

1	ОСНОВНИ ПОНЯТИЯ В АЛГОРИТМИТЕ И ПРОГРАМИРАНЕТО	7
1.1	Кратка история на науката за алгоритмите	8
1.2	Алгоритъмът като решение на задача.....	10
1.2.1	Алгоритъмът, представен с аналогии.....	10
1.2.2	Свойства на алгоритъма.....	15
1.2.3	Пример. Алгоритъм на Евклид.....	17
1.3	Алгоритъмът – предназначен за изпълнение от машина	20
1.3.1	Пример за RAM – програма.....	22
1.3.2	Елементи на управление. Схема на управление на алгоритъм.	23
1.3.3	Примерна схема на управление на алгоритъма на Евклид.....	25
♦	Примерна задача за упражнение	26
1.4	Реализиране на алгоритъма като програма	26
1.4.1	Основни постановки	26
1.4.2	Езикът като средство за формулиране на среда и действия.....	28
1.4.3	Организиране на последователността на действията	30
1.4.4	Конструкти за управление	31
1.4.5	Езикови фрази, реализиращи конструкти за управление	36
♦	Примерна задача за упражнение	40
2	БАЗОВИ ПОЗНАНИЯ ПРИ ПРОГРАМИРАНЕ	41
2.1	Понятие за Тип. Стандартни типове.....	41
2.2	Целочислен тип. Операции с целочислени променливи.....	44
2.3	Примерна програма за алгоритъма на Евклид	46
♦	Примерни задачи за упражнение	49
2.4	Някои приложения на операциите в множеството на целите числа.....	49
2.5	Модел на множеството на целите числа в машината.....	52
2.6	Множество на реалните числа. Пример с НОМ на отсечки	54
2.7	Типът “плаваща запетая”. Преходи между числовите типове.....	55
2.8	Итерация	58
2.9	Логически тип. Логически операции	60
2.10	Предпазване на итеративен цикъл от зацикляне.....	63
♦	Примерна задача за упражнение	67
2.11	Типът “Плаваща запетая” и представянето на рационални числа.....	68
2.12	Типът “Печатен символ”	70
♦	Примерна задача за упражнение	74
2.13	Ординални (скаларни) типове.....	75
2.14	Натрупване на суми и произведения	79
♦	Примерни задачи за упражнение	83
3	БАЗОВИ АЛГОРИТМИ И ПРОГРАМИ.....	86
3.1	Структурно програмиране.....	86
3.1.1	Изграждане на алгоритъм по блокове	86
3.1.2	Тор – Down методика за изграждане на алгоритъм.....	89
♦	Примерна задача за упражнение	96
3.2	Процедури.....	97
3.2.1	Изграждане на процедура	97
3.2.2	Процедури и обмен на данни.....	100
♦	Примерна задача за упражнение	108
3.2.3	Функция. Основни понятия	110
3.2.4	Понятие за числен метод и числово решение	112
3.2.5	Метод на “сканиране на интервала”.....	114

3.2.6	Дихотомично решаване на уравнение.....	120
◆	Примерна задача за упражнение.....	124
3.3	Структуриране на данните.....	Error! Bookmark not defined.
3.3.1	Структурата “Запис”.....	Error! Bookmark not defined.
3.3.2	Структурата “Масив”.....	Error! Bookmark not defined.
3.3.3	Алгоритъм за намиране на максимален (минимален) елемент на масив.....	Error! Bookmark not defined.
◆	Примерни задачи – претърсване на масив и умножение на матрици.....	Error! Bookmark not defined.
3.4	Сортиране.....	Error! Bookmark not defined.
3.4.1	Сортиране в друг масив.....	Error! Bookmark not defined.
3.4.2	Сортиране по метода “пряка селекция”.....	Error! Bookmark not defined.
◆	Примерни задачи за упражнение.....	Error! Bookmark not defined.
3.4.3	Сортиране по метода на пряката размяна (на мехурчето).....	Error! Bookmark not defined.
3.4.4	Сортиране по метода на прякото вмъкване.....	Error! Bookmark not defined.
◆	Примерни задачи за упражнение.....	Error! Bookmark not defined.
3.5	Алгоритми за претърсване.....	Error! Bookmark not defined.
3.5.1	Претърсване при ненаредени данни.....	Error! Bookmark not defined.
3.5.2	Претърсване по дихотомичен принцип.....	Error! Bookmark not defined.
◆	Примерни задачи за упражнение.....	Error! Bookmark not defined.
4	ПРИМЕРИ.....	Error! Bookmark not defined.
4.1	Предговор от съставителя на примерите.....	Error! Bookmark not defined.
4.2	Общи характеристики на текста на програма на процедурен език от високо ниво.....	Error! Bookmark not defined.
4.3	Езикова реализация на конструкти за управление.....	Error! Bookmark not defined.
}	Error! Bookmark not defined.
4.4	Целочислени алгоритми с циклична структура.....	Error! Bookmark not defined.
4.5	Коректност на цикличното управление.....	Error! Bookmark not defined.
4.6	Скаларни типове данни.....	Error! Bookmark not defined.
4.7	Разработване на алгоритъм по подхода “отгоре-надолу”.....	Error! Bookmark not defined.
4.8	Процедури.....	Error! Bookmark not defined.
4.9	Функции.....	Error! Bookmark not defined.
4.10	Съставни типове данни.....	Error! Bookmark not defined.
4.11	Сортиране.....	Error! Bookmark not defined.
4.12	Претърсване.....	Error! Bookmark not defined.

Предговор

Настоящото издание е учебник, създаден по едноименния курс за студентите от базова програма “Информатика и телекомуникации” на Нов Български Университет. Това предопределя неговата задача да даде от една страна основни познания, необходими като първа стъпка в по-нататъшното обучение на бъдещите информатици, а от друга – достатъчен обем от основни знания и умения на бъдещите бакалаври по телекомуникации, които няма да изучават други курсове, свързани с алгоритмите.

При разработването на методическия подход на представения тук материал, първата цел беше той да бъде разбираем, общообразователен, леко да се запаметява и да създава стабилни представи за основни понятия в алгоритмите и програмирането. Поради това материалът е придружен с много аналогии, метафорични илюстрации, опорни схеми и т.н.

Втората цел беше да подпомогне обучаемите да създават сами програми за по-елементарни алгоритми, като им даде методическа опора и тренира у тях умението да извършват преход от задачата към алгоритъма и към съответстващ му програмен текст. Поради това изложението е придружено с много подробни обяснения както на методите и алгоритмите, така и на практически задачи “трениращи” алгоритмичното мислене и уменията за кодиране.

Третата цел беше знанията и уменията да бъдат универсализирани по отношение на използвания език за програмиране. Това наложи да се изобрети един подход за обобщено изобразяване на алгоритъма-програма посредством схема за управление, която позволява “пренасяне” в който и да е процедурен език за програмиране. Като опорни за примерите са избрани Pascal, като идеен основоположник на структурното програмиране, най-близък до естествения език и най-отчетлив на абстрактно равнище на организация на работата с паметта и данните, и C, който е основа за привикване към синтаксиса на най-често използвания език при обучение в обектно-ориентирано програмиране. Практически няма никакво значение за обучаемите кой от езиците ще изберат за упражненията в компютърните класове, дори се препоръчва да опитат и двата опорни езика.

Предполага се, че студентите са програмирали в средния курс на обучение и са запознати на начално равнище с базовия синтаксис на някой език за програмиране, както и със значението на термини от рода на “машинна памет”, “компилятор”, “бит”, “взаимно еднозначно съответствие”, “ред”, “вектор” и други базови термини от областта на информатиката и математиката.

Учебникът съставен от две функционални части както следва:

Първа част – основно изложение. Основните знания и приложените примери в нея са разделени на три дяла като материал за аудиторни занятия. Автор на тези дялове е Велина Славова.

Втора част – решени примери на задачите за самостоятелна работа, съдържа и задачи с методическо значение за умението за съставяне на алгоритми при тясно обвързване с примерите от теоретичната част. Приведени са и работещи

програмни текстове, с коментари, предназначени за упражнения с компютър и разширяващи познанията по програмиране. Този дял от учебника е разработен от Станислав Иванов.

Благодарности

Методическият подход тук се базира на теорията, разработена и експериментално доказана от професор Лоуранс Барсалоу (prof. Lawrence W. Barsalou), свързана с откриване на наличието на така нареченото “базово ниво” на концептите и на параметри, свързани с основни характеристики на вътрешното структуриране в човешката понятийна система. Авторът на това не съвсем типично изложение на учебен материал от областта на точните академични дисциплини, Велина Славова, изказва благодарност на Lawrence W. Barsalou, който се запозна с ръкописа на лекциите и насърчи настоятелно издаването на учебно помагало по записките. Авторите изказват благодарност на студента Добромир Динев, който набра първи вариант на текста по ръкописа.

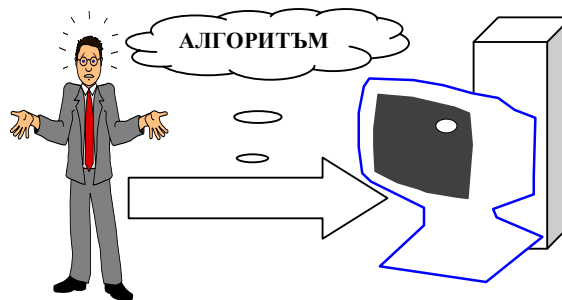
1 ОСНОВНИ ПОНЯТИЯ В АЛГОРИТМИТЕ И ПРОГРАМИРАНЕТО

➤ *Обща постановка*

Известно е, че съвременният компютър е машина, която изпълнява инструкциите, задавани от потребителя, както и това, че всичко, което тя извършва, се контролира и управлява от програми. Компютърът, в общ смисъл, е автомат, управляван от програми.

За да изясним принципно значението на алгоритмите за програмирането, да си представим, че един автомат трябва да бъде програмиран да работи така, че да изпълнява определена задача, или както е прието да се казва – да “*решава определена задача (определен проблем¹)*”. На илюстрацията по-долу алгоритъмът е показан образно като необходим преход, като свързващо звено между човека, програмиращ работата на машината, и самата машина – изпълнител. Така представеният алгоритъм има две страни, две негови “същности” – едната е отражение на мисленето на разумен човек, а другата е свързана с машината.

Самата задача и намирането на нейното решение характеризират алгоритъма в неговата “Абстрактна” същност. “Автоматната” страна на алгоритъма е свързана с възможностите на машината да изпълнява такива действия, че да имитира решаване на задачата така, както е според мислено намереното решение.



Най-общо, програмите са съвкупност от команди към машината. Те се създават с оглед решаване на конкретни, възникнали в практиката задачи и са съобразени с възможностите на изчислителната машина. Алгоритъмът е съществена концептуална съставка на програмата, той съдържа начина, по който да се изпълнят действия, водещи към решаване на задачата.

Съставянето на алгоритми, предназначени да управляват машини, изисква подход, при който са застъпени едновременно и двете “страни” на алгоритъма. Алгоритъмът, в качеството си на решение на определена задача, е абстрактна постройка, изразяваща разсъждения на разумен човек. Алгоритъмът като съвкупност от команди за действия, които могат да бъдат изпълнени от машината, е изказ на това решение, съобразен с възможностите на автомата. Предметът на нашите разсъждения тук е създаването на алгоритми, предназначени за изпълнение от компютър. В тази част от изложението ще се запознаем в някаква степен и с двете страни на алгоритъма, за да изградим обща представа за компютърен алгоритъм.

¹ Терминът “Проблем” се употребява в теорията на алгоритмите в смисъл на “задача за решаване” (гр. Problēma → лат. Problema - въпрос за разрешаване в дадена наука → англ. и мн. др. Problem – задача.)

1.1 Кратка история на науката за алгоритмите

В началото на това кратко изложение за науката за алгоритмите, да поясним какъв е произходът на думата *алгоритъм*. През 825 година Мохамед ибн Муса Абу Джафар Яфар Ал-Хорезми написал съчинения по математика, в които се описват правила за аритметични действия и постъпкови преобразования за извършване на пресмятания. Името на автора, видоизменяно през вековете и в зависимост от езика, на който е превеждано, е станало корен на думите: “Algorism” и “Algorithm”. В английски речник от 1957 год. например, думата “algorithm” не съществува, дадена е сродната ѝ дума: “algorism – правило за изпълнение на аритметични действия ...”.

През вековете представата за “алгоритъм” не се е изменяла съществено. Понятието “алгоритъм” се свързва и досега с понятия като: последователност от инструкции, план, предписание, рецепта. Терминът “алгоритъм” се използва за означаване на процедури, например класическите процедури за умножение, деление, диференциране, както и за стратегиите за печалби в игри или решаване на различни задачи-главоблъсканици. Такива процедури са били съставяни и описвани векове преди да бъдат наречени “алгоритми”. Пример за такава процедура е известният “алгоритъм на Евклид”, създаден в III в. пр. н. е., с който ще се запознаем още в тази тема.

Теорията на алгоритмите е дял на съвременната математика. Тя възниква в началото на 20 век, когато на базата на постановки от математическата логика са правени първите изследвания, целящи да се дефинира математически понятието *алгоритъм*. Тези изследвания са довели до развитието на редица съвременни клонове на математиката. Голяма част от научните разработки са свързани със създаване на формални системи, имащи за цел да бъде изведено математическо описание на една интуитивна идея, близка до представата за алгоритъм и наречена тогава “*ефективна процедура*”. Теоретичните изследвания на Тюринг, Пост, Чърч, Ербран, Гьодел, Клини, Кук и др. са довели до фундаментални теоретични изводи и постановки. Появил се един интуитивен “математически образ” на алгоритъма, наречен *ефективно изчислима функция*, т.е. функция, за която от зададена стойност на аргумента, с прилагане на *алгоритъм*, се намира съответната стойност на функцията. През първата половина на 20 век са били създадени и така наречените “машини” – абстрактни устройства, “работещи” по дефинирани правила за преобразуване на множество от аргументи в множество от функционални стойности. Такива абстрактни образи на “ефективно изчислими функции” са например известните “Машина на Тюринг” и “RAM Машина”. Независимо от това, че принципите на “работа” на абстрактните машини съществено се различават помежду си, се е оказало, че всички те “изчисляват” функции от един и същи математически клас. Това е класът на *частичните рекурсивни функции*, който се разглежда подробно в съответната математическа теория. От 1940 година насам, в теорията на алгоритмите този клас функции се изследва като “образ на алгоритъм” въобще. Най-общо, една функция f от този клас се нарича “рекурсивна”, защото се представя със зависимост, в която f участва в ролята и на функция, и на аргумент. Това позволява да се опише постъпков

изчислителен процес, в който функцията може да се прилага за “изчисление” над резултат, изчислен чрез самата нея в предходна стъпка.

Интуитивното заключение, което следва от това много кратко изложение, е, че алгоритмите имат някакви, присъщи за тях характерни особености, които могат да бъдат описани и изследвани с апарата на съвременната математика. Алгоритмите се оказват изразими посредством абстрактни обекти със собствено “поведение” и свойства, които не зависят от компютъра.

Същевременно, създадените абстрактни изчислителни машини могат да се представят и като средство за преработка на информация. Те създават концептуална основа за развитие на изследвания, свързани с процесите на обработка на информацията². Това развитие е тясно свързано с възникналата също през 20 век теория на автоматите, клон на Теорията на Системите за Управление. Основен обект на научно изследване в теорията на автоматите са математическите модели на “преобразователи” на дискретна информация, наричани “автомати”. Съществуващите до момента изчислителни машини са реализация на такива автомати.

Периодът от втората половина на 20 век до наши дни е свързан с интензивно реално използване на изчислителни машини. Това поражда въпроса за практическата изпълнимост на алгоритмите. Оказва се, че редица задачи, чиито решения са принципно изразими с алгоритми, не могат да бъдат решавани с приемливи реални изчислителни ресурси. Налага се изследване на връзката между алгоритъма като съществуващо абстрактно решение на дадена задача и изчислителните ресурси от памет и време, необходими за изпълнението му от машина. Така възниква един значим клон на съвременната теория на алгоритмите, занимаващ се с анализ на алгоритмите *по сложност*. Най-общо, той представлява математическо изследване и описание на свойствата на алгоритмите с оглед тяхното класифициране в йерархии от теоретично обосновани и изведени мерки за сложност. Теорията, свързана с изчислителната сложност на алгоритмите, се разработва интензивно от средата на 20 век и се базира на формален апарат, ползващ различни подобласти на съвременната математика. Получените значими теоретични резултати от изследванията на Пост, Блум, Стърнс, Кук, Колмогоров, Чейтин и др. имат силно влияние върху самите основи на математиката. Задачата тук е само да се създаде начална представа за понятието “сложност на алгоритъма” като понятие, обвързано с факта на изпълнение на алгоритъма като поредица от стъпки, извършвани от абстрактен изпълнител на решението на задачата. Съществено е да се подчертае, че “сложността” е свойство, присъщо на алгоритъма в качеството му на начин на решаване на дадена задача. Сложността на алгоритъма не зависи от машината, на която той ще се изпълнява, тя е негова собствена характеристика.

² Съвременната теория на информацията, основана на математически описания, се развива активно през 20 век (Шенон, Фишер и др.).

1.2 Алгоритъмът като решение на задача

1.2.1 Алгоритъмът, представен с аналогии

От изложеното в предходната точка може да се направи заключението, че кратка математическа дефиниция за алгоритъм няма. Нещо повече, това е понятие, което не може да се изясни с няколко изречения без да се ползват точни термини. Поради това ще създадем представа за алгоритъм, като разгледаме негови характерни особености. За целта ще се придържаме към аналогии, т.е. ще сравняваме алгоритъм с други обекти и явления, поведението и свойствата на които са известни и които ще наричаме “парадигми”³.

