# **Верификатор за прости свойства**

# **Абстракт**

В този проект осъществяваме верификацията на прости свойства за цели числа, числови масиви, низове и булеви променливи.

# **1 Въведение**

## 1.1 Предмет

Проекта разглежда прости свойства на типовете: цяло число, низ, масив от цели числа, булеви променливи. Низовете се бележат с единични кавички „' '“, а масивите с фигурни скоби „{}“ и запетаи между отделните елементи. Празният символ се игнорира освен ако не е част от низ.

Пример: „'а а'“ различно от „'аа'“, но „а = {5, 6}“ е еднакво с „а = {5,6}“.

## 1.2 Цел

Целта на проекта е да се верифицират прости свойства за различните горепосочени типове:

<, >, =, ==, <=, >=, !=, &&, ||, !, като всеки символ, включен в тези операции е запазен символ и не може да се въвеждат такива променливи. Валидни са всякакви входове, използващи неколкократно сравнение и еднократно присвояване (а = b = 3 не се поддържа), докато булевите операции могат да бъдат използвани многократно, но само между булеви изрази, като още нещо специфично при булевите променливи е, че не могат да се използват операции за сравнение при тях (false < true е невалиден вход). За всички други типове са валидни операциите за сравнение, както и операции +, \*. Възможно е използването на кръгли скоби „()“ (стандартна употреба).

## 1.3 Дейности

Сравняване на изрази по всички възможни начини за сравнение (<, >, <=, >=, ==, !=). За булеви променливи само „==“ и „!=“.

*Числа:*стандартни операции за сравнение (само естествени числа и 0); изваждането е възможно само ако не се получава отрицателно число

*Масиви от числа:* лексикографско сравнение

*Низове:*лексикографско сравнение

*Булеви променливи:*стандартни операции за сравнение

Операциите, които трябва да се поддържат от отделните типове:

*Числа***:** събиране, изваждане, умножение и деление на число с друго такова или с масив от числа, като действията между число и масив връщат масив и не може от число да се изважда масив или да се дели на масив; също така е възможно извършването на умножение между число и низ и делението на низ на число, но деление число на низ не е възможно

*Низ***:** конкатенация с друг низ (бележи се със знак +), изваждане на низове (знак -; пример ‘abc’ – ‘c’ = ‘ab’); умножение с число и деление на число (пример ‘abcabc’ / 2 = ‘abc’)

*Булеви променливи***:** логически операции „*и*“ и „*или*“ с други булеви променливи, константи и изрази

*Масив от числа***:** събиране и умножение на масив от числа с друг такъв или с число, като операциите между число и масив представляват събиране, изваждане, умножение или деление на всеки елемент от масива с даденото число; събирането на два масива става поелементно – първи с първи елемент, втори с втори; може да се извършва събиране и между масиви с различен брой елементи (пример {1,2,3} + {2,3} = {3,5,3})

# **2** **Методи и материали**

За поддържането на множество сравнения се използва двоично дърво, като корена и вътрешните върхове представляват операнди (<, >, =, ==, <=, >=, !=, &&, ||, !), а листата са прости изрази, които не съдържат горепосочените операнди. Този вид представяне на изразите е удобен, тъй като почти всички операции са бинарни с изключение на отрицанието(„!“), което е унарна операция. Когато „!“ е част от въведеното твърдение, знакът остава в някое от листата и се прилага върху съответния елемент. Също така корена и всеки вътрешен връх имат или точно две поддървета или нула. Едно твърдение се счита за грешно, ако дървото, което е построено от него, се състои само от листо, с изключение на случаите, в които листото е булева променлива или константа. Представянето с двоично дърво улеснява сравняването на типовете за всяка операция. Ако лявото и дясното поддърво имат несъответстващи си типове или несъотвестващи на корена типове, то тогава израза е невалиден. Входът се „разбива“ по приоритет на операциите. Могат да се използват (), за променяне на приоритета.

(В низходящ ред: "=", "||", "&&", "!=", "==", "<=", "<", ">=", ">").

