
 } while (n < 1 || n > 10);
 for (i = 0; i <= n-2; i++)
   for (int j = i+1; j <= n-1; j++)
   {cout << "connection between " <<  i << " and "
         << j << " 0/1? ";
    cin >> arr[i][j];
```

```
      arr[j][i] = arr[i][j];
     }
   int j;
   do
   {cout << "start and final towns: ";
    cin >> i >> j;
   } while (i < 0 || i > 9 || j < 0 || j > 9);
    foundminway(i, j);
    writeway();
    return 0;
  }
```

Задачи

Задача 1. Да се направи програма, която анализира символен низ въведен от клавиатурата и определя дали той е израз в смисъла на следната граматика:

```
<израз> ::= <израз>+<терм>|
            <израз>-<терм>|
            <терм>;
<терм> ::= <терм>*<буква>|
           <терм>/<буква>|
           <буква>;
<буква> ::= a|b|c ... |z.
```

Задача 2. Да се напише програма, която проверява дали символен низ, въведен от клавиатурата е израз съгласно формулата:

```
<израз> ::= <цифра>|
            f(<израз>)|s(<израз>)|p(<израз>)
<цифра> ::= 0|1|...|9
```

и ако това е така, пресмята стойността на израза, ако f(x), s(x) и p(x) намират съответно x!, x+1 и x-1.

Задача 3. Да се състави програма, която проверява дали низ е израз, определен от формулите:

```
<израз> ::= <цифра>|(<израз><знак><израз>)
<знак> ::= +|-|*|/;
```

<цифра> ::= 0|1|...|9

и ако това е така, намира стойността му.

Задача 4. Да се напише програма, която проверява дали символен низ, въведен от клавиатурата е израз съгласно формулата:

<израз> ::= <цифра>|
pow(<израз>,<израз>)|gcd(<израз>,<израз>)

<цифра> ::= 0|1|...|9

и ако това е така, пресмята стойността на израза, ако pow(x, y) и gcd(x, y) намират съответно x на степен y и най-големия общ делител на x и y (Ако някое от числата x или y е 0, за gcd(x,y) да се приеме другото или 0, ако и x, и y са 0).

Задача 5. Квадратна мрежа има n реда и n стълба. Всяко квадратче е или проходимо, или е непроходимо. От проходимо квадратче може да се премине във всяко от четирите му съседни като се прекоси общата им страна. В непроходимо квадратче може да се влезе, но от него не може да се излезе. Да се напише програма, която намира най-краткия път от квадратче (0, 0) до квадратче (n-1, n-1), който минава през произволно зададено проходимо квадратче на мрежата. Пътят да се маркира със '*' (ако съществува).

Задача 6. Дадени са n града (n е естествено число, $1 \leq n \leq 10$) и целочислена матрица Anxn, така че a_{ij} е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \leq i, j \leq n-1$). Да се напише програма, която намира и извежда всички ациклични пътища с указана дължина между два произволно зададени града в случай.

Задача 7. Дадени са n града (n е естествено число, $1 \leq n \leq 10$) и целочислена матрица Anxn, така че a_{ij} е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \leq i, j \leq n-1$). Ацикличен път, в който началния и крайния град съвпадат, се нарича цикъл. Да се напише програма, която намира и извежда всички цикли за произволно зададен град в случай, че цикъл съществува.

Задача 8. Дадени са n града (n е естествено число, $1 \leq n \leq 10$) и целочислена матрица Anxn, така че a_{ij} е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \leq i, j \leq n-1$). Ацикличен път, в който минава през всички върхове, ще наричаме Хамилтонов цикъл. Да се напише програма, която намира и извежда всички Хамилтонови цикли за зададените градове.

Допълнителна литература

1. Рейнгольд З., Нивергальт Н. Део, Комбинаторные алгоритмы. Теория и практика, Москва, Мир, 1980.

2. Узерелл Ч., Этюды для программистов, Москва, Мир, 1982.

3. Тодорова М., Езици за функционално и логическо програмиране. Логическо програмиране, София, СОФТЕХ, 1998.