➤ *Алгоритъмът в парадигмата “Цел – План на действие”*

Тази парадигма често е използвана като опора за създаване на представа за алгоритъм. Тук също ще се позовем на нея, най-вече за да въведем допълнителни термини и понятия. Да вземем за пример някаква несложна практическа задача, която трябва да бъде изпълнена – разместване на предмети в помещението. Изпълнителят на такава задача, знаейки към какво крайно състояние на предметите се стреми, мислено извършва следното: 1. Запознава се със съществуващото разположение на предметите. 2. Съставя план на последователни действия, които да доведат съществуващото разположение на предметите до желаното.

Важно за съставянето на самия план на действия е да са известни началното и крайното положение на предметите. Алгоритъмът може да се оприличи на план на действие, за който са дефинирани както началното състояние, така и целта, т.е. крайното състояние.



Можем да представим алгоритъма като написан от разумен човек точен план за действия. Представен на хартия като текст с инструкции на естествен език, стрелки и т.н., планът е предназначен за изпълнител, който стриктно спазва написаните инструкции. Ще наричаме изпълнителя на алгоритъма “Агент” (лат. *agens* – “този, който действа”, от *agere* – “правя, извършвам”). Планът е съставен така, че Агентът да е в състояние да извърши изброените там действия. Ще наричаме “Среда” на алгоритъма обектите, над които се налага Агентът да извърши действия, за да бъде постигната целта.

³ парадигма (гр. Paradeigma → лат. Paradigma – пример, модел).

Често за пример на “алгоритъм” се посочват готварски рецепти. Целта на алгоритъма, представен вдясно като план на действие, е да се получи коктейл “Добро утро” от 100 гр. студено кафе и 20 гр. коняк.

Агентът на така представения алгоритъм не е машина, текстът на плана може да бъде разбран и изпълнен само от разумен човек.

Парадигмата “Цел – план на действие” представя алгоритъма като план за извършване на трансформация на нещо съществуващо в нещо, предварително дефинирано като цел. Между тези две крайни положения ... нещо се извършва.

АЛГОРИТЪМ

Коктейл «Добро Утро»

*Продукти : студено кафе
и коняк « Napoléon »*

1. Поставете 100 гр. студено кафе в чашата за коктейл.
2. Добавете 20 гр. коняк « Napoléon »
3. Разклатете добре.

*Получихте чудесен
утринен коктейл.*

➤ Алгоритъмът в парадигмата на Черната кутия

Съществува едно общо представяне на процесите, често използвано в абстрактните дисциплини, наричано “Модел на черната кутия”. Това представяне е удачно винаги, когато между началното състояние на някакъв процес и крайното му състояние “нещо се случва”.

Нека си представим, че алгоритъмът е еднопосочен процес на трансформиране на входа в изход. Представен в тази парадигма, алгоритъмът е активно въздействие. Не всяка активна трансформация, обаче, е алгоритъм. Съществува и една съществена допълнителна особеност – веднъж започнал, алгоритъмът трябва да завършва. За да бъде съставен “алгоритъм”, е необходимо да се намери такава трансформация на входа, която да завършва в изход при всички обстоятелства, при всички допустими входове.

Да подчертаем, че за разлика от обикновения план на действие, алгоритъмът не може да бъде променян по време на изпълнението му. Принципно, алгоритмите за управление на машина не се променят по време на изпълнение.

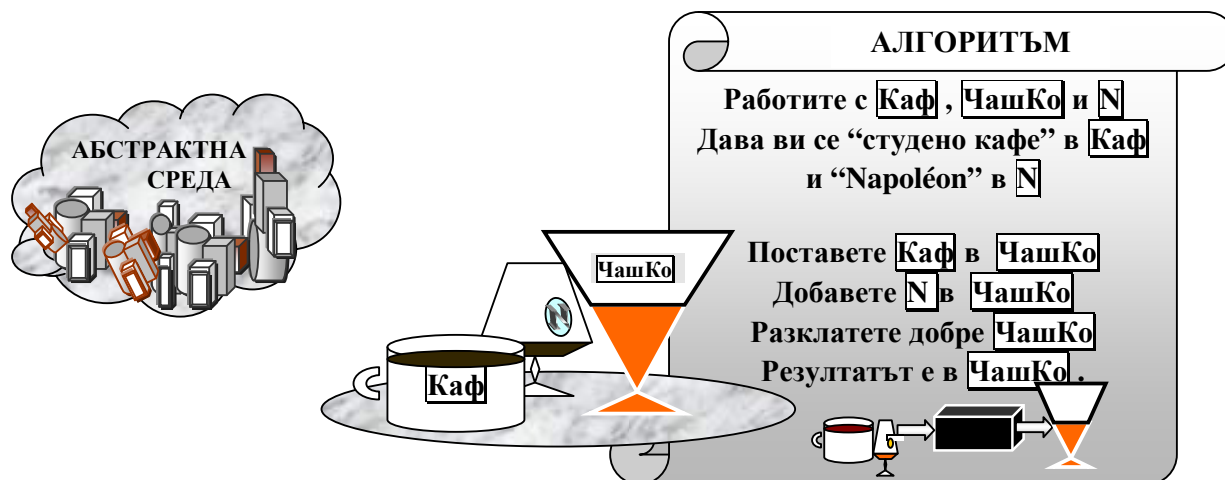


В представянето му като “Черна кутия”, алгоритъмът има точно един вход, един изход и това, което получи на входа, го преобразува в изход. Веднъж стартирал с някакъв допустим вход, алгоритъмът “няма право” да не завърши и не може да получи резултат, които не е дефиниран като допустим изход. Както е образно показано при изобразяването му като “Черна кутия”, алгоритъмът преобразува входа в изход, “работейки” в “затворено пространство”.

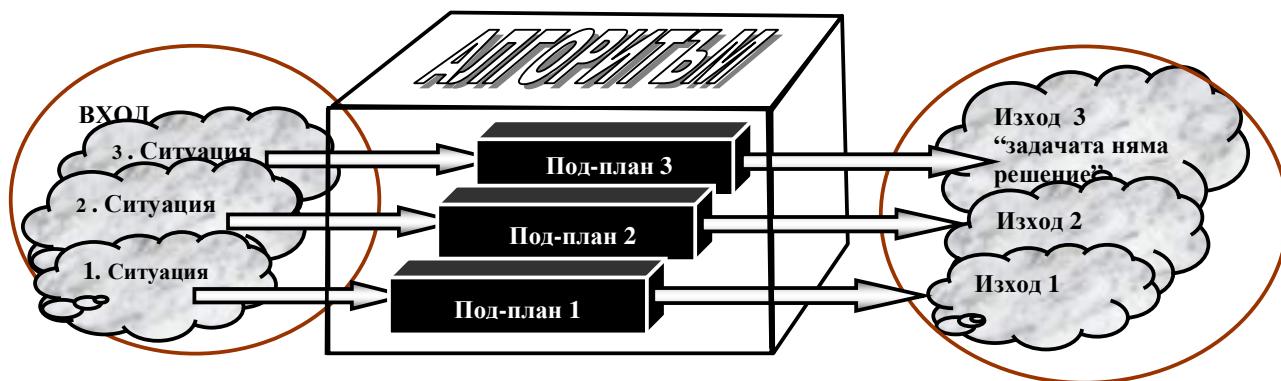
Средата на алгоритъма

Обвързахме алгоритъма от парадигмата “план на действие” с едно необходимо за работата на Агента обкръжение, което нарекохме “Среда”. Да предположим, че за да извърши действията от плана, Агентът трябва да “борави” с някакви елементи. Ще наричаме тези елементи “Среда” на алгоритъма. За съставянето на алгоритъм е необходимо да се определят елементите на Средата, с която Агентът ще работи. Тези елементи трябва да са му предоставени преди да започне да извършва посочените в плана действия..

Предполагаемият изпълнител на примерния алгоритъм за коктейла “Добро Утро” работи със « *студено кафе* » и « *Napoléon* », за да получи в резултат въпросния коктейл. Необходими са му например чаша за студеното кафе, чаша за коняка и чаша за самия коктейл. Това е неговата “Среда”. В тази аналогия се разграничава “съдържанието” от “чашата”. Нека Агентът разпознава чашите единствено по техните имена (етикети), а съдържанията имат за него само тип. (Например, той различава кафето от коняка и е “възпитан” да приема в кафената чаша « *студено кафе* » и нищо друго.) Един елемент от средата на такъв алгоритъм се представя с име и стойност.



Примерен текст на алгоритъм, предназначен за работа в дефинирана по този начин среда, е представен на фигурата горе. В този си вид текстът може да се оприличи на програма за робот. Да обърнем внимание, че при това представяне новополученият алгоритъм има по-общ смисъл на формално преобразуване на съдържанията на елементите. Роботът би получил като вход някакви “съдържания”, би извършил посочените в алгоритъма действия и би получил “нещо” в чашата за коктейл. Например, той би “разклатил добре” и празна коктейлна чаша, ако получи команда “старт”, а му се предоставят нулеви количества от « *студено кафе* » и « *Napoléon* ». Такова представяне налага да се предвидят възможните ситуации, в които би попаднал Агентът-робот и като се има предвид, че той спазва стриктно предписаните действия в зададената последователност, планът на работата му да се преобразува в много подробно предписание.



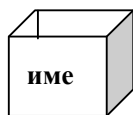
Особеността, илюстрирана на предходната фигура, се отнася за принципа, по който се извършва работата на Агент по зададен алгоритъм. Тъй като алгоритъмът не може да се променя по време на изпълнението, в него трябва да са описани с подходящи средства “под-планове” на работа в зависимост от “входната ситуация”. Образно казано, за да бъде оставен Агентът да “се справя сам”, трябва предварително да бъдат съобразени всички ситуации, в които той би попаднал, те да бъдат мислено обособени като “разрешен” вход и за всяка от тях да се състави съответен “под-план”. Агентът може да изпълнява единствено и само заложените в алгоритъма под-планове.

Най-общо, представянето на Средата посредством имена и съдържания, при това отделно от извършваните действия, се налага заради машината като краен изпълнител на алгоритъма. Въпреки това, абстракцията “Среда” е неразделна част от “мисления” алгоритъм – решение на задачата. Начинът на решаване на задачата е силно обвързан с избора на организирана Среда, в която Агентът да извършва действията си, за “да постигне целта”. Следователно, този избор влияе пряко на вида на решението – алгоритъм. Различните възможности, начини и средства за организиране на Средата както на абстрактно, така и на машинно равнище, в последните 50 години са предмет на специални теоретични усилия и приложни разработки. В резултат, до момента са развити и се ползват активно разнообразни типове “Среди”: среди – машинни образи на множества, среди със заложена вътрешна структура, среди, които “растат и намаляват по обем”, среди, свързани неразделно с действията в тях и различни други среди с наистина абстрактни свойства.

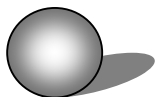
Понятието “Среда” е тясно обвързано с машинната памет. Поради това, че не може да се прави никакво сравнение между работата със съдържимото в машинната памет и възможностите за обработване на информация от човешкия мозък, за да избегнем евентуални обърквания, ще разглеждаме съставните елементи на понятието “Среда” в помощната парадигма “Кутия – Съдържание”.

Средата на алгоритъм, изразена в парадигмата “Кутия – съдържание”

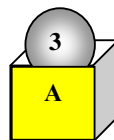
Променливите са елементи на Средата на алгоритъм – мислено решение на задача. Математическата представа за “променлива” е различна от понятието “променлива” от Среда на алгоритъм. В контекста на компютърен алгоритъм, “променлива” означава елемент от паметта, чието съдържание може да се променя. На следващите схеми са представени променливи – елементи на Средата, предназначени за работата на Агент-машина. Имената на променливите, представени като кутии, са отделени от техните стойности, които от своя страна са изобразени като “съдържания”.



Променлива
(кутия)



стойност
(съдържание)



Стойността (съдържанието) на
променливата с име А е 3

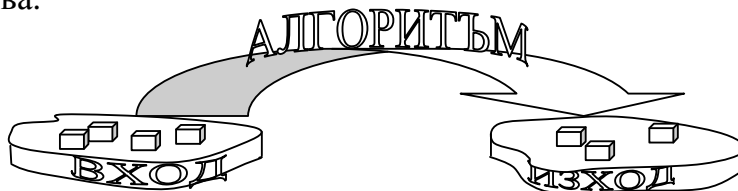
В изложението, което следва, ще се придържаме към тази “машинна” парадигма за променлива. В таблицата долу е илюстрирано означението на някои действия с променливи.

Описание	Означение	Представяне	Резултат
Променливата с име “А” добива стойност “3”.	$A \leftarrow 3$		
Променливата с име “А” добива стойност, равна на стойността на променливата с име “В”.	$A \leftarrow B$		
Променливата с име “А” добива нова стойност, равна на старата, изменена с (+ “1”).	$A \leftarrow A + 1$		

➤ *Алгоритъмът и Алгоритмичната Задача*⁴

Алгоритъмът, представен в изложените дотук парадигми, съдържа характеристики, необходими за представянето му като решение на така наречената “Алгоритмична задача”.

Нека се търси решение на някаква задача, зададена по общоприетия начин с “Дадено” като начално състояние и “Търси се”, като цел на решаването ѝ. Да дефинираме две множества от стойности – “входно множество” I_{alg} и “изходно множество” O_{alg} , аналози съответно на “Дадено” и “Търси се”. Нека задачата, дефинирана в термините на “Дадено–Търси се”, е предварително решена и е известно в общ смисъл какви са възможните отговори. Известни са и ограниченията за входа и изхода, например какви са допустимите стойности в двете множества.



Алгоритъмът е такова крайно, дискретно (постъпково), детерминирано преобразование, което, приложено над произволен допустим набор от стойности на входното множество, довежда до получаването на единствен набор от допустими стойности на изходното множество. При това полученото на изхода представлява правилен “отговор” на решаваната задача.

Алгоритмичната задача е задача за намиране на алгоритъм.

Известна под наименованието “Algorithmic Problem”, тази задача се определя така: *Да се намери общ единен метод (алгоритъм), за решаване на неограничен брой еднотипни единични задачи.* Често тази задача е означавана с термина “масова задача”.

1.2.2 Свойства на алгоритъма

Разгледаните парадигми представят алгоритъма по различни начини: като решение на задача, като постъпково преобразование, като план, изпълняван в определена “Среда”, като преобразуване на стойности от едно множество в стойности на друго и т.н. Нека обединим характеристиките на алгоритъма от различните парадигми и приемем, че елементи на входното и изходно множества на алгоритъма са част от “Средата”. Елементите на Средата представихме образно като кутии с имена, които могат да съдържат различни стойности. Нека приемем, че Агентът работи само с елементи на Средата. Той получава стойности за елементите от входа, извършва действия с елементи от Средата, променя стойностите им и в резултат получава стойности на елементите от изхода. След това обединяване на парадигмите, нека да представим общите свойства на преобразованието, наречено “Алгоритъм” така:

⁴ **Algorithmic Problem** – това е термин. Думата “проблем” тук е използвана в смисъл на “задача за решаване”

Алгоритъмът е преобразуване със следните свойства:

Дискретност. Преобразуването на входа в изход става на стъпки. Всяка стъпка е отделно елементарно действие над елементи на Средата, водещо до промяна на стойностите им.

Детерминираност.

Елементите на Средата са дефинирани.

Последователността на изпълнение на стъпките е определена по единствен начин.

Всяка елементарна стъпка се извършва по точно определено правило, по единствен начин.

В резултат, задаването на конкретни стойности на входа води до получаване на единствен “изход”.

Крайност. Елементите на Средата са краен брой. Преобразованието на входа в изход завършва след краен брой стъпки.

Резултативност (ефективност). Всяка стъпка е изпълнима в момента, в който предписанието изисква това. Алгоритъмът е изпълним за обозримо време.

Времето за изпълнение на алгоритъма зависи от скоростта на изпълнение на една стъпка. Въпреки тази относителна неопределеност, съществуват ефективни алгоритми за решаване на широк кръг задачи и в същото време – съществуват неефективни (неизпълними за “добро” време) алгоритми за решаването на много задачи, някои от които изглеждат дори прости.

Остава да отворен въпросът за това кои задачи могат да се решават с алгоритми, или, иначе казано – кои са задачите, за които могат да се намерят постъпкови, дискретни, детерминирани, крайни преобразования *Algorithm*: $I_{\text{alg}} \rightarrow O_{\text{alg}}$. Както споменахме, някакъв “математически” отговор на този въпрос съществува. Той е формулиран в известния “Тезис на Чърч”. Тезисът изказва следното твърдение: “Класът функции, изчислими с помощта на алгоритъм (в широк интуитивен смисъл), съвпада с класа на частичните рекурсивни функции.” Тезисът на Чърч не е теорема и изказаното в него твърдение не е доказано, но то е валидно за всички известни до момента алгоритми.

Очевидно, в практиката на програмиста не се налага непрекъснато да се проверява дали задачата, за която трябва да се състави алгоритъм, може да се представи чрез рекурсивна функция. Въпреки това, съществуването на задачи, които не могат да се решат с прилагане на алгоритъм поставя ограничения в използването на машините, а опитните програмисти придобиват усет за това кога една задача представена така, че да е решима с алгоритъм.