Например вход: true && 5 == 6, първо се разбива по „&&“, а след това по „==“ и дървото придобива вида

&&

true ==

5 6

Така този вход е валиден и се оценява до „FALSE“, докато израза – (true && 5) == 6, е невалиден.

Вход от вида (((7 <= 8) && true) == true) променя приоритите на && и ==. Дървото има вида

==

&& true

<= true

7 8

За изчислението на самите изрази в листата се използва обратен полски запис. В обратния полски запис операторите следват своите операнди. Например, за да съберем 3 и 4, ще напишем "3 4 +", вместо "3 + 4". Ако има множествени операции, операторът е даден веднага след втория операнд, така че ако изразът е написан "3 - 4 + 5" в обратен полски запис ще бъде написан по следния начин "3 4 - 5 +". Предимството на обратния полски запис е това, че премахва нуждата от скоби, които се изискват в инфиксната нотация. Докато "3 - 4 \* 5" също може да бъде написано "3 - (4 \* 5)", което означава нещо много по-различно от "(3 - 4) \* 5. Предходният израз също може да бъде написан "3 4 5 \* -", което недвусмислено означава "3 (4 5 \*) -", което след това се изчислява като "3 20 -".

При сравнителни тестове на обратния полски запис с алгебричен запис се установява, че обратният полски запис води до по-бързи изчисления поради две причини. Изчисленията на обратния полски запис нямат нужда от изрази, за да бъдат параметризирани, по-малко операции трябва да бъдат вкарани, за да се извършат обичайните изчисления. Като допълнение, при изчисленията с обратен полски запис се правят по-малко грешки, отколкото при другите типове на изчисления. По-късно проучване показва, че увеличената скорост от обратния полски запис може да е сведена до по-малкия брой на натискане на клавишите необходим да се въведе тази нотация, отколкото по-малкия познавателен товар на неговите потребители. Въпреки това, много доказателства показват, че обратният полски запис е по-труден за научаване от потребителите, отколкото алгебричната анотация.

Алгоритъмът за изчисляване на резултат на израз, записан с обратен полски запис, е сравнително лесен. За целта се използва структурата от данни Стек (*Stack*).

* За всеки пореден символ се извършва една от следните операции:
  + Ако символът е число:
    - добавяме числото в стека
  + Ако символът е оператор (операторът може да включва и функции):
    - Знае се, че операторът използва *n* аргумента - за стандартните математически операции събиране, изваждане, умножение, деление и степенуване се използват два аргумента - *n = 2*
    - Ако в стека има по - малко от n числа:
      * *Грешка* – потребителят е въвел непълен израз - пример: "3 + "
    - Ако има достатъчно числа, изваждаме **n** на брой от стека
    - Извършваме операцията
    - Добавяме в стека резултата от операцията
* След като изпълним горните операции за всеки символ в записа:
  + Ако има останало само едно число в стека:
    - числото е резултатът от израза
  + Ако има повече от едно число
    - *Грешка* – потребителят е въвел непълен израз

# **3 Резултати и дискусии**

В проекта е реализирано преобразуването на израз в обратен полски запис, както и пресмятането му след това, за да се верифицира подаден от потребителя израз. Проектът поддържа желаните операции за сравнение, присвояване, събиране, умножение, конкатенация, логически операции. Поддържат се твърдения с множествени сравнения, като във всяко отделно сравнение може да участват числа, низове, числови масиви, булеви константи, както и промениливи, за които предварително са зададени стойности или не.

Поддържат се входове от вида: 2\*a + 4 > 3 \* a + 3 – 1 (не е винаги вярно).

При дефинициите и сравненията с променливи, чиито стойности са неизвестни, не се поддържат деление и скоби (освен за булеви операции).

За повече примери, могат да се разгледат тестовете в самия проект.

# **Външни библиотеки**

Използвана е библиотеката StringUtils (<https://commons.apache.org/proper/commons-lang/download_lang.cgi>, (commons-lang3-3.5-bin.zip))

# **Литература**

# [1] [FMI lecture – RPN](https://learn.fmi.uni-sofia.bg/mod/resource/view.php?id=46721)

# [2] [Wikipedia - RPN](https://bg.wikipedia.org/wiki/Обратен_полски_запис)