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.

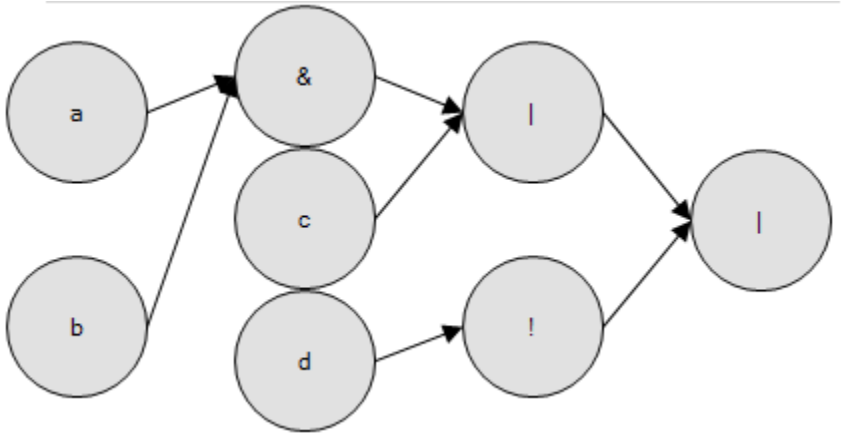
При изпълнение на команда **DEFINE**, студентът трябва да разчете въведения от потребителя текст и да обработи отделните му части:

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

- **((a & b) | c) | !d** - логически израз (тяло на функцията).

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

От прочетения израз студентът трябва да изгради дърво[1], в което всеки възел е операнд или оператор:



Пример за дърво, отговарящо на израза: **((a & b) | c) | !d**

Коренът на дървото, заедно с наименованието на функцията (в случая **funcX**), трябва да се запазят в хеш таблица[2], в която ключ е името на функцията. Всякакви структури различни от масив и динамичен масив (в това число хеш таблица, стек, опашка дърво и тн.) трябва да се реализират от студента.

2. Решаване на функция за дадени параметри

Командата **SOLVE** дава възможност на потребителя да реши някоя от дефинираните функции за определени стойности на аргументите.

Примери:

SOLVE func1(1, 0) -> Result: 0;

SOLVE func2(1, 0, 1) -> Result: 1;

[1] Л4 разглежда структурата стек и дърво. У6 разглежда работа с дървета. Изграждане на дърво от обратен полски запис може да бъде постигнато, чрез употребата на стек.
[2] Л5 и У5 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[3].

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[4].

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

[3] Търсене в дърво (BFS, DFS) се разглеждат в Л4 и У6.
[4] 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[5] на четирите аргумента и да се използва реализираният в т.3 метод за решаване на функция за всяка от вариациите.

5. Намиране на логическа функция

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.

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:

[5] Лексикографското генериране на вариации е разгледано в У2.

1. Търсене чрез пълно изчерпване[6]. 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));
   - …

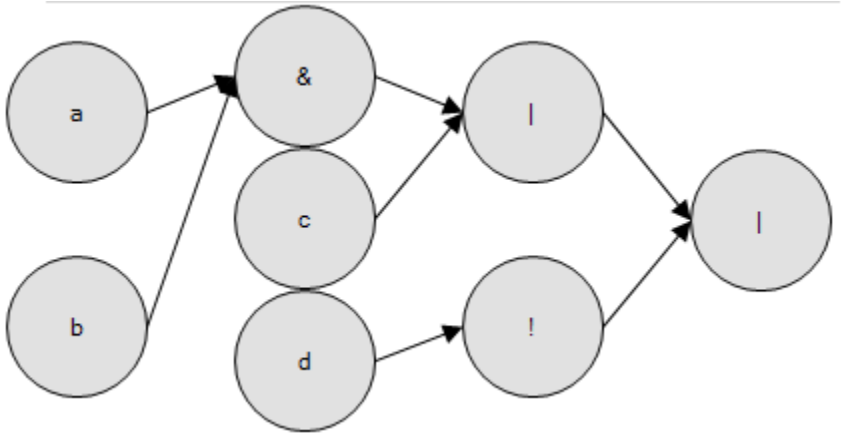
2. Търсене, чрез евристичен метод. Възможна е евристична реализация, например чрез генетичен алгоритъм[7]. За целта трябва да се реализират операции по кръстосване на дървета, което е подход на генетичното програмиране.

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[8].

6*. Визуализация на функция.

Визуализация на функция като дървовидна структура чрез графичен интерфейс (GDI), с възможност за показване на изчислената за всеки възел стойност.

Пример:



Фиг. 1. Визуализация на функцията като дървовидна структура.

[6] Търсене с пълно изчерпване е разгледано в У2.
[7] Евристични алгоритми се разглеждат в Л9 и У10.
[8] Методът се разглежда в Л8, У8 и У9.

Реализация и точки

Всички функции за обработка на текст трябва да се реализират от студента (не е разрешено използването на функциите string.Split, string.IndexOf, Regex, функциите на LINQ и т.н.). Всички помощни структури и типове трябва да се реализират от студента, в това число стекове, свързани списъци, хеш таблици, дървета и т.н.

Студентът трябва да реализира програмата в следната задължителна последователност:

1. Въвеждане на логически функции.
   Допустимо е изразите да се въвеждат чрез обратен полски запис или както са показани в примерите.

   Макс. брой точки за реализация: 20;

2. Запис и четене от файл.
   Функциите за запис и четене трябва да записват информацията в текстов (четим) вид.

   Макс. брой точки за реализация: 5;

3. Решаване на функция за дадени параметри.
   Студентът трябва да реализира разнасяне на изчислените стойности от листата на дървото (операндите на израза), към корена (крайният резултат). Изчислението трябва да бъде рекурсивно.

   Макс. брой точки за реализация: 10;

4. Изготвяне на таблица на истинност за логическа функция.
   Не е разрешено използването на готови функции за генериране на вариации за операндите.

   Макс. брой точки за реализация: 10;

5. Намиране на логическа функция.
   Ако търсенето се реализира чрез пълно изчерпване, трябва да се реализират всички възможни вариации на дърво с един възел, с два възела и тн, стига дърветата да са с коректен брой операнди.

   Макс. брой точки за реализация: 15;

6. * - Тази точка е незадължителна. При реализиране на всички точки, вкл. незадължителната, студентът ще бъде освободен от изпит с отлична оценка.