Фактът, че не всяка задача се поддава на решаване чрез алгоритъм, води до разширение на определението за Алгоритмична Задача: задача за *съществуването* на алгоритъм с който се решават неограничен брой еднотипни задачи ... и за намирането на такъв, ако съществува.

1.2.3 Пример. Алгоритъм на Евклид

Много често даван пример за алгоритъм е една процедура за намиране на най-голям общ делител на две цели числа, описана от Евклид през III в. пр. н. е. Тази процедура е класически пример за алгоритъм и носи името “Алгоритъм на Евклид”, въпреки че Евклид не я е наричал така. Нека обърнем внимание на факта, че по времето на Евклид представите за цели числа са се ограничавали с положителните цели (естествените) числа, а нулата не е била известна като абстракция за означаване на “нищо”. Алгоритъмът на Евклид, в неговата оригинална древногръцка версия, е бил описан в геометрична форма. Векове по-късно свойството “делимост” на две цели числа, вградено в Евклидовата процедура, е станало основно понятие в теорията на числата. Ние ще се придържаме към съвременната интерпретация на алгоритъма на Евклид.

Алгоритъмът на Евклид представлява постъпкова процедура за последователни преобразования с числа, с която се решава следната *задача*:
Задача:

Дадени са: Две цели положителни числа A и B .	Търси се: Най-големият общ делител на A и B .
--	--

Казваме, че едно цяло число C *дели* друго цяло число A , ако C се съдържа в A цяло число пъти. (C е по-малкото от двете числа.)

Всеки две цели числа A и B , от които A е по-голямото, могат да се представят така: $A = (\text{цяло число пъти}) \cdot B + (\text{цял остатък})$, или още:

$$A = x \cdot B + r, \quad \text{където } x \text{ и } r \text{ са цели числа.}$$

Ако допълним това твърдение с ограничението “остатъкът r трябва да е по-малък от B ” и кажем, че в такъв случай горното представяне на числото A чрез x , B и r е *единствено*, преразказваме една основна теорема от теорията на числата. Горният израз е начин за представяне на “делене”, но в множеството на целите числа. Той дефинира операция, наречена “*целочислено делене*” (представяне на известното ни “делене”, но само посредством цели числа).

Нека сега пресъздадем процедурата на Евклид във вид на алгоритъм, базиран на горното представяне – връзка между две цели положителни числа A и B .

Стъпка	Коментар
1. $A = x \cdot B + r$ Целочислено делене на A с B .	“Нанася” се малкото число B в голямото – A колкото пъти това е възможно и се намира какъв е остатъкът r .
2. $r > 0$? Проверка на остатъка. Ако няма остатък, отговорът е B . Иначе, премини към стъпка 3.	Ако няма остатък, тогава B <i>дели</i> A и следователно B е търсеният най-голям общ делител на A и на себе си. Иначе,
3. $A \leftarrow B$ $B \leftarrow r$ Смяна на стойностите.	Извършва се смяна на “ролите” – делителят става делимо, а намереният остатък става делител.
4. Премини към изпълнение на стъпка 1.	Прилагат се същите стъпки, отначало, но за получените нови стойности на делителя B и делимото A .

Изобразен схематично в парадигмата на "Черната кутия", алгоритъмът на Евклид може да се представи така:



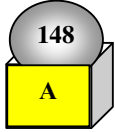
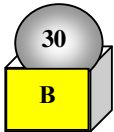
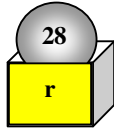
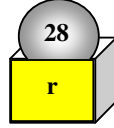
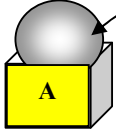
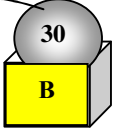
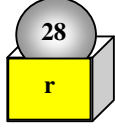
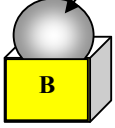
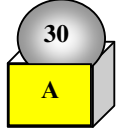
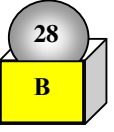
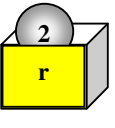
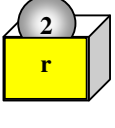
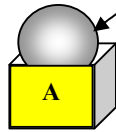
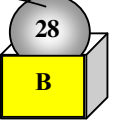
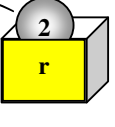
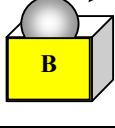
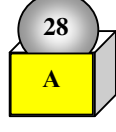
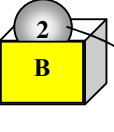


Средата на този алгоритъм съдържа променливите A и B като входни стойности, както и променлива r за остатък, който се изчислява при преминаването през стъпка 1. В края на изпълнението, отговорът се съдържа в променливата B .

Очевидно този алгоритъм е дискретно постъпково преобразование над стойностите от крайно входното множество, в крайно изходно множество. Преобразованието е детерминирано, защото са дефинирани както елементите на Средата, така и всички стъпки, които трябва да се извършват с тях. В резултат, прилагането му над две фиксирани естествени числа довежда винаги до получаване на един и същи отговор – естествено число. Стъпките включват извършване на елементарни, изпълними от човек действия. В такъв смисъл, процедурата на Евклид е ефективна. Дали тя завършва винаги? Интуитивно е ясно, че при изпълнението на тази процедура, стойностите, постъпващи съответно в ролята на делимо A и делител B от стъпка 1, намаляват. (Като се има предвид, че най-малкото естествено число е 1, това е и най-малката стойност, която делителят B би могъл да приеме по принцип. Но ако делителят B е “намалял” до $B=1$, той дели A с остатък 0). Можем да предполагаме, че независимо от входните стойности за A и B , тази процедура винаги ще завършва. Алгоритъмът на Евклид е краен и това се доказва математически, със съответен анализ на евклидовата процедура.

Алгоритъмът на Евклид е свързан с основни свойства на числата и е една от най-изследваните и прилагани през вековете изчислителни процедури. Простите числа, взаимно простите числа, числата от редицата на Фибоначи, както и ред други математически понятия са обвързани с Евклидовата процедура. Тя не е загубила значението си и в програми от нашето съвремие. Това се дължи и на използването на цели числа и на техните свойства в съвременните алгоритми, например тези, прилагани в крипто-системите. Подходът при оценяване на сложността на такива алгоритми преминава през оценката за сложност на алгоритъма на Евклид.

В приведената на следващата страница таблица е илюстриран стъпка по стъпка алгоритъмът на Евклид, приложен за един примерен вход от две естествени числа.

Начало – Вход – стойност за A : 148, стойност за B : 30

	A			B		r
1.		=	4.		+	
2.						
3.						
4.						
1.		=	1.		+	
2.						
3.						
4.						
1.		=	14.		+	
2.						

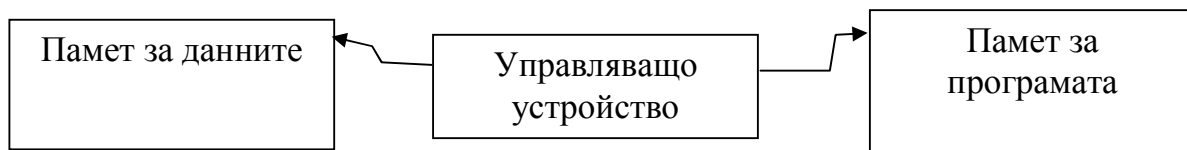
Край.

Отговор : 2.

1.3 Алгоритъмът – предназначен за изпълнение от машина

Съществуват множество абстрактни модели на изчислителни машини, които са послужили за теоретична основа при изследването на алгоритмите. Такива са например Машината на Тюринг и Машината с Произволен Достъп (Random Access Machine), наричана RAM-машина. В хода на тези изследвания е доказано, че ако даден алгоритъм може да бъде “изпълнен” при използване на един абстрактен модел на машина, той може (след подходящо преобразуване) да бъде изпълнен и с всеки друг от съществуващите модели на машина. Този факт е основание да разглеждаме алгоритъма, като изпълняван от машина въобще, независимо от това какъв именно формализъм е в основата на работата ѝ. Нашата задача е да въведем подходяща изразна форма за алгоритъма–решение на задача, която да е във вид, удобен с оглед на представянето му за изпълнение от машина по принцип.

Ще представим модела на Машината с Произволен Достъп, за да свържем въведените в предходната точка термини, обвързани с характерни особености на алгоритъма като абстракция, с предназначението му да бъде изпълнен от машина.



На горната схема е илюстрирана принципно организацията на абстрактна “RAM-машина”. Най-общо, едно устройство, наречено “управляващо”, при изпълнение на подадена му програма, променя стойностите на записани в паметта данни, като при това ги преобразува от някакво начално състояние – в крайно състояние. Управляващото устройство е “способно” да извършва точно дефиниран набор от елементарни действия над паметта за данни. Всяка стойност на “данна” се записва в единица от паметта за данни, наречена “регистър”. Регистрите имат имена (номера), по които са “разпознавани” от управляващото устройство. Програмата е крайна дискретна последователност $i_1, i_2, i_3, \dots, i_n$ от инструкции, без ограничение на техния брой. Програмата не може да бъде променяна по време на изпълнението. Управляващото устройство изпълнява инструкциите дискретно, по една инструкция на всяка стъпка. Инструкциите имат поредни номера, етикети, като етикетът на всяка инструкция е единствен за дадена програма. Кой номер инструкция да бъде изпълнявана на дадената стъпка се определя от *състоянието* на управляващото устройство. Номерата на състоянията $q_1, q_2, q_3, \dots, q_n$ съответстват на номерата на инструкциите. Преди началото на изчислението, управляващото устройство е в състояние q_1 и затова изпълнението на програмата започва от инструкция i_1 . Определени инструкции, наречени “*инструкции за преход*”, определят кое да е следващото състояние на управляващото устройство. Ако дадена изпълнявана инструкция не е инструкция за преход, след като я изпълни, устройството преминава на следващата инструкция от програмата. Изпълнението завършва с последната инструкция. Ще приемем, че програмата завършва със специална

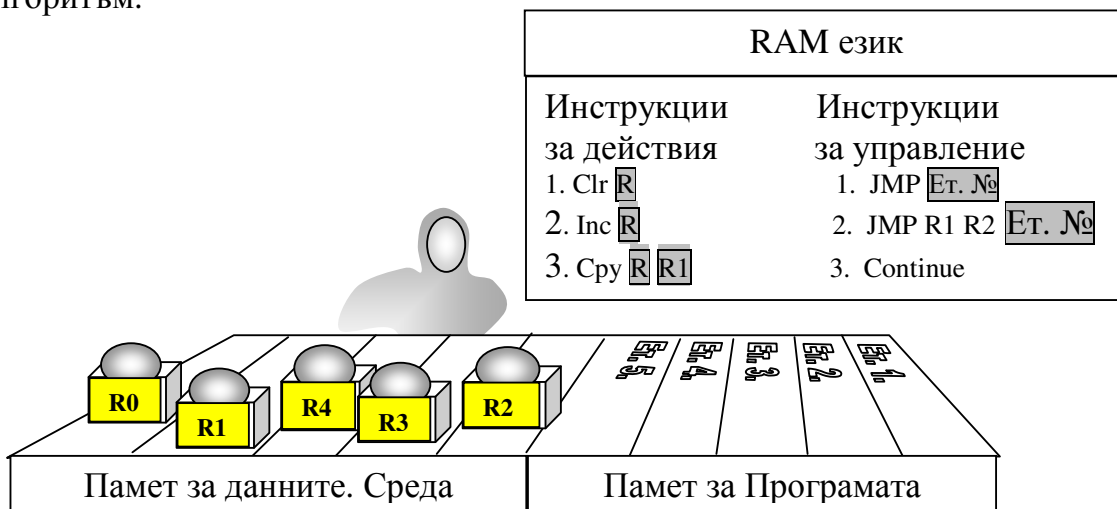
инструкция, чийто смисъл е “спри”, ако няма следваща инструкция, или “продължи”, ако има такава.

Ето примерен базов набор от инструкции, с които работи тази машина:

2. Нулирай съдържанието на регистъра “R”. (Clr R).
3. Добави единица към съдържанието на регистъра “R1”. (Inc R1).
4. Копирай съдържанието на регистъра “R2” в регистър “R3”. (Cpy R3 R2)
5. Премини към изпълнение на инструкцията с етикет “Ет №”. (*преход*) (Jmp №)
6. Ако съдържанието на регистър “R2” е равно на това на регистър “R3”, премини към изпълнение на инструкция с етикет “Ет №”. (*преход*) (Jmp R2 R3 №)
7. Край на изпълнението (ако инструкцията е последна в програмата) или “продължавай към следващата поред инструкция”. (Continue)

Очевидно на така описаното устройство може да бъде задавана програма, която не води до получаване на смислен резултат, както и на програма, която го “кара” да работи безкрайно дълго.

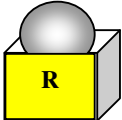

Нека обвържем това устройство с разгледания в предходната точка абстрактен алгоритъм, който, освен че винаги завършва, представлява начин за решаване на задача и следователно има смислен резултат. На илюстрацията долу е представена RAM-машина, като са спазвани означения от досега използваните парадигми. Нека приемем, че така “изглежда” изпълнителят на алгоритъм.



Агентът изпълнява алгоритъма, записан като програма, съставена с помощта на изброените горе инструкции. Средата е множество от именувани регистри, необходими на агента за изпълнение на дадената програма. Действия са *елементарните действия*, които агентът “знае” как да изпълнява в средата. На фигурата списъкът с действия над съдържанията на регистри е представен отделно. Всяко действие се извършва по единствен начин, регламентиран с точно правило. Агентът извършва действия само в дефинираната среда. Управлението представлява инструкции за промяна на последователността на извършваните над средата действия. На фигурата инструкциите за управление са в отделен списък. Управлението е “логическият скелет” на програмата. Програмата е съвкупност от инструкции за управление и инструкции за действия в средата.

1.3.1 Пример за RAM – програма

Ще се ограничим с използване на примерния RAM-език, за да съставим елементарен алгоритъм и програма. В следващата таблица са изяснени инструкциите от този език.

Среда	Действия		Агент	Управление	
	Синтаксис (представяне)	Семантика (означава)		синтаксис	семантика
 Регистър – Единствен вид елемент.	Clr R	Нулирай съдържанието на регистър R		JMP Et. №	Премини към изпълнение на инструкция “ Et. № ”
	Inc R	Увеличи съдържанието на регистър R с единица.		JMP R1 R2 Et. №	Ако съдържанията на регистри R1 и R2 са равни, премини към изпълнение на инструкция “ Et. № ”
	Cpy R R1	Копирай съдържанието на регистър R1 в регистър R		Continue	А) Край на изпълнението Б) Продължавай към изпълнение на следващия ред

Примерната програма, която ще разработим, има за цел да бъдат събрани две цели положителни числа, записани в регистрите “А” и “В”, като сумата им бъде записана в регистъра “С”. Колкото и елементарна да изглежда тази задача, тя изисква разработване на алгоритъм. Списъкът с *действията*, познати на Агента показва, че “той не може събира произволни числа”. За да постигнем целта си посредством действия, изпълними от Агента, ще използваме действието “увеличаване на стойността с единица”.

Ето и алгоритъм, решение на задачата: В регистър “С” най-напред се копира стойността, записана в регистър “А”. Към тази стойност се добавя единица толкова пъти, колкото е стойността, записана в регистър “В”.

Този алгоритъм налага да се въведе контролна променлива “К” – регистър, предназначен за натрупване на толкова единици, колкото се добавят в регистъра “С”. Когато стойността в контролния регистър “К” стане равна на стойността в регистър “В”, процесът на добавяне на единици към “С” трябва да спре. В началото на изпълнението, регистър “К” трябва да съдържа нула.

1. Cpy **C A**

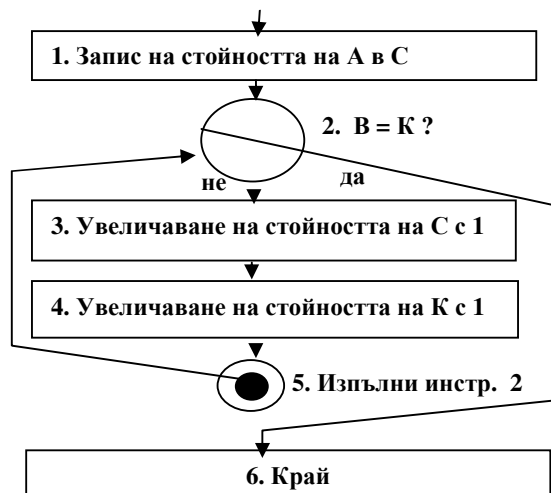
2. Jmp **B K 6**

3. Inc **C**

4. Inc **K**

5. Jmp **2**

6. Continue



Съставеният алгоритъм съдържа само действия, които са изпълними. Управлението, което се реализира по описания алгоритъм, е илюстрирано на схемата вдясно от програмния текст. Това е *схемата на управление* на алгоритъма. Проследяването тази схема по посока на стрелките, с примерни входни данни и с изпълнение на действията и проверките, е еквивалентно на изпълнение на алгоритъма от RAM - машина.

В таблицата долу е показано какви стойности биха се получили при изпълнение на тази програма с вход $A=6$ и $B=3$. Встрани от таблицата, за удобство при проследяването на работата на програмата, е дадена схемата на управление на алгоритъма. Проследете промяната над стойностите от регистрите стъпка по стъпка:



За проследяването на работата на програмен текст е удобно да се използват паралелно схемата на управление и таблица за получаваните стойности, както е илюстрирано с горния пример.

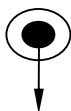
Принципът на работа на алгоритъма от примера е изразен достатъчно ясно чрез схемата на управление. Както може да се забележи, на тази схема инструкциите за действия са изобразени в правоъгълници, а инструкциите за управление обвързват действията, като с това образуват цялостно графично изображение на начина на работа на алгоритъма. На тази схема са визуално разграничени инструкциите за действията, и елементите на управлението, по които може да се проследи последователността на изпълнение.

1.3.2 Елементи на управление. Схема на управление на алгоритъм.

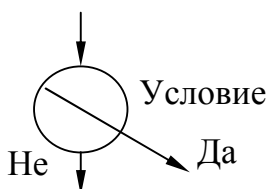
Приведеният пример за RAM-машината илюстрира какъв е принципът, по който работи един абстрактен автомат. Можем да твърдим, че изброените действия са наистина елементарни и че те могат да бъдат изпълнявани от реален автомат, дори на механичен принцип. (Съществуват реализации на такива автомати). Примерът е предназначен най-вече да илюстрира как, управляван от алгоритъм, един толкова елементарен автомат може да извършва и по-сложни действия, например събиране на числа. Изграждането на обвързана система от подобни процедури би увеличило възможностите на автомата. Така той би извършвал по-сложни действия.

Нека разгледаме отделно елементите на алгоритъма, които управляват последователността на изпълнение. В примера това са инструкциите за преход “Jmp B K 6” и “Jmp 2”, както и инструкцията за край на изпълнението “Continue”.

Инструкцията “Jmp Et №”, в контекста на цялостното управление на алгоритъма, има смисъл на задължителен преход към изпълнение на друга инструкция в друга “точка” от схемата на управление. Ще наричаме такава инструкция “Безусловен преход” и ще я изобразяваме с елемента:



Инструкцията “Jmp R1 R2 Et№”, в контекста на цялостното управление на алгоритъма, има смисъл на “разклонител” на изпълнението на последователността от действия, в зависимост от това дали дефинираното в инструкцията условие е изпълнено или не. Ще изобразяваме този вид елемент по подобие на “разклонител” и ще го наричаме “Условен преход”.

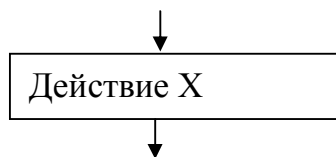


Важно е да се подчертае, че Условието в така дефинирания “разклонител” е проверка, която се отнася до получени в Средата стойности. В резултат на тази проверка се получава точно един от всички два възможни отговора – Да или Не. В примера проверката е “Е ли съдържанието на “С” равно на съдържанието на “К”?”. В зависимост от получените стойности на елементи от Средата, изпълнението на алгоритъма продължава “по една от двете стрелки”. Следователно описаният елемент на управление обвързва хода на изпълнение на алгоритъма със стойностите на елементите от Средата.

Двата елемента на управление – “Условен преход” и “Безусловен преход”, са основни елементи на управлението на машинен алгоритъм. Тяхното изпълнение от машина е възможно.

Инструкциите за действия и инструкциите за управление образуват програмен текст. Да приемем, че в езика за програмиране, използван за “изказване” на алгоритъма, са дефинирани точно определени инструкции за действия и за управление. За да изобразяваме алгоритъма така, че той да е едновременно образ на решението на задачата и първообраз на програма, ще използваме “схемата на управление” на алгоритъма.

Един алгоритъм включва множество инструкции за действия. Действията довеждат до промяна на стойности на елементите от Средата. Ще ги изобразяваме с правоъгълници, както е показано на следващата схема.



Преди да се изпълни действието X, средата е била в едно състояние, а след изпълнението му е в друго състояние.

Управлението, съществената съставна част на алгоритъма, задава в каква последователност да се извършват тези действия. За да представяме графично тази последователност, ще използваме базовите графични елементи за “Безусловен преход” и “Условен преход”.

1.3.3 Примерна схема на управление на алгоритъма на Евклид

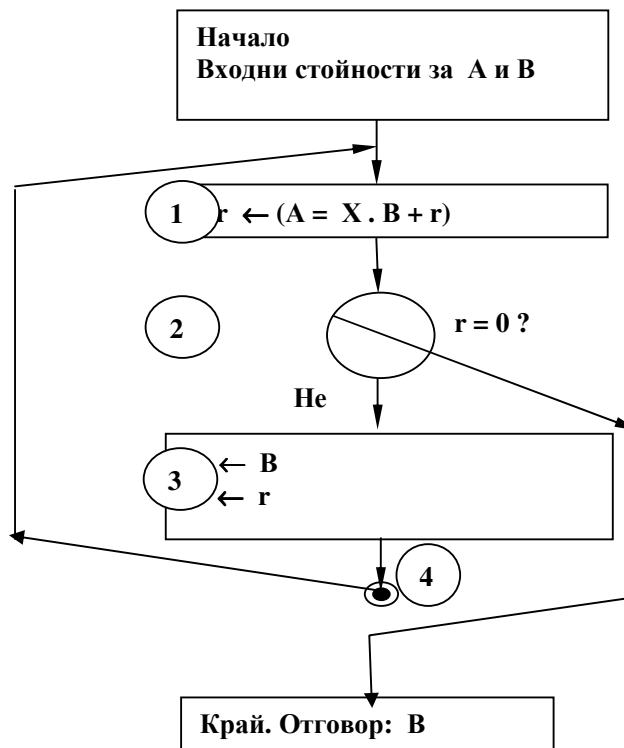
Нека на основа на изложеното дотук да приемем, че алгоритъмът – решение на задачата, е “удобен” за програмиране на машина, ако е изразен със схема на управление, в която участващите действия са изпълними от машината, а елементите на управление са “Условен преход” и “Безусловен преход”. За да се превърне такава схема в програма, достатъчно е всички елементи от схемата на управление да се запишат със съответните инструкции на “разбираемия” за машинната език.

За да покажем как се свързва абстрактния алгоритъм – решение на задача с алгоритъм, изпълним от машина, ще дадем пример за схема на управление, реализираща алгоритъма на Евклид. Да предположим, че действията, които една предполагаема машина е в състояние да извърши над среда от променливи с целочислено съдържание, са следните:

1. $r \leftarrow (A = X \cdot B + r)$	Намиране на остатъка r от целочислено делене на A и B за две цели числа A и B по правилото на целочисленото делене: $A = X \cdot B + r$
2. $A \leftarrow B$	Копиране на стойността на една променлива B в стойност на друга променлива A (както беше илюстрирано в парадигмата “кутия-съдържание”)

Както казахме, ще предполагаме, че всяка машина изпълнява елементите на управление “Условен преход” и “Безусловен преход”. (Това може да се покаже с много примери.) Нека тези елементи са “изказани” например с инструкциите: “go to **Етикет**” за безусловен преход и “If α **Етикет**” за условен преход, като условието е записано в частта “ α ”, и може да бъде например “ $r = 0$?” (Е ли стойността на променливата r равна на нула?)

Схемата на управление, реализираща алгоритъма на Евклид и предназначена за изпълнение от предполагаемата машина е показана по-долу, като са спазени номерата на стъпките, с които този алгоритъм беше представен отначало:



Като се следва показаната горе схема, може да бъде записана програмата за предполагаемата машина-изпълнител, а после – приведена в изпълнение.

♦ Примерна задача за упражнение

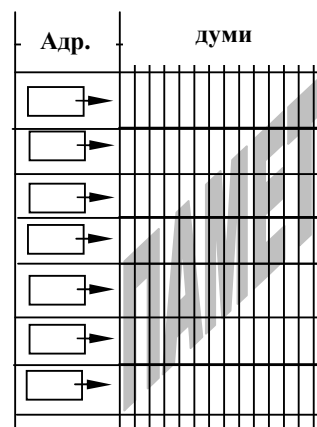
Като използвате инструкциите от представения предполагаем език за програмиране, съставете програмен текст, реализиращ алгоритъма на Евклид. Приемете, че инструкции за въвеждане и извеждане (разпечатване например) на стойности са съответно “Input [...]” и “Output [...]”.

1.4 Реализиране на алгоритъма като програма

1.4.1 Основни постановки

В предходната тема разгледахме един от моделите на абстрактна машина, изпълняваща алгоритъм. Работата на RAM-машина може да бъде управлявана от програма на елементарен език, на който е изказан алгоритъм. Като приехме за основа модела на RAM-машина, показахме как би изглеждал алгоритъмът на Евклид в примерен “програмируем” вид.

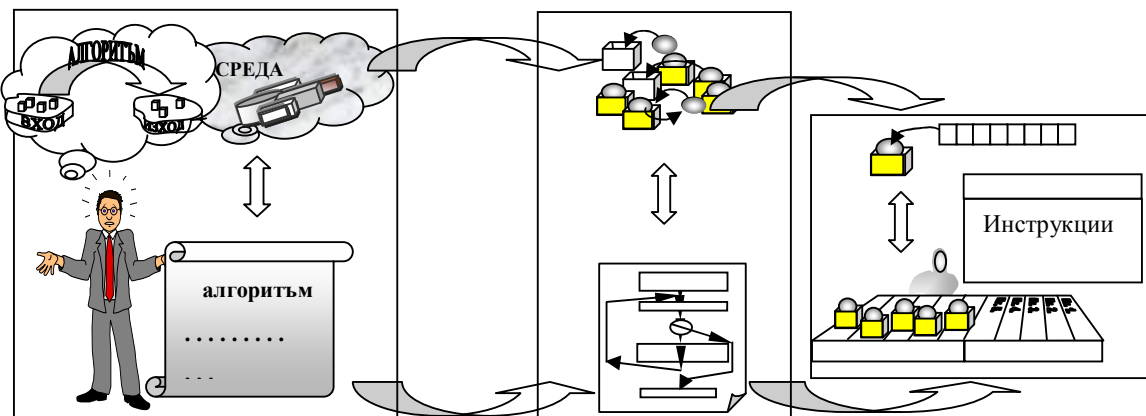
Можем да приемем, че принципът на работа на съвременния компютър е реализация на модела на RAM-машина. Да проследим съответствието, като припомним най-напред един известен факт: информационната единица в компютъра е така нареченият “бит”, физически реализиран като “няма-има” и математически интерпретиран като нула - едно.



Да представим най-общо принципа на организация на машинната памет: тя е съставена от паметови единици, наричани “машинни думи”. Машинната дума е последователност от битове, както е илюстрирано на фигурите горе, или още – машинната дума е едномерен масив от битове. Броят битове в една дума е краен, фиксиран за всяка определена реализация на машина. Всяка дума има адрес, който служи за идентифициране на нейното физическо място. Стойността, записана в думата, е последователност от нули и единици, т.е. тя се представя чрез двоичен вектор. Паметта е крайна последователност от адресирани машинни думи, т.е. вектор от машинни думи. Физически реализирана на този принцип, машинната памет изпълнява ролята на “памет за данните” и на “памет за програмата” от RAM-модела.

Ролята на абстрактния Агент от RAM-машината се изпълнява от процесора на компютъра. В така организираната физическа памет, той е в състояние да извършва почти същите действия като тези, които изброихме за абстрактния му RAM-първообраз. На това машинно равнище и най очевидното за човека аритметично действие се реализира при управление от алгоритъм, работещ над стойност, представена с двоичен вектор. Изчислителната машина работи по този начин и агентът-процесор това “може”.

Тази постановка, макар и твърде обща, дава представа за много голямата разлика между програмен алгоритъм като решение на задача и възможните действия на RAM-машината като изпълнител.



На предходната илюстрация е показана образно “комуникация”, която би свързала алгоритъма като решение на задача, с машината-изпълнител.

Да приемем, че дадена задача има решение с алгоритъм. Нека алгоритъмът е намерен като преобразование, отговарящо на всички изисквания, които дефинирахме като “свойства” на алгоритъма. Той е представен като последователност от действия в абстрактната Среда. Действията са изобразени на предходната схема с вертикална стрелка, илюстрираща факта, че посоченото в текста на алгоритъма е “въздействие” над абстрактната среда.

За бъде изпълнен от машина, този алгоритъм трябва да се дефинира като преобразование, което се извършва в Среда, представима в машината. Алгоритъмът трябва да включва действия, които са изпълними от машината. Логиката на алгоритъма трябва да е представима като инструкции за управление на последователността на извършваните от машината действия. Така алгоритъмът може вече да се “преведе” с инструкции, “разбираеми” за машината, да се оформи като програма и да се изпълнява от машината.

На предходната фигура умишлено са представени отделно пътищата на “метаморфози” на средата, действията и управлението, свързващи абстрактния алгоритъм с машината. Ще проследим поотделно пътя на средата с действия и пътя, съответстващ на управлението.

1.4.2 Езикът като средство за формулиране на среда и действия

Основна функция на *езика* е да бъде средство за комуникация. Да разсъждаваме за езика за програмиране като за език, който служи за “комуникация” с машината, при това става въпрос за еднопосочно “съобщаване” на инструкции.

Да си представим, че между алгоритъма, създаван от разумен човек и машината-изпълнител има “нива” на удобство за комуникация (от гледна точка на човека). Например, ако в езика може направо да се представи понятието “права в равнината” и съществува изпълним оператор за “намиране на пресечна точка на две прави”, този език би бил удобен при изказване на алгоритми за геометрични задачи. Ако представим най-общо езика за програмиране като краен брой от инструкции, стои въпросът за това в каква степен те са “образи” на понятия, с които човек си служи при дефиниране и решаване на задачи. Да приемем, че има езикови единици, които могат да се използват направо за “съобщаване” на елементите от средата и на действията, които трябва да бъдат извършени в нея. Да наречем този набор “*език за среда и действия в нея*”.

Най-близко до машината е езикът, с който се управляват елементарните действия на процесора. Алгоритмична задача, типична за това най-първично ниво е например съставянето на изчислителна процедура за събиране на двоични вектори при използване на действието “добавяне на единица”.

В езиците, наричани “машинни езици” съществуват единици, съответстващи на адресирани машинни думи, а командите за действия “работят” с регистрите и техните адреси. Най-общо, тези езици са съставени и работят по

иллюстрирания за RAM-машината принцип. Такъв е например езикът Assembler, близък до приведения примерен RAM-език.

Системите за двоично кодиране позволяват да бъдат съставени двоични “образи” на цели и реални числа, на печатни символи. За някои от извършваните операции в множествата например на целите и на реалните числа, са разработени съответни алгоритми, работещи с двоичен код. Това позволява в програмните езици да се въведат изразни средства за дефиниране на променливи със стойности от тези множества, а действията с тях да се означават по обичайния за човек начин. Такъв “език за среда и действия в нея” позволява например да се “съобщи” направо, че трябва да се изчисли резултатът от деленето на две реални числа.

Това езиково представяне на среда и действия не изчерпва понятията и обектите, с които въображението на човека си служи, за да формализира и решава възникналите задачи. Съществуват разнообразни мислени структури, елементи на абстрактни “Среди”, при използването на които много класове от задачи са решими с алгоритъм, а организирането и обработката на големи количества данни е възможно. Могат да се посочат за пример вектори, матрици, стекове, дървета, графи и т.н. Разработени са езици, които позволяват дефинирането на такива структурирани елементи в средата да става по естествен начин, а алгоритъмът-решение да се “изказва” лесно с програмен текст. Очевидно самото създаване на такова езиково средство става при откриване на принципи, по които могат да се обвържат разнообразните абстрактни обекти и извършваните над тях операции, като при това се изобрети и начинът на тяхното представяне в машината.

Нека приемем, че алгоритъм, разработен на език за дадено ниво, може да се представи като инструкция за действие в език от по-високо ниво. Макар, че така “езиците за среда и действия” са представени много общо, това представяне дава частична представа за така наречените “процедурни езици” за програмиране. Развиват се все по-абстрактни езици, с които все по-малко може да се контролира в детайли хода на изпълнение от машината. При всички положения, управлението на работата на компютъра се основава на използването на алгоритми на всички нива на “комуникация” с него.

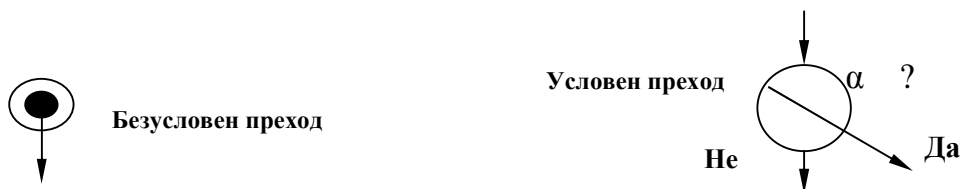
Тук алгоритъмът ще се разглежда в контекста на материала, изложен в първа глава, т.е. – алгоритъмът като “постъпкова процедура”, която е решение на алгоритмична задача. За представянето на разглежданите тук алгоритми е избрано обобщено представяне, съответстващо на подхода на така нареченото “процедурно програмиране”. Избраното представяне на алгоритъм чрез схема за управление позволява примерните алгоритми да бъдат програмно реализирани при ползване на езици с изказ, близък до естествения език, както и с близки до съзнанието среди и действия. Такива са Pascal и C, традиционно използвани като опорни програмни езици в курсовете по алгоритми.

1.4.3 Организиране на последователността на действията

Реализирането на “логиката” (в общ смисъл) на алгоритъма става при използване на тези инструкции, които нарекохме “инструкции за управление”. Ще проследим какво управление е приложимо в машината-изпълнител и какви “логически конструкции” възникват и се използват на абстрактното ниво на алгоритъма-решение.

➤ Базови елементи на управлението

Инструкцията за безусловен преход има смисъл на задължителен преход към друга “точка” от схемата на управление. Такава инструкция съществува във всички процедурни езици за програмиране и обикновено изказът ѝ е дословен (английски) израз на значението ѝ, например “Go To **Етикет**”.



Инструкцията за условен преход опириличихме на “разклонител”. Описаният “разклонител”, в момента на изпълнението си, извършва проверка α , която сравнява определени стойности, получени до този момент. Резултатът от извършването ѝ е от логически тип “Да-Не”. Инструкцията от типа “Ако α е изпълнено, то – премини на...” съществува във всички процедурни езици и обичайно е с изказ, подобен на “If α then **Етикет**”.

Този елемент на управление обвързва хода на изпълнение на алгоритъма със стойностите на елементи от Средата. В зависимост от получените стойности, изпълнението на алгоритъма продължава “по една от двете стрелки”. Най-елементарните проверки α се отнасят до сравнение на две стойности и имат смисъл на въпроси от вида на:

α ? : $\boxed{A} = \boxed{B}$ “Е ли съдържанието на променливата A равно на това на променливата B?”

α ? : $\boxed{A} > \boxed{B}$ “Е ли съдържанието на променливата A по-голямо от това на променливата B?”

Да представим сега управлението като система от обичайно използвани конструкции, с които се дефинира желаната последователност на извършваните действия. Може да се приеме, че описанието на алгоритъм като постъпкова процедура се представя с :

Последователности, Клонове, Повторения.

При организиране на управлението в конкретен алгоритъм възникват някои типови схеми за организация. Те могат да бъдат реализирани с изброените два базови елемента за управление и следователно могат да се изкажат с кой да е “език за среда и действия”. В повечето езици съществуват готови фрази за изказ на тези схеми. Ще ги наречем “конструкции за управление”.⁵

⁵ Другият използван термин е “Структури за управление”.

1.4.4 Конструкции за управление

Последователност. Блок

Ще разгледаме този конструкт, за да обърнем внимание, че съществуват различни езикови средства за неговата реализация. В някои езици за програмиране се използва номериране на всички инструкции в програмния текст с така наречените “етикети”, за да се укаже, че определена поредица от действия трябва да се изпълнява в посочената последователност. Това средство за указване на последователността на изпълнение се използва например в BASIC.

В повечето съвременни процедурни езици последователността, в която трябва да бъдат изпълнявани действията се задава чрез последователността в самия програмен текст. Всяка промяна на исканата последователност на изпълнение става с корекция на съществуващия програмен текст. Последователността е конструкт, който ще изобразяваме по естествен начин така:



Разглеждането на последователността като конструкт позволява въвеждане на понятието “Блок от последователни действия”. Илюстрираната горе последователност представлява такъв блок. Той има единствена входна точка и единствена изходна точка. Под “Блок” ще разбираме такава част от схемата за управление, която има единствена входна и единствена изходна точка.

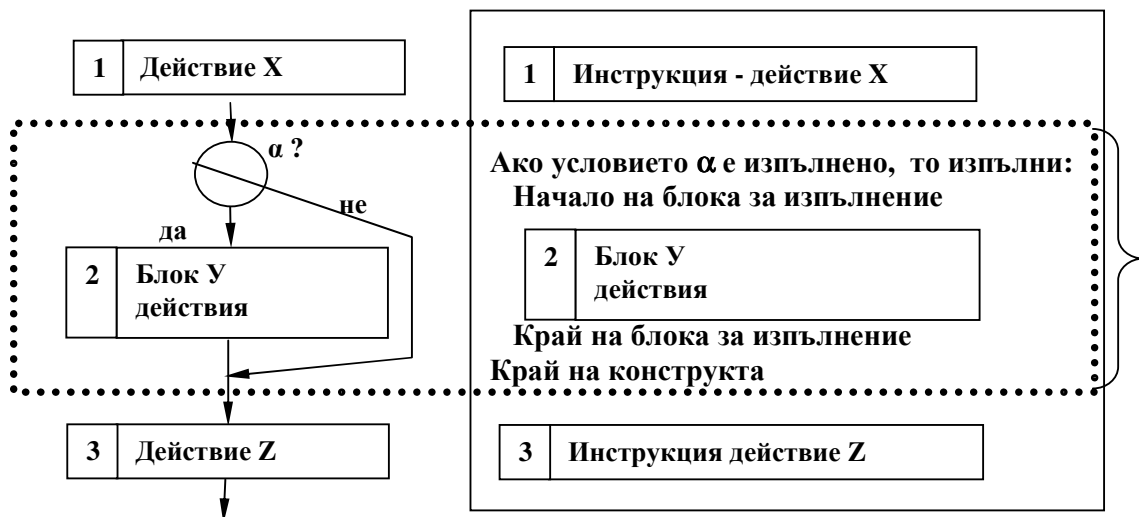
Конструкции – Клонове

Характерно за тези конструкции е, че те предоставят възможност изпълнението на алгоритъма да става в различни последователности в зависимост от стойностите, получени в средата. Основен при реализирането на клоновете е елементът на управление “условен преход”. Ще разгледаме три вида клонове – Байпас, Двуклон и Многоклон. Както е показано и на следващите схеми за управление, всички тези конструкции са блокове с единствена входна точка и единствена изходна точка.

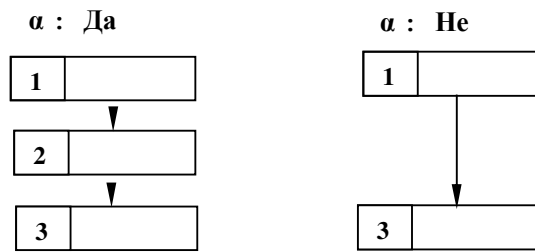
Байпас (“мини покрай”)

Както се вижда от начина, по който сме я назвали тук, тази организация на управлението “работи”, като дава възможност да бъде “прескачано” изпълнението на определен блок.

Схемата на управление, съответстваща на конструкта “Байпас” ще представяме графично както е илюстрирано:

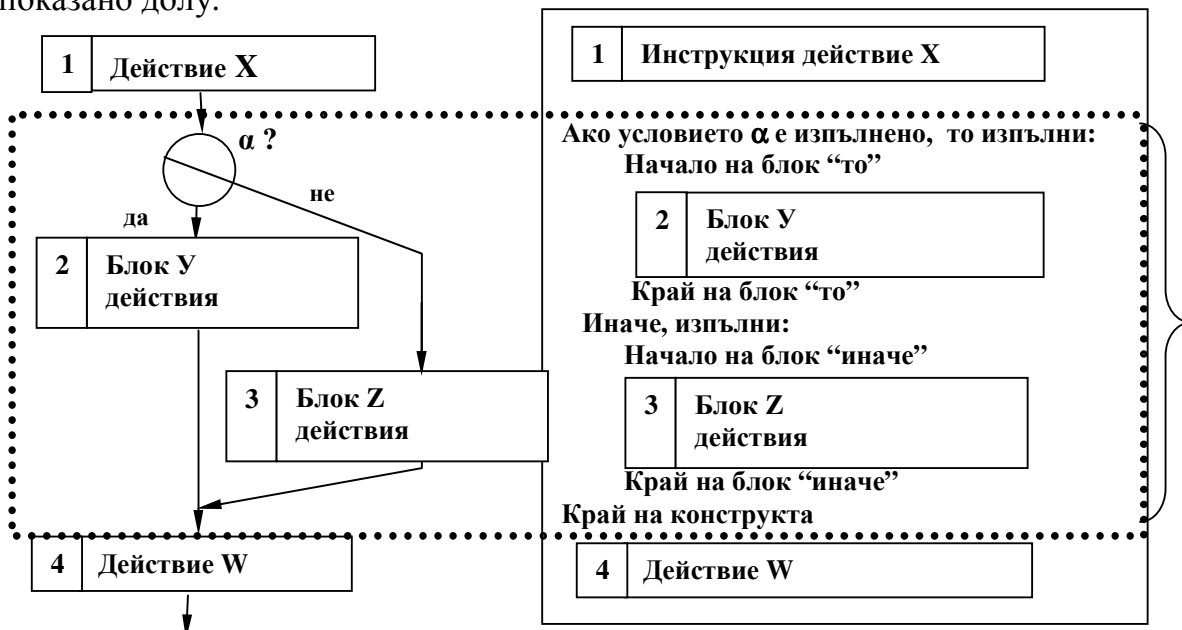


В момента на “навлизане” в този конструкт, стойността на α е или “Да” или “Не” (в зависимост от стойностите на определени променливи от Средата). Изпълнението на алгоритма ще премине по **точно един** от следните два пътя:

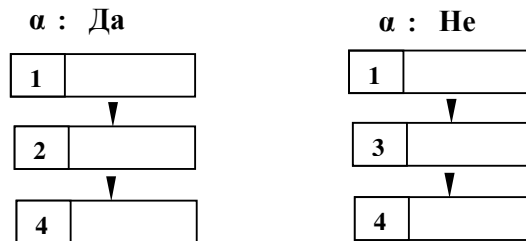


Двуклон

При организация на управлението по конструкта “Двуклон”, изпълнението на алгоритма преминава през един от два възможни блока, организирани както е показано долу.

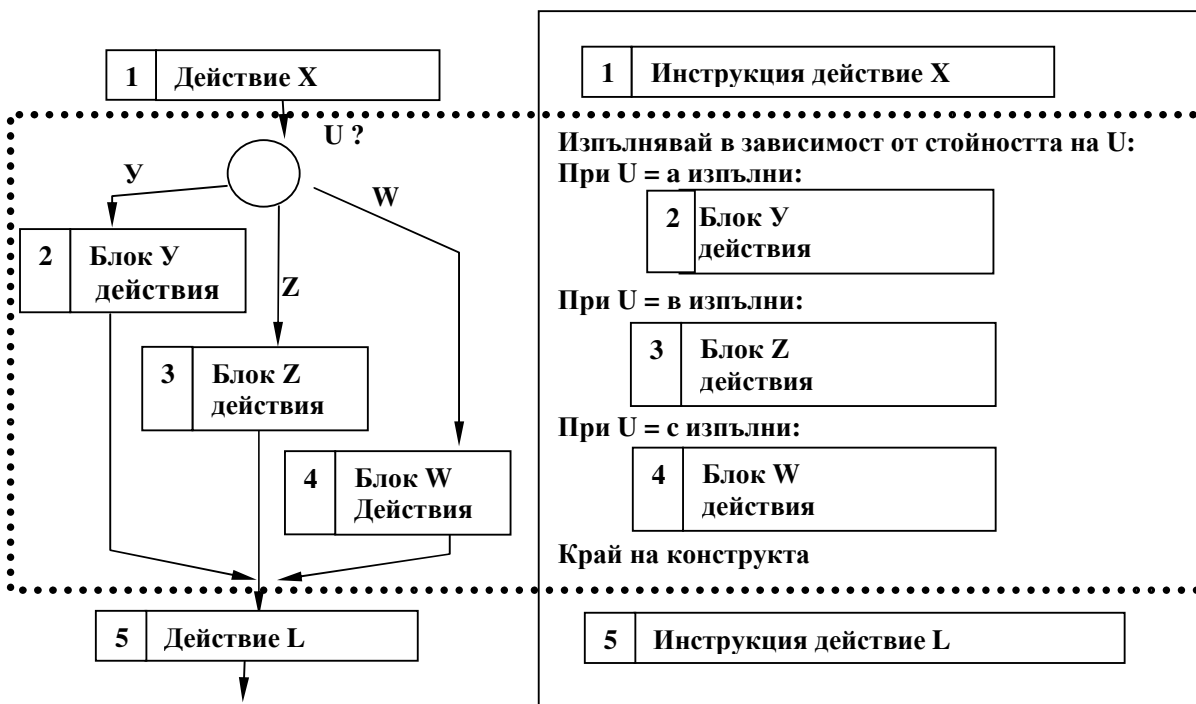


В зависимост от резултата, получен при извършване на проверката, алгоритъмът довежда до изпълнение на точно една от двете последователности, илюстрирани долу:

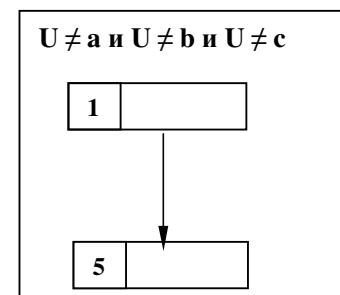
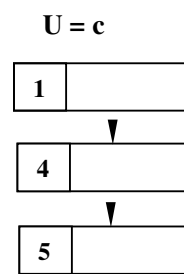
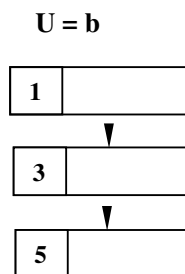
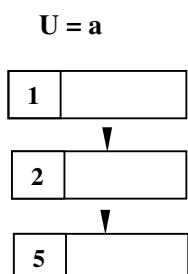


Многоклон

Подобно на байпаса и двуклона, конструктът, наречен тук “многоклон” обвързва пътя на изпълнение на алгоритъма с получени в средата стойности. Проверката, която се извършва при “влизване” в блока на този конструкт има (по идея) повече от два възможни отговора. Както е илюстрирано долу, тази проверка прави сравнения за равенство на стойността на един “управляващ” израз U с няколко шаблонни стойности.



В зависимост от стойността, получена за управляващия израз, алгоритъмът изпълнява точно една от следните последователности:

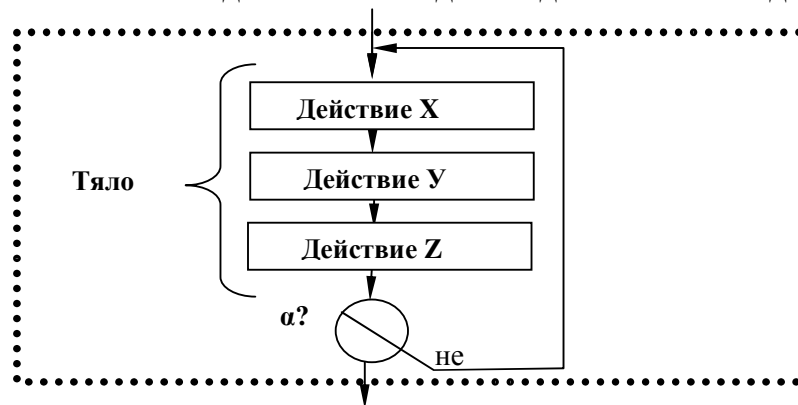


Повторения

Повторенията са конструкции, които могат да се реализират при използване на базовите елементи на управление “условен преход” и “безусловен преход”. Те могат да се оформят като блокове в схемата на управление, както това е илюстрирано за различните видове повторения, приведени в примерите долу.

Цикъл със следусловие

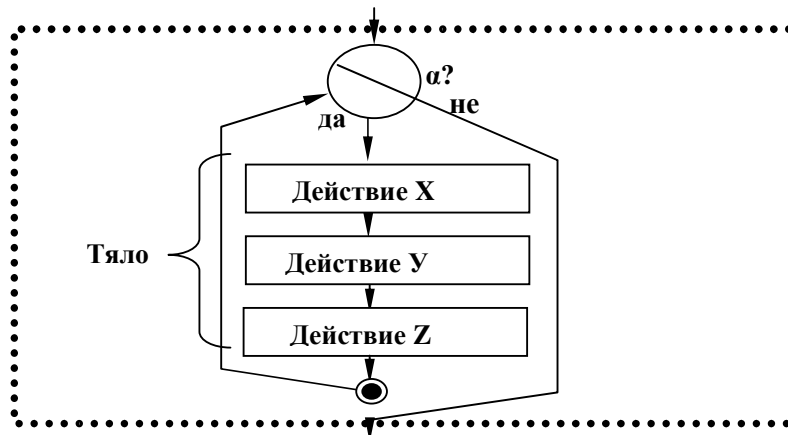
Както е показано на следващата схема за управление, при този конструкт, една от “стрелките” на проверката α “препраща” изпълнението към предходна точка от алгоритъма и изпълнението на някои инструкции започва отново. Блокът, който се повтаря, се нарича “тяло” на цикъла. Самият конструкт “цикъл със следусловие” е блок с единствена входна и единствена изходна точка.



Цикълът, показан на примерната схема горе, ще предизвиква изпълнение на действията XYZ, XYZ, XYZ дотогава, докато стойностите, на които се основава проверката α ?, станат такива, че изпълнението да продължи през изходната точка на блока.

Цикъл с предусловие

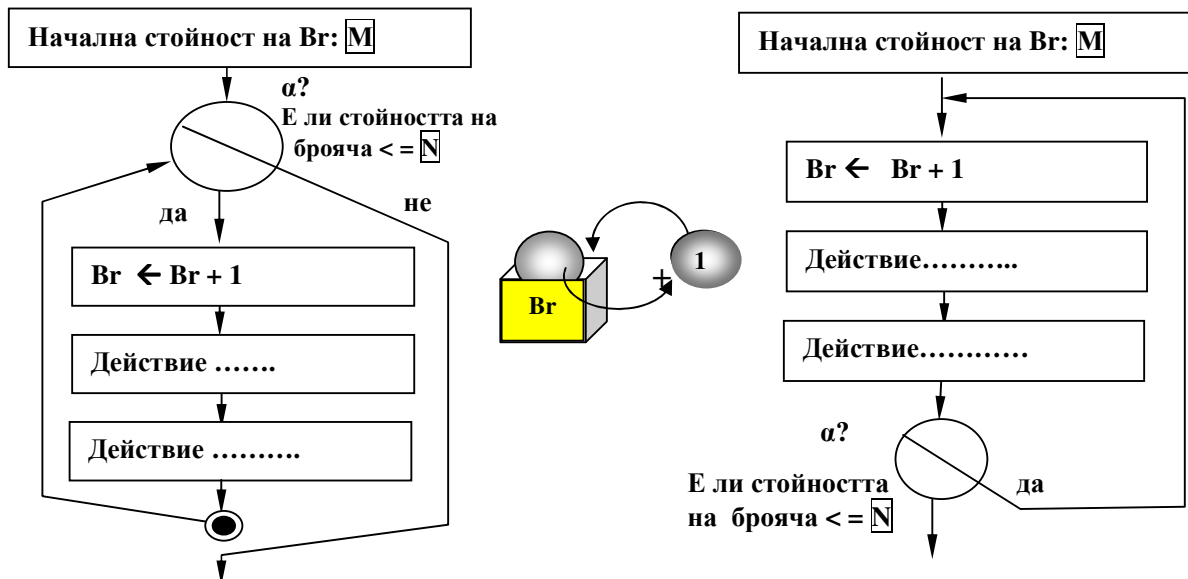
Цикъл с предусловие се “конструира” с елементите на управление на машината така, както е илюстрирано долу. Този цикъл налага използване на безусловен преход към проверката α , която е преди тялото на цикъла. Тялото на този вид цикъл може да не се изпълни нито веднъж, ако стойностите в средата са такива, че α насочи изпълнението направо към изходната точка на блока.



Повторения определен брой пъти

Разгледаните конструкции за организиране на повторения при изпълнението на алгоритъма могат да се използват по два начина в зависимост от това какъв е смисълът на условието за край на цикъла. Циклите, с които може да се предизвика изпълнение на тялото “докато това се налага” се наричат “итеративни цикли”. Циклите, с които се дефинира повторение на тялото точно определен брой пъти ще наричаме “цикли по брояч”.

Циклите по брояч се реализират посредством специално предвидена променлива-брояч, която “следя” колко пъти се преминава през тялото на цикъла. Основната схема на “отброяването” е то да става чрез увеличаване на стойността на променливата-брояч в тялото на цикъла. Условието за край на цикъла определя колко пъти трябва да бъде изпълнено тялото и се отнася до стойността на тази променлива. На двете схеми долу е илюстрирано изграждане на цикъл по брояч B_r при използване на предусловие и следусловие. И при двете схеми броячът има начална стойност M и се увеличава до крайна стойност N , посочена в условието за край на цикъла. Очевидно по подобен начин могат да бъдат конструирани цикли, в които стойността на брояча се променя със стъпка, различна от единица, както и цикли, при които стойността на брояча намалява.



На предходните схеми ясно личи, че в тялото на цикъла не трябва да има допълнителни промени на стойността на брояча, защото това би нарушило функцията му и би попречило на цялото управление.

Всички изложени дотук конструкции за управление могат да се реализират и при използване на езика Assembler и на всеки друг език за програмиране, например Fortran, Algol, Pascal, C, Ada и т.н. За целта е достатъчно елементите на управление “условен преход” и “безусловен преход” от горните схеми да се заместят със съответните оператори.

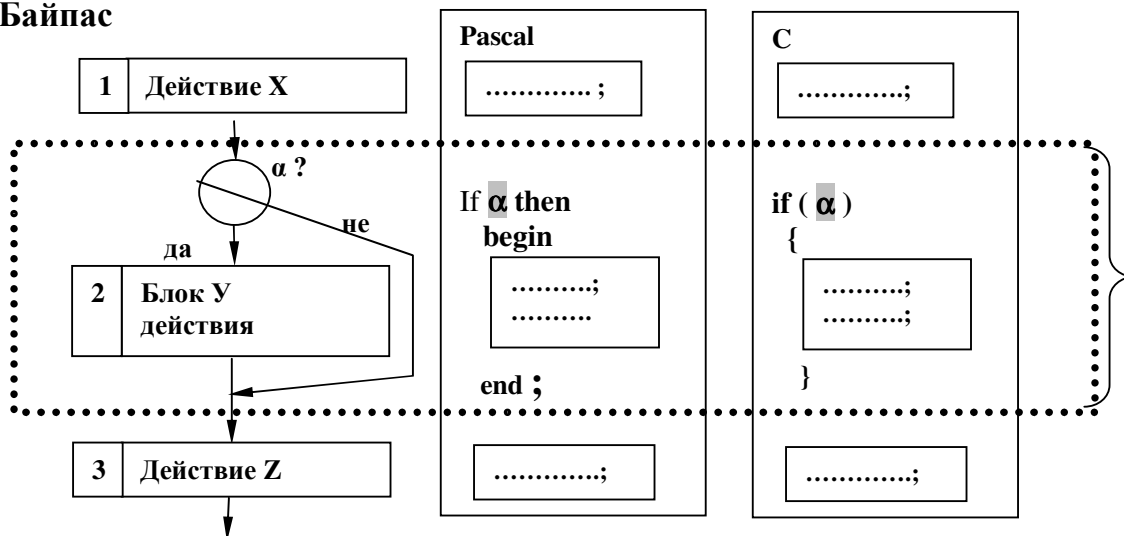
1.4.5 Езикови фрази, реализиращи конструкции за управление

В процедурните езици, конструктите за управление могат да бъдат изказани с готови езикови фрази. Така програмният текст се доближава до изказ на естествения език, а с това се постига по-голяма яснота и удобство при работата с него. Чрез такива готови фрази, правилната вътрешна организация на конструктите за управление се спазва “по принуда”, с което намалява вероятността за логически грешки при писането на програма. Например, вместо да се организира цикъл с използване на инструкции от рода на «Ако “това”, иди да изпълняваш етикет 231» и «Иди сега на етикет 26», използва се фраза от типа «Докато не стане “еди какво”, прави “това”». Аналогично, вместо да се създава специална променлива-брояч и да се пресъздава някоя от схемите за цикъл по брояч, направо се използва фраза от типа на «Прави “това” за стойности на брояча от М до N».

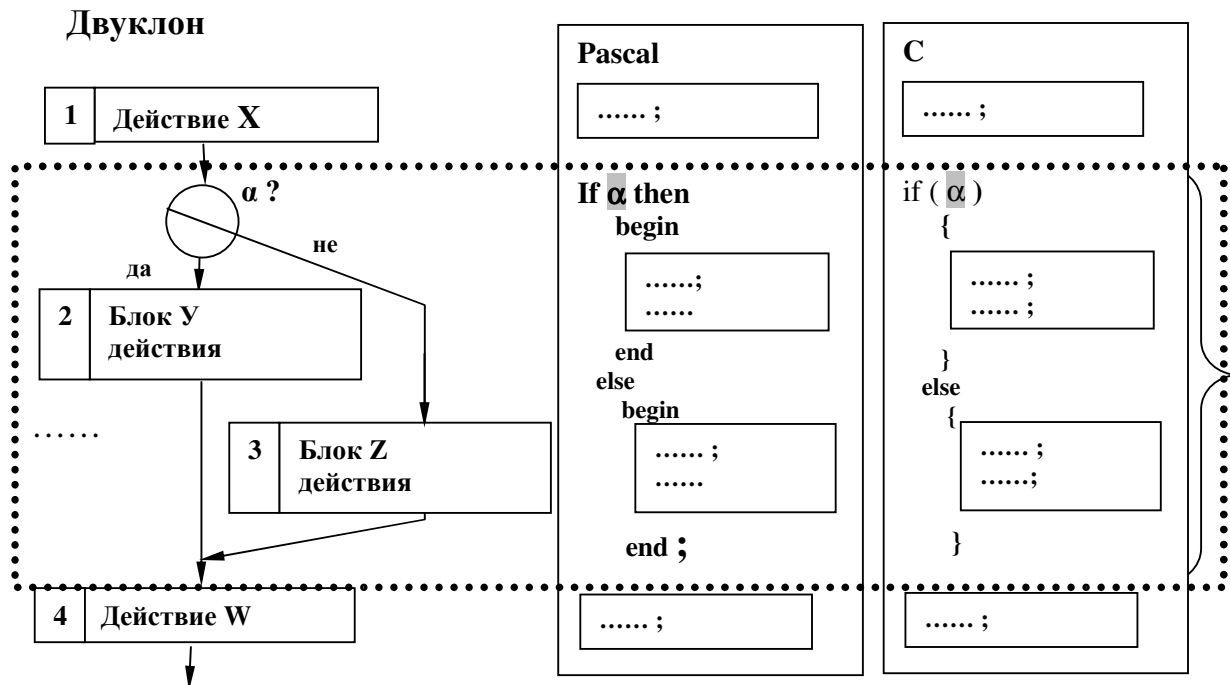
Конструктите, използващи в организацията на управлението безусловен преход се изразяват с готови фрази от типа на горните. По този начин е избегната необходимостта от използване на инструкция за безусловен преход е избегната. Езиците, в които това е възможно, се наричат “структурни”, а самото програмиране – “структурно”. Терминът “структура” се употребява тук в тесен смисъл, т.е. като изграждане на програмен текст чрез конструкции за управление.

На схемите, които са приведени долу, е илюстрирана езиковата реализация на някои от конструктите с фразите на два примерни езика – Pascal и C. Двата езика ползват за фразите служебни думи.

Байпас



Нека обърнем внимание на факта, че байпасът и двуклонът (и в двата езика) са реализирани със свои собствени фрази. Наличието на една и съща служебна дума – “if” и в байпаса, и в двуклона, е причина за тяхното недостатъчно ясно разграничаване от начинаещи програмисти. Синтактически, наличието в Pascal на знак за край на фраза “;”, поставен след края на блока “then”, означава и край на конструкта. Да проследим разликата в синтаксиса на двуклон и байпас със следващата схема:

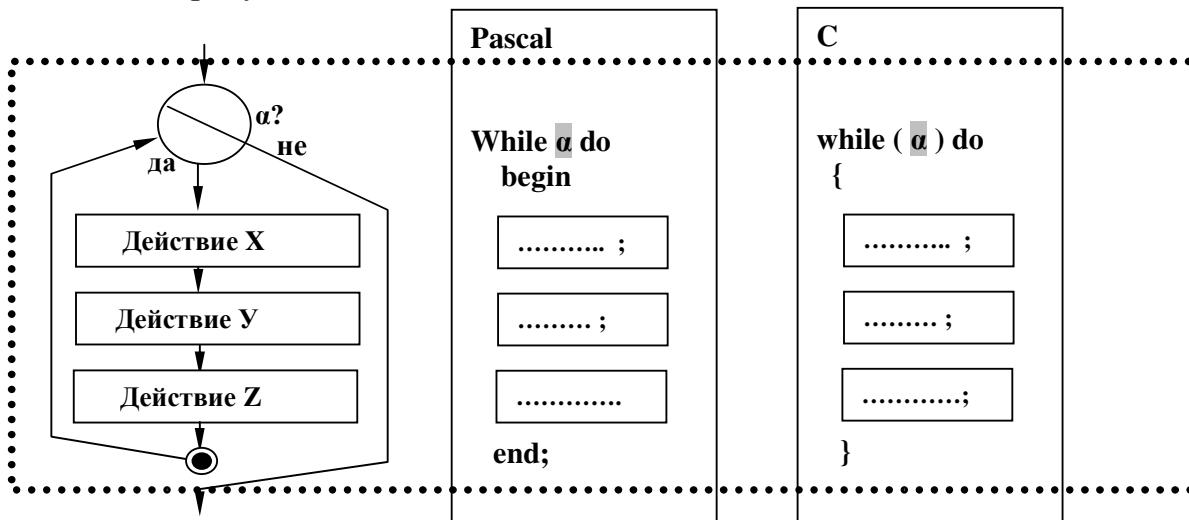


Цикли

По принцип са възможни два варианта в зависимост от това на основа на каква схема авторите на един език са избрали да конструират определена фраза за цикъл – цикълът да продължава, ако условието α е изпълнено или цикълът да продължава, ако то не е изпълнено. Тази разлика ще изказваме на български съответно с два различни изказа : “Щом като” и “Докато”. Схематично, разликата се изразява в това дали на изхода от проверката α стрелката, насочена към край на цикъла, съответства на “не” или на “да”.

Да разгледаме някои примери за фрази за програмно реализиране на разгледаните конструкции, осъществяващи цикли.

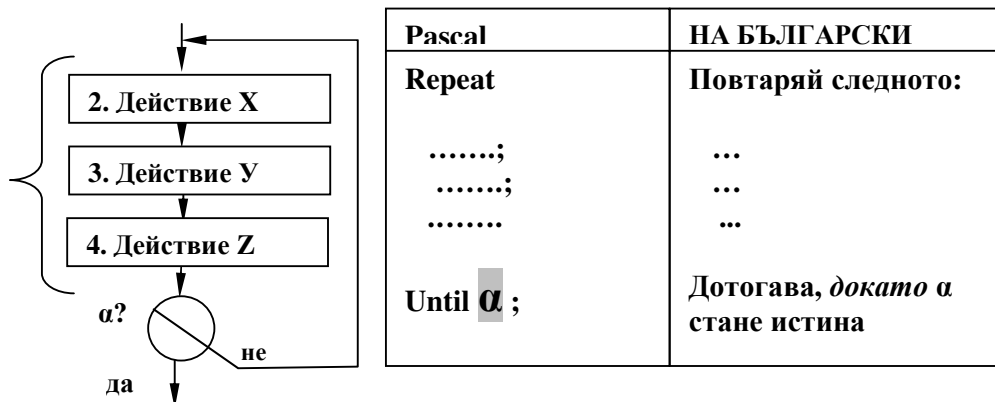
Цикъл с предусловие.



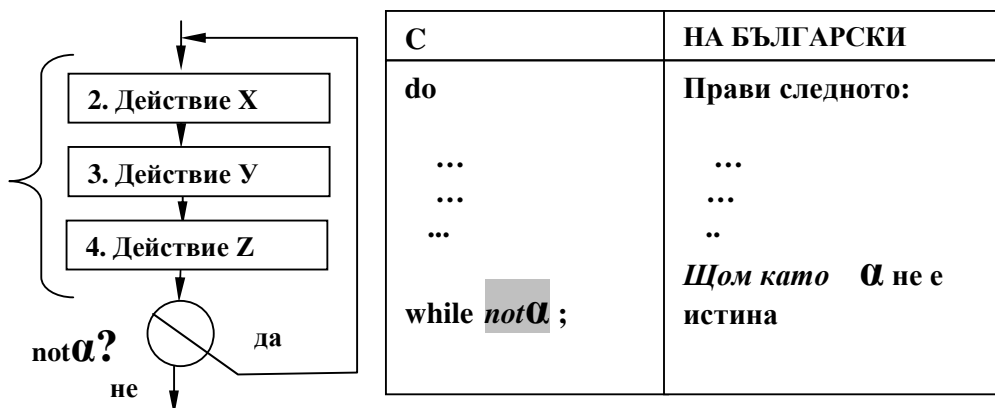
Изказана на български, фразата за цикъл с предусловие би звучала така : “Щом като α е вярно, прави това – оттук, дотук”. Както е показано, и двете примерни фрази реализират схема, при която в тялото на цикъла се влиза при α – да.

На следващите две схеми за управление се илюстрира един и същи цикъл със следусловие. Неговото изпълнение приключва, когато една проверка α за стойности в средата, даде резултат “истина”.

В първата схема е дадена фразата за този цикъл, реализирана на Pascal. Както е посочено на схемата, от цикъла се излиза в случай на удовлетворено условие α , формулирано с “Докато”.

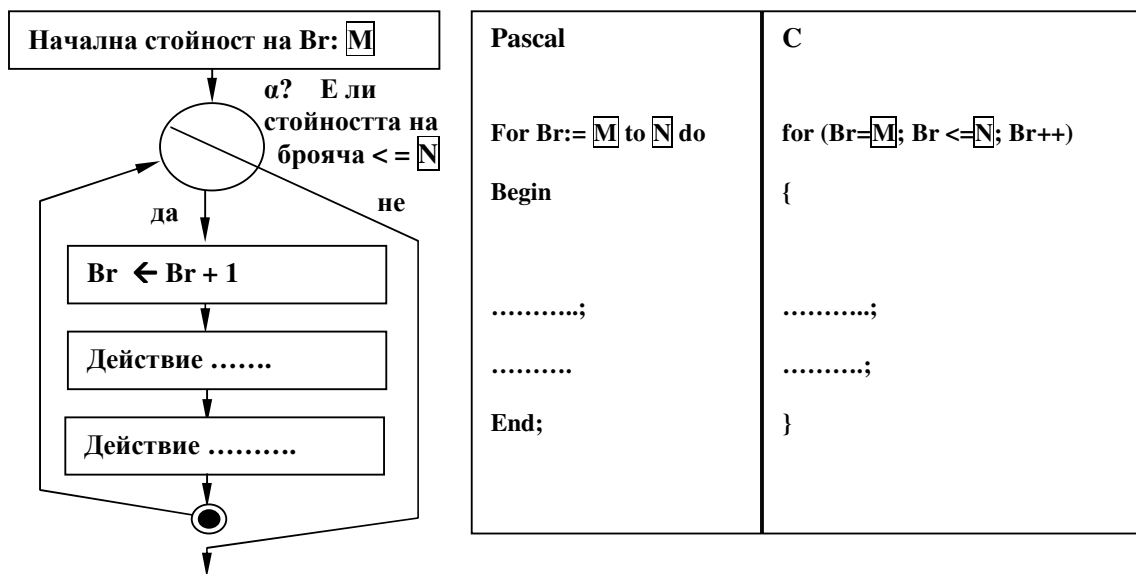


Избраната в езика C схема се различава от горната по това, че фразата за цикъл със следусловие има следния смисъл: щом като условието, посочено на края на цикъла е изпълнено, изпълнението на тялото продължава. Това е изказано с познатото от циклите с предусловие “Щом като”. Схемата на тази фраза е илюстрирана долу. Както е показано, когато проверката в края на цикъла е с отговор “да”, цикълът продължава. Поради това, за да реализираме цикъл, функционално еднакъв на горния, в примера долу се проверява за истинност на $\text{not } \alpha$ – обратното на α . Така двата цикъла, “паскалският” и този на C, “работят” еднакво.



Да завършим това начално представяне на конструктите за управление с примерни фрази за конструкта “цикъл по брояч”. В съществуващите готови фрази броячът е вграден, никакъв отделен оператор не се грижи специално за началната му стойност, нито за увеличаването (намаляването) му.

Цикъл с вграден брояч – нарастващ.

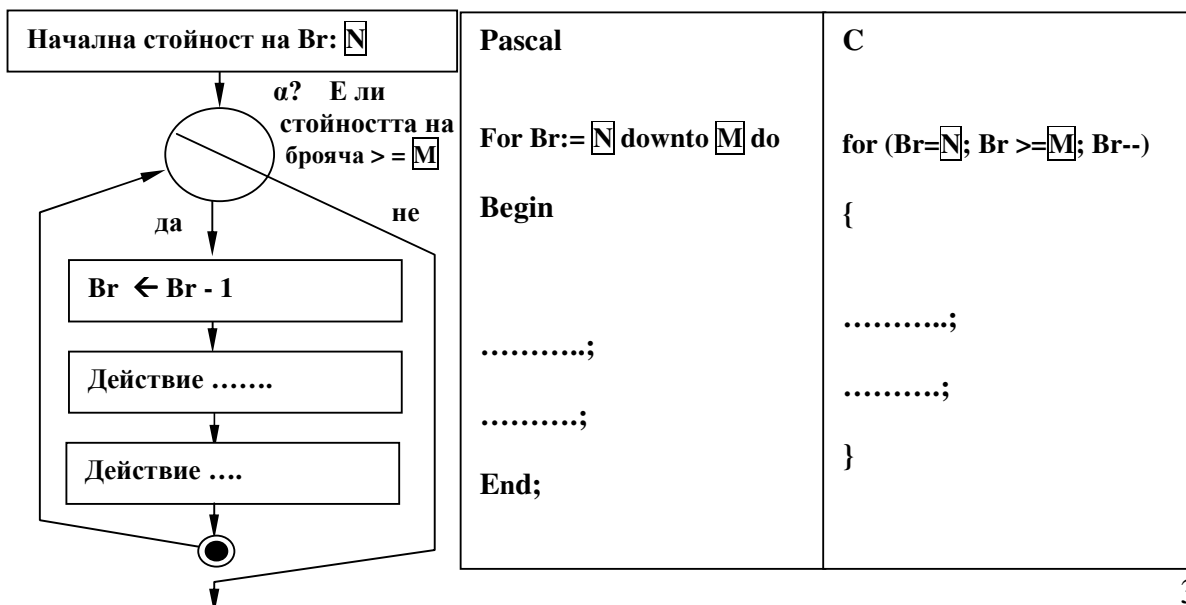


В стандартния Pascal M и N са изрази, които се изчисляват преди започване на цикъла и не могат да се преизчисляват по време на изпълнението му. Управлението с оператор *for* в C++ съдържа три елемента в скобите, разграничени с “;” – инициализация, условие за продължение и подготовка за нов такт на цикъла. По такъв начин горният *for* е еквивалентен на:

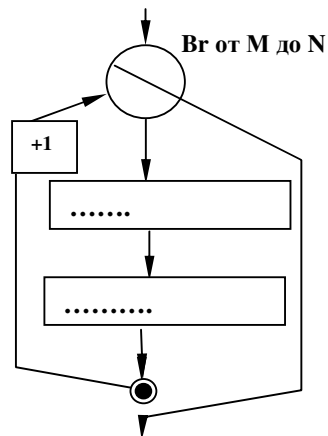
```

Br=M;
while (Br<=N)
{
.....;
Br++
}
  
```

Цикъл с вграден брояч – с намаляваща стъпка.



Ще наричаме горните фрази “цикъл по вграден брояч” и ще ги изобразяваме условно така:



♦ **Примерна задача за упражнение**

Задача 1. Да се състави алгоритъм и програма (схема на управление и съответстващия ѝ програмен текст) за намиране на най-голямото от три числа.

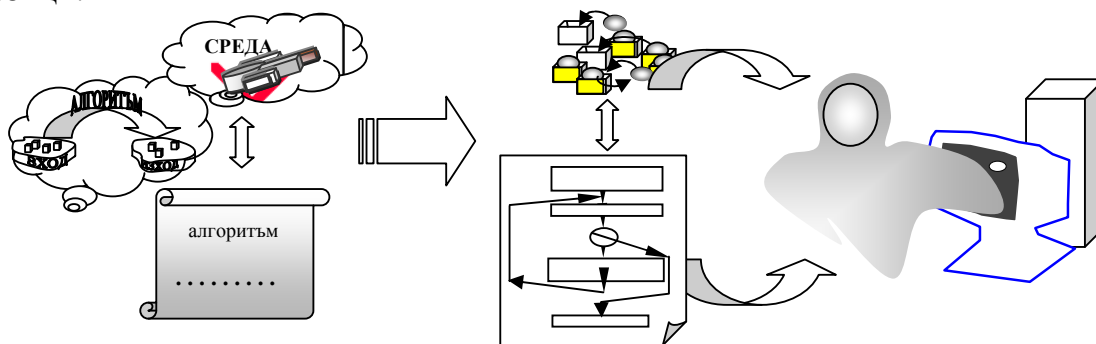
Задача 2. Да се състави по три схеми – предусловие, следусловие и брояч, цикъл за извеждане на екран на съобщението “изпълнявам цикъл” седем пъти.

Задача 3. Същият резултат като този на Задача 2 да се получи без използване на готови фрази за цикъл.

2 БАЗОВИ ПОЗНАНИЯ ПРИ ПРОГРАМИРАНЕ

➤ *Обща постановка*

В този раздел се разглеждат някои базови постановки на програмирането, както и принципите на някои алгоритмични похвати, представени с примери и коментари. В съответствие с изложеното в първата част, при избрания подход алгоритъмът е представен като система от изпълними от компютъра “действия”, довеждащи до промяна на стойности в “среда”, която може да се представи направо със средствата на език за програмиране. Основно, алгоритъмът е представен тук със “схема на управление”, състояща се от блокове-конструкти за управление, съответстващи на фрази от програмните езици Pascal и C. Действията в схемите на управление съответстват на изпълними оператори от тези езици. Може образно да се каже, че алгоритмите тук са за “абстрактен Агент”, който “разбира” Pascal, C и всички подобни на тях езици.



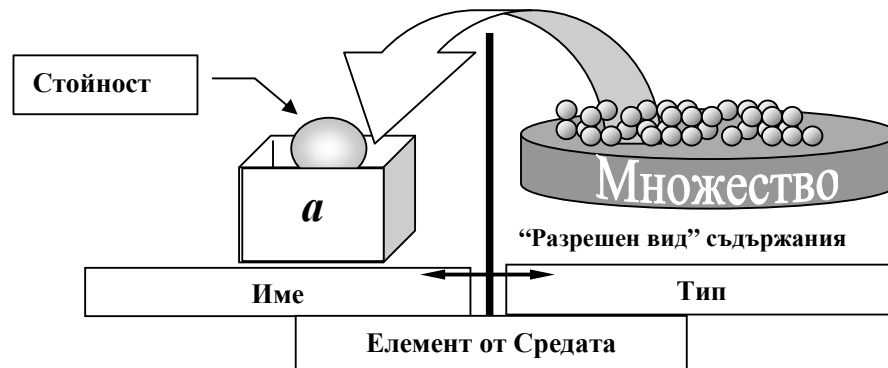
Представените по този начин алгоритми съответстват на езиковите изразни средства и могат лесно да се преобразуват в програми. Вниманието ни е концентрирано над принципните постановки при програмиране – типовете, използването на конструктите за управление и умението те да бъдат правилно формулирани и съчетавани в програмите по принцип. При това не се спираме в подробности на синтаксиса на който и да е език за програмиране, като предполагаме, че поне един е познат на читателя.

2.1 Понятие за Тип. Стандартни типове

Да припомним, че в алгоритъма се оперира точно определен краен брой елементи, образуващи Средата. Всеки елемент трябва задължително да има *име*, което да не съвпада с името на нито един друг елемент.

Нека се върнем към парадигмата “кутия-съдържание” и си представим елемента като променлива-кутия, която има име. Да припомним и RAM-машината, битовите и машинните думи. За да може да се работи със съдържанието на една променлива, трябва предварително да бъде “съобщено” по какво правило е кодирана стойността на съдържанието, записано в машинните думи, както и кои операции могат да се прилагат над кода и как. Това води до въвеждането на едно основно понятие в езиците за програмиране, а именно – **тип на променливата**. Образно казано, типът “съобщава” на

Агента по какво правило е кодирано съдържанието в поредица от битове и как именно се извършват действията, позволени над съдържание от този тип.



Нека, в съответствие с изложеното горе, всяка променлива е от определен тип. Тя би могла да приема различни стойности – толкова, колкото позволяват съответният начин на кодиране и представянето в паметта. Стойностите, които би могъл да приема един елемент от Средата дефинират множество от възможните, “разрешени” съдържания за елемента. Именно това множество е типът на елемента.

От гледна точка на алгоритъм-програма, възможната стойност на всеки елемент от Средата може да бъде *само* от множеството на разрешените за този елемент стойности, т.е. от обявления за тази променлива тип.

➤ **Стандартни типове на променливите**

Информацията, от гледна точка на човека, най-общо се възприема и “обработва” като организирана мислено в различни категории. Съществува компютърно “представяне” за:

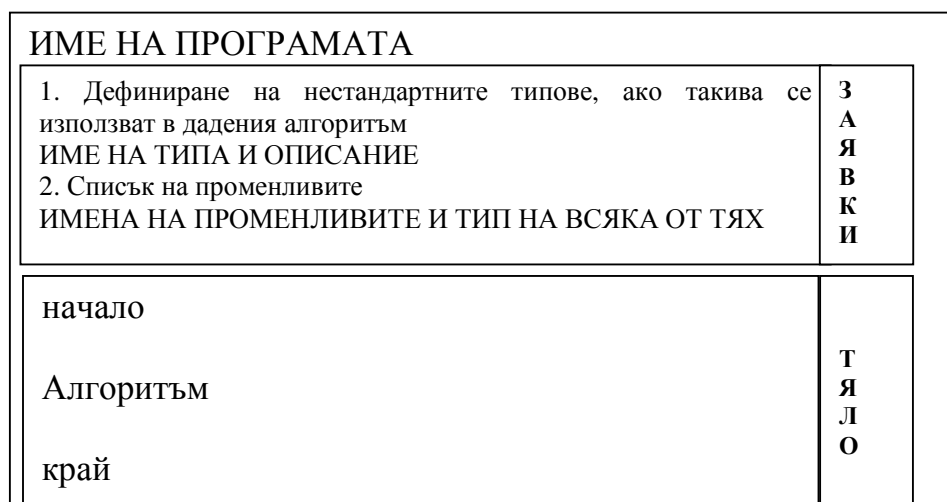
1. Отговор на въпроса “вярно ли е, че...?”, изразяван с Да и Не.
2. Количество, изразявано с число.
3. Знакове и поредици от знакове (печатни символи).

Изброените горе основни категории могат математически да се разглеждат като множества с операции над тях, а за елементите им да се състави машинен образ в двоичен код, като операциите се алгоритмизират. В повечето езици за програмиране съществуват така наречените “стандартни” или “първични” типове на променливите – машинни образи на разнообразни варианти на посочените множества с операции. Такива типове съществуват със съответни, дефинирани в езика имена, които се използват в програмния текст като имена на “поначало съществуващи типове”. Наименованията на стандартните типове не се различават много в различните езици. В следващата таблица са приведени някои означения на стандартни типове променливи.

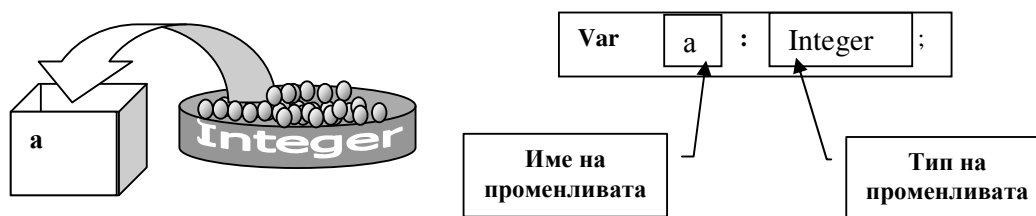
	Pascal	C++
Множество на логическите стойности	Boolean	bool
Множество на целите числа	Integer	int
Множество на рационалните числа	Real	float
Множество на знаковете (печатни символи)	Char	char

Компютърните “множества-образи” съответстват само отчасти на математическите множества-първообрази. На първо място, това е така, защото машинните образи на изброените множества са *крайни*. Единственото множество, което достатъчно добре се имитира, е множеството на отговорите {Да/Не}.

В програмите се посочва кои са елементите на средата за дадения алгоритъм и от какъв тип са те. В езиците за програмиране това става по определени правила. Обикновено това се прави с оператори-декларации, които представляват “заявки за среда”, разположени преди началото на “същинския” програмен текст, който ще наричаме “тяло” на програмата. Долу е илюстрирана такава организация на програмния текст.



Конкретните езикови реализации на “заявки” за среда са подробно описани в съответните справочници за синтаксис на езика. Ще илюстрираме смисъла на заявяване на елемент от средата чрез примерна фраза, в Pascal.



Фразата, илюстрирана горе е: “Да се създаде променлива с име “a”, която е от тип Integer”. Това означава, че:

1) В средата на алгоритъма има елемент с име *a*. Очаква се, че в алгоритъма над съдържанието на този елемент ще се извършват действия и проверки (α).

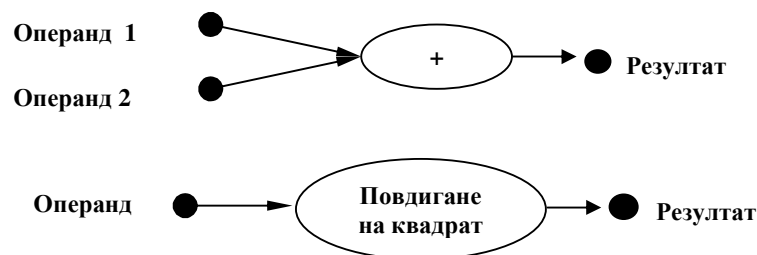
2) Съдържанието (стойността) на тази променлива е от множеството Integer и може да бъде само от това множество.

3) В алгоритъма, с променливата “*a*”, е възможно да бъдат извършвани само действия (или проверки), които са позволени в множеството Integer.

Същото се изказва в C така: `int a`.

2.2 Целочислен тип. Операции с целочислени променливи

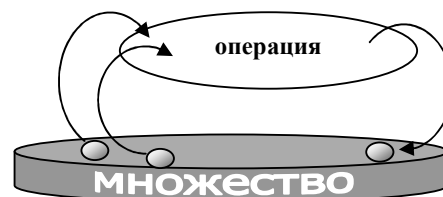
Да изясним общата постановка на това, което тук наричаме “операция”. Операцията се извършва над стойности и води до получаване на единствен резултат, както е илюстрирано долу. Тя се извършва по единствен начин над краен и фиксиран брой операнди (аргументи на операцията), които са вход към преобразованието за получаване стойността резултат.



Операциите биват едноместни, двуместни, триместни и т.н. в зависимост от броя на операндите. В Pascal и C++ за множествата на числата са “вградени” едноместни и двуместни операции, всяка от които е реализация на алгоритъм за съответно изчисление на резултата от получените входни стойности на операндите.

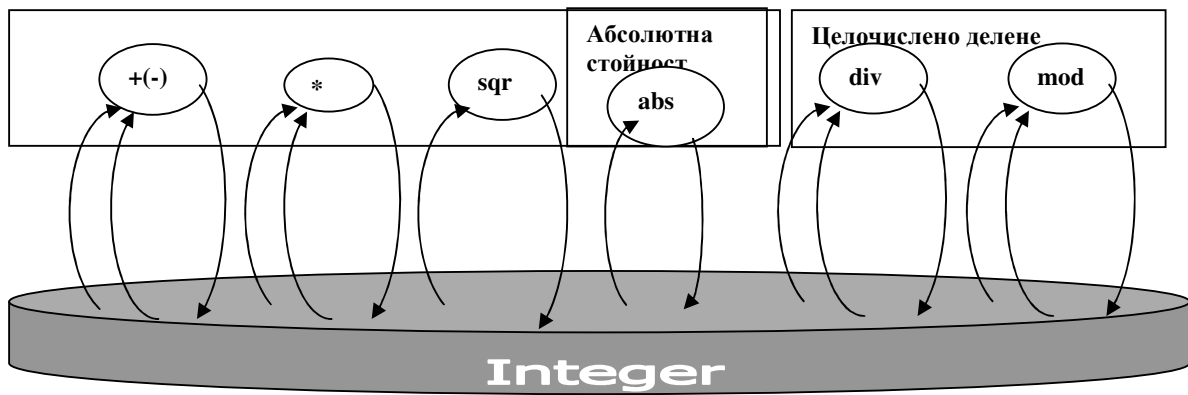
Да разгледаме най-напред *аритметичните операции* в множествата – образи на числата. Някои аритметични операции могат да се прилагат и в двете множества на числата (на целите и на рационалните), а други – не.

Когато всички операнди на една операция са от едно множество и резултатът от операцията принадлежи на същото множество, се казва, че множеството е затворено относно операцията.



Освен заради общата представа за типа като образ на множество с операции, познаването на това относно кои именно операции са затворени множествата на стандартните типове е необходимо в програмирането за правилно съставяне на изрази и при използването на процедури и функции.

Следващата схема илюстрира спрямо кои операции е затворено множеството – машинен образ на множеството на целите числа в неговата “паскалска” интерпретация “Integer”.



Нека a , b и C са имена на променливи от тип “Integer”. Записаните долу програмни инструкции са коректни, защото:

а) изразите от дясната страна на оператора за присвояване на стойност са съставени с използване на операции, които могат да се прилагат над операнди с целочислени стойности;

б) операциите имат целочислен резултат, следователно след като бъде изчислен, той може да бъде записан в променливата C .

$C := a + b;$	събиране
$C := a - b;$	събиране
$C := a * b;$	умножение
$C := \text{sqr}(a);$	(вградена функция) повдигане на квадрат
$C := \text{abs}(a);$	(вградена функция) намиране на абсолютна стойност

Ще се спрем специално на целочисленото делене. То е изразено в Pascal с две двуместни операции – “**div**” и “**mod**”.

Както видяхме, връзката между всеки две цели числа A и B може да бъде изразена по единствен начин при използване на други две цели числа – X и r както следва:

$$A = X \text{ пъти } B + r, \text{ където } r < B.$$

\uparrow

A div B

\uparrow

A mod B

Операциите **div** и **mod** се дефинират както е илюстрирано горе в израза за целочислено делене. Резултатът от:

$A \text{ div } B$	показва колко пъти B се съдържа в A ;
$A \text{ mod } B$	показва какъв е остатъкът.

Нека отбележим, че множеството на целите числа е затворено относно тези две операции. Тяхното математическото наименование е близко до изказа, избран в Pascal, а именно – *делене*⁶ по *модул*⁷. Поради това, ще ги наричаме условно “div” и “mod”, като подчертаем, че за тяхното изразяване има съответни (преки или не) средства във всички програмни езици от категорията,

⁶ Dividere (лат.) → (англ. напр.) divide (v) – дели

⁷ Modulo (мат.) – остатък от целочислено делене

приета като опорна тук. Затова, ще изразим X и r от горния израз за целочислено деление на A и B така:

$$X \leftarrow A \text{ div } B$$

$$r \leftarrow A \text{ mod } B$$

2.3 Примерна програма за алгоритъма на Евклид

Ще приложим препоръчвания тук подход на съставянето на програма, а именно – ще изградим първо схемата на управление. За действията ще използваме само блокове от изпълними оператори, а от конструкции за управление ще “сглобяваме” общото управление.

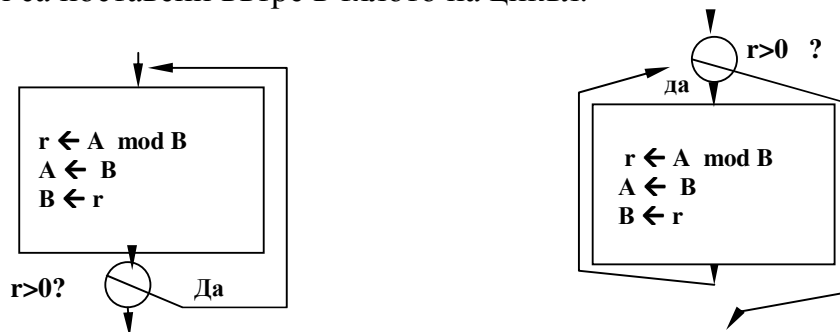
Нека алгоритъмът работи в среда, състояща се от променливите A , B и r , които са от целочислен тип. Да изразим необходимите за алгоритъма на Евклид действия над средата. Както се вижда от разгледаната вече схема на Евклид за промяна на стойностите на променливите, действията в алгоритъма са:

1. $r \leftarrow A \text{ mod } B$
2. $A \leftarrow B$
3. $B \leftarrow r$

Те спират да се повтарят когато се удовлетвори условието $r=0$.

Конструктът, който може да се избере за реализиране на повторението е итеративен цикъл (а не по брояч). Въпросът, който трябва да се реши е дали да се използва цикъл с предусловие или цикъл със следусловие. Да припомним, че в алгоритъма на Евклид действието, което се извършва задължително поне един път е намирането на остатък. То е необходимо, за да се извърши проверката за стойността на остатъка, която предопределя дали алгоритъмът да завърши в този момент от изпълнението.

На следващите схеми към повторението се подхожда “буквално”, т.е. трите действия са поставени вътре в тялото на цикъл.



Да припомним, че при цикъла със следусловие действията от тялото се изпълняват поне един път. Ако изброените горе три действия се оформят като тяло на цикъл със следусловие, пренасяне на стойността на B в A и на r в B ще бъде винаги извършвано преди проверката за край на изчислението.

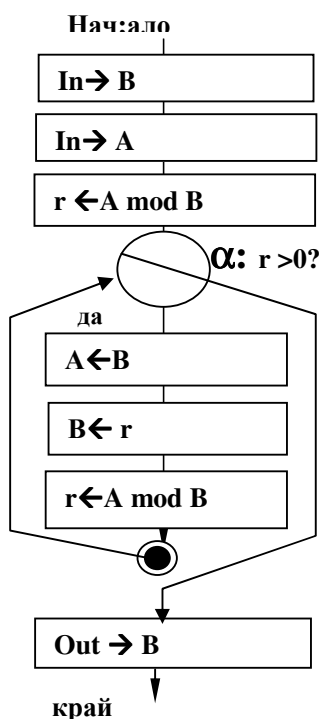
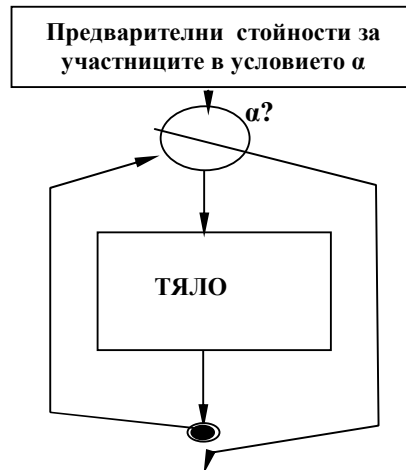
В схемата на цикъла с предусловие пък не е ясно как точно ще се извърши първата проверка за стойността на остатъка r , след като тя се изчислява вътре в тялото.

Алгоритъмът на Евклид може да се реализира и с двете схеми, достатъчно е да се вземат под внимание изложените горе разсъждения. Тук за пример ще

разработим схемата с предусловие, за да обсъдим една характерна особеност на използването на цикли с предусловие.

Както е илюстрирано вдясно, за да функционират коректно, циклите с предусловие трябва да са предшествани от оператори, в които променливите-участници в проверката α “придобиват” стойности, изчислени въз основа на някакви обосновани съображения. Може да се приеме, че схемата на добре съставен цикъл с предусловие изглежда винаги така, както е показано, т.е. задължително включва “подготвителния” блок.

Долу илюстрирана схемата на управление на алгоритъма на Евклид при ползване на цикъл с предусловие. В схемата инструкциите за въвеждане и извеждане на стойности са означени съответно с “In \rightarrow ” и “Out \rightarrow ”. Паралелно на схемата са дадени примерни програмни текстове.



```

Program Euclid;
Var A,B,r : integer;
Begin
  Readln (A);
  Readln (B);
  r := A mod B;
  while r>0 do
    begin
      A :=B;
      B :=r;
      r :=A mod B
    end;
  writeln (B);
end.

```

```

void main ()

{int A, B, r ;
  cin >>A;
  cin >> B;
  r = A % B;
  while (r>0)
    {
      A = B;
      B = r;
      r = A%B;
    }

  cout << B;
}

```

Съставените по илюстрираната схема на управление програми са коректен начален вариант на програмна реализация на Евклидовата процедура. В този си

вид те ще работят именно за намирането на НОД, но ще работят лошо и не винаги, поради което трябва да се допълнят по подходящ начин.

```
Вие работите с програма за намиране на НОД на две
положителни числа, А и В. (А е по-голямо или равно на В)
Моля, задайте А.
148
Ако обичате, задайте В.
30
Момент, търся най-голям общ делител на 148 и 30.
Ето как го получавам:
148=4.30+28
28=1.28+2
28=14.2+0
Най-големият общ делител на зададените от Вас числа е 2.
Довиждане, пак заповядайте да работите с мен.
```

Най-общо, съставена само така, една програма е “недружелюбна”, например тя “не съобщава” какво точно се очаква от потребителя. Въвеждането на стойности за входните променливи на програмата трябва да бъде придружавано винаги със съответни съобщения, “подсещане”, което да ориентира потребителя за това, което се очаква от него. Същото се отнася за изведените на екран съобщения и отговори. За да се превърне от недружелюбна в дружелюбна, програмата трябва да се допълни с “подсещания” и разнообразни съобщения, както е илюстрирано горе.

➤ *Примери за програмна защита на входа*

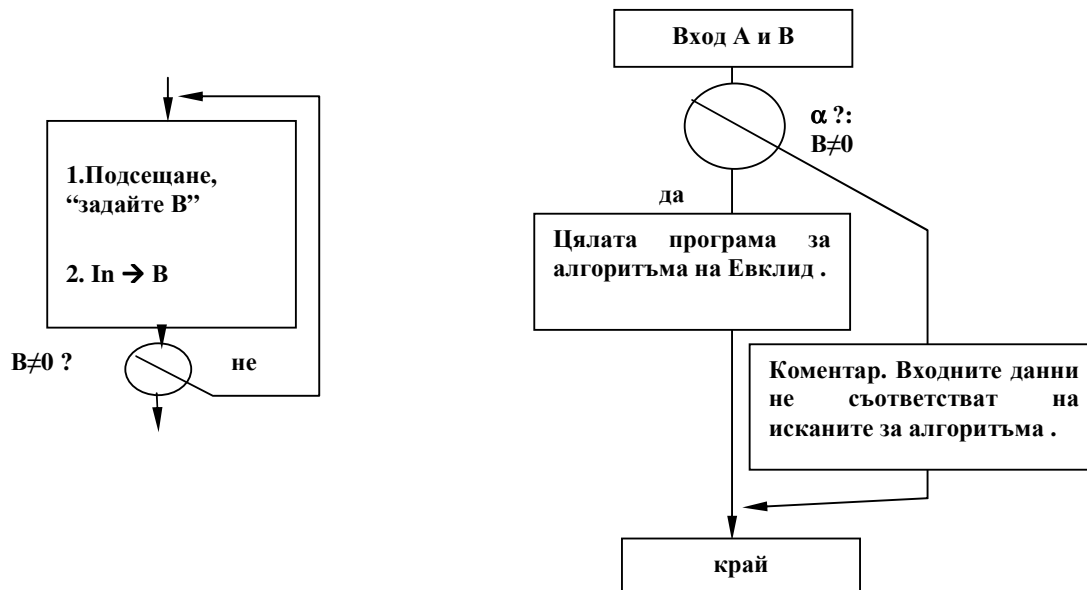
Ще използваме разработената програма за алгоритъма на Евклид, за да приведем някои примери за това как се организира защита на входа към програма на равнището на управлението. Да припомним, че принципно всеки алгоритъм работи с определени допустими стойности на входното множество. Обявяването на тип на променливите представлява някаква начална защита в смисъл, че ако на входа не постъпят стойности от обявения тип, вероятно ще настъпи аварийно прекъсване на изпълнението. Да приемем засега, че такова поведение на програмата е приемливо, макар и не дотам приятно. Тук ще се спрем на случаите, в които множеството от допустими стойности на входа не съвпада със стандартен тип, а е негово подмножество.

Определихме в примерния програмен текст А и В като цели числа в термините на Integer и int, т.е. те могат да бъдат и отрицателни, и нула. Това е допустим за програмата вход, който не съвпада с дефинирания от Евклид вход от естествени числа.

Нека програмата “казва” любезно, че очаква въвеждане на положителни числа и предположим, че потребителят въпреки това въведе за В например 0. В този случай програмата няма да работи коректно.

Да разгледаме няколко примерни схеми на защита.

Илюстрираната по-долу защита посредством цикъл със следусловие ще връща изпълнението към подсещащия коментар дотогава, докато потребителят не зададе число, различно от нула. Схемата, ползваща двуклон, ще “предотврати” изпълнение при некоректно зададена стойност на V .



Съществуват много възможности за защита, например в цикъла може да се вгради брояч и когато стойността му стане 7, да се изведе на екран дефиницията за положително число.

Във всички случаи, добре съставената програма има защитен вход и подходящи подсещащи коментари.

♦ Примерни задачи за упражнение

Да се състави схема на управление и програмен текст за алгоритъма на Евклид при ползване на цикъл със следусловие.

Да се направи анализ на работата на алгоритъма, ако стойностите на A и B са отрицателни. Дали за работата на алгоритъма е необходимо A и B да са положителни или е достатъчно B да е различно от 0? Или е наложително и A да е различно от 0, ако B е отрицателно?

Да се преработи съставената програма като “самодокладваща се”, както е илюстрирано на примерния екран за дружелюбна програма. Това изисква да бъде извеждана на екрана, в развитие, формулата $A = X \cdot B + r$.

2.4 Някои приложения на операциите в множеството на целите числа

Съставянето на алгоритми за аритметични действия в двоична и дори в произволна бройна система се основава на един и същи принцип. Единствената разлика между десетичната и двоичната позиционна бройна система се състои в това, че при едната кодирането на броя (цялото число) се извършва с

множество от десет различни знака (цифри), а при другата – от два. Двете множества се наричат “база” на съответните системи. Ще разгледаме като пример алгоритъма за събиране в двоична бройна система.

Нека припомним най-напред начина на записване на числата чрез вектори в десетична и в двоична бройна система:

<div><div>Дес. БС</div><div><div>вектор</div><div><div><div>5</div><div>4</div><div>3</div><div>2</div><div>1</div><div>0</div></div><div><div><div>0</div><div>1</div><div>3</div><div>8</div><div>6</div><div>5</div></div><div><div><div>↓</div><div>↓</div><div>↓</div><div>↓</div><div>↓</div><div>↓</div></div><div><div><div>10⁵</div><div>10⁴</div><div>10³</div><div>10²</div><div>10¹</div><div>10⁰</div></div></div></div></div></div></div></div>	<div>изразено е следното число:</div> <div><div>0.10⁵ + 1.10⁴ + 3.10³ + 8.10² + 6.10¹ + 5. 1</div><div>0+10 000+3 000+ 800 + 60 +5 =13865</div></div>
<div><div>Дв. БС</div><div><div>вектор</div><div><div><div>5</div><div>4</div><div>3</div><div>2</div><div>1</div><div>0</div></div><div><div><div>0</div><div>1</div><div>0</div><div>1</div><div>1</div><div>0</div></div><div><div><div>↓</div><div>↓</div><div>↓</div><div>↓</div><div>↓</div><div>↓</div></div><div><div><div>2⁵</div><div>2⁴</div><div>2³</div><div>2²</div><div>2¹</div><div>2⁰</div></div></div></div></div></div></div></div>	<div>изразено е следното число:</div> <div><div>0.2⁵ + 1.2⁴ + 0.2³ + 1.2² + 1.2¹ + 0. 1</div><div>0+16+ 0 + 4 + 2 +0 =22</div></div>

Да припомним и правилата за извършване на елементарни аритметични операции, паралелно в двете бройни системи:

0	9
---	---

+

0	1
---	---

1	0
---	---

“пренос” в ДесБС

Получава се единадесето число, за което няма цифра.

0	1
---	---

+

0	1
---	---

1	0
---	---

“пренос” в ДвБС

Получава се трето число, за което няма означение

1	0	0	1	0	1
---	---	---	---	---	---

+

1	0	0	1	1	0
---	---	---	---	---	---

1	0	0	1	0	1	1
---	---	---	---	---	---	---

Събиране в двоична бройна система.

Тези правила се отнасят за произволни позиционни бройни системи. Пренос при събирането се налага, когато няма елемент от множеството от единични знакове на системата (базата), на който да съпостави полученото при събирането число.

На основата на схемата за събиране двоични вектори, нека съставим алгоритъм за извършване на събирането. Да напомним, че възможните “наум” са 0 и 1, а преносът съответства на смяна на състоянието на един бит.

Да започнем с примерен алгоритъм за събиране на два n мерни вектора – a и b . Сумата им се записва във вектора c , който е $(n+1)$ -мерен. “Колоните” при събирането са индексирани с “ k ”, като е спазена номерация в съответствие на начина, по който обичайно събираме, т.е. отдясно наляво. За удобство преносът

е кръстен “*наум*”, а получаваният мислено резултат при поколонното събиране – “*Sum*”.

Вдясно е дадена примерна схема на управление на онзи алгоритъм, за който човек се сеща на първо време, като проследи това, което извършва мислено при събиране в позиционна бройна система.

По правило, когато се съставя схема на управление, тя трябва да се “*проиграе на ръка*” преди да се състави програмният текст, който ѝ съответства. Ето как става проиграването:

