

# Интерпретатор на логически изрази

## Условие

Да се направи програма, позволяваща работа с логически изрази. Програмата трябва да има конзолен интерфейс, като потребителят трябва да може да изпълни някоя от следните команди: **DEFINE**, **SOLVE**, **ALL** и **FIND**.

### 1. Дефиниране на логически функции

Командата **DEFINE** предоставя възможност на потребителя да дефинира логическа функция със зададено име. Логическите функции могат да са съставени от базовите логически операции **AND**, **OR** и **NOT** или други вече въведени от потребителя функции. Въвеждането на логическа функция трябва да проверява за наличие на грешки и може да бъде в [инфикс](#) или [постфикс](#) (обратен полски запис), според предпочитанията на студента.

Примери (в инфикс):

**DEFINE func1(a, b): "a & b"** //дефинира функция **func1** с операнди **a** и **b**, извършваща логическата операция **AND** между тях;

**DEFINE func2(a, b, c): "func1(a, b) | c"** //дефинира функция **func2**, извършваща логическата операция определена от **func1** между операндите **a** и **b**, и логическа операция **OR** между резултата от **func1** и операнда **c**. Функцията **func1** трябва да е била дефинирана от потребителя преди дефинирането на функция **func2**.

**DEFINE func3(a, b, c, d): "a & (b | c) & !d"**

**DEFINE func4(a, b, c): "a & b | c | !d"** //грешка операндът **d** не е дефиниран

**DEFINE func5(a, b, c, d): "func1(a, b) & func2(a, b, c) & func3(d)"** //грешка функцията **func3** се нуждае от четири параметъра, а е използвана само с един.

При изпълнение на команда **DEFINE**, студентът трябва да разчете въведения от потребителя текст и да обработи отделните му части:

- **DEFINE** - наименование на командата;
- **funcX(a, b, c, d)** - наименование на дефинираната функция и поредност на аргументите ѝ;

<sup>2</sup> L5 и U5 разглеждат реализацията на хештаблица, чрез масив и свързани списъци (отворено хеширане).

При изпълнение на команда **SOLVE**, студентът трябва да разчете въведения от потребителя текст и да обработи отделните му части:

- **SOLVE** - наименование на командата;
- **funcX(1, 0, 1, 0)** - наименование на дефинираната функция и стойности на аргументите;

Разчитането на командата трябва да се извърши символ по символ, като НЕ е разрешено ползването на готови функции като String.Split, String.IndexOf, Regex, LINQ и др. Ако студентът иска да ползва някоя от тях, трябва да разпише нейната функционалност сам.

От прочетения израз студентът трябва да намери запазената в хеш таблицата дефинирана функция, отговаряща на името **funcX**. От намерения запис се вижда, че поредността на аргументите е (**a, b, c, d**), а от командата **SOLVE** се вижда, че стойностите за тях са (**1, 0, 1, 0**).

Всеки от възлите **a, b, c** и **d** трябва да бъде намерен в дървото и за него трябва да се зададе съответната стойност<sup>3</sup>.

Трябва да се реализира рекурсивен метод, решаващ дървото, след изпълнението на който резултатът от изпълнението на целия израз трябва да се намира в корена на дървото<sup>4</sup>.

При повторно изпълнение на команда **SOLVE** за функция, която вече е решавана, да се преизчисляват само частите от израза, за които това е необходимо. Например, ако се изчислява резултатът за функцията **func2(1, 0, 0)** и след това за **func2(1, 0, 1)**, това трябва да доведе до преизчисление само на операцията **OR**, тъй като **func1** вече е била изчислена за тези аргументи при първото решаване на **func2**.

### 3. Изготвяне на таблица на истинност за логическа функция

Командата **ALL** дава възможност на потребителя да реши някоя от дефинираните функции за всички възможни стойности на аргументите.

Пример:

```
ALL func1 -> a , b : func1
           0 , 0 : 0
           0 , 1 : 0
```

---

<sup>3</sup> Търсене в дърво (BFS, DFS) се разглеждат в Л4 и У6.

<sup>4</sup> Рекурсивният метод може да се изпълни, като модификация на методите за търсене.

1, 0 : 0  
1, 1 : 1

Разчитането на командата трябва да се извърши символ по символ, като НЕ е разрешено ползването на готови функции като `String.Split`, `String.IndexOf`, `Regex`, `LINQ` и др. Ако студентът иска да ползва някоя от тях, трябва да разпише нейната функционалност сам.

От прочетения израз студентът трябва да намери запазената в хеш таблицата дефинирана функция, отговаряща на името **funcX**. От намерения запис се вижда, че поредността на аргументите е (**a**, **b**, **c**, **d**). Трябва да се генерират всички възможни вариации<sup>5</sup> на четирите аргумента и да се използва реализираният в т.3 метод за решаване на функция за всяка от вариациите.

#### 4. Реализация в поток

Хеш таблицата и изградените дървета да се реализират във файлов поток<sup>6</sup> (може да са в отделни файлове), така че изпълнението на програмата да може да се прекъсне във всеки един момент, без това да води до загуба на информация. За реализация на хеш таблицата е допустимо указване на максимален размер.

#### 5. Намиране на логическа функция (незадължителна)

Командата **FIND** дава възможност на потребителя да намери логическа функция, която не е налична, по въведена таблица на истинност. Логическата функция може да е съставена от всички базови или дефинирани от потребителя функции.

Пример:

```
FIND 0,0,0:0;  
      0,0,1:0;  
      0,1,0:0;  
      0,1,1:0;  
      1,0,0:0;  
      1,0,1:0;  
      1,1,0:0;  
      1,1,1:1
```

или чрез файл, в който последната колона е за резултата, а предходните са за операндите.

---

<sup>5</sup> Лексикографското генериране на вариации е разгледано в У2.

<sup>6</sup> Реализацията на масив и свързан списък в поток е разгледана в У3 и У4. Реализацията на дърво в поток е сходна на реализацията на свързан списък.

Пример:

**FIND** "d:\table.csv"

Result: "a & b & c" или "func1(a, b) & c"

За реализацията на това условие може да бъде предложен подход от студента. Някои от възможните варианти са:

1. Търсене чрез пълно изчерпване<sup>7</sup>. Програмата трябва да генерира функции (дървета) с нужния брой аргументи (в примера 4). Ако допуснем, че потребителят е дефинирал само една функция (func1), тогава някои от възможните функции са:
  - a & b & c & d;
  - a & b & c | d;
  - ...
  - func1(a, b) & c & d;
  - ...
  - func1(func1(a, b), func1(c, d));
  - ...
2. Търсене, чрез евристичен метод. Възможна е евристична реализация, например чрез генетичен алгоритъм<sup>8</sup>. За целта трябва да се реализират операции по кръстосване на дървета, което е подход на [генетичното програмиране](#).

Очевидно е, че няма горна граница за броя използвани функции, което налага търсенето да продължи до намиране на съвпадение или до изтичане на максимално допустимо време. Акцент в случая са бързината и ефективността на операциите, като може да се използват техники като branch & bound<sup>9</sup>.

---

<sup>7</sup> Търсене с пълно изчерпване е разгледано в У2.

<sup>8</sup> Евристични алгоритми се разглеждат в Л9 и У10.

<sup>9</sup> Методът се разглежда в Л8, У8 и У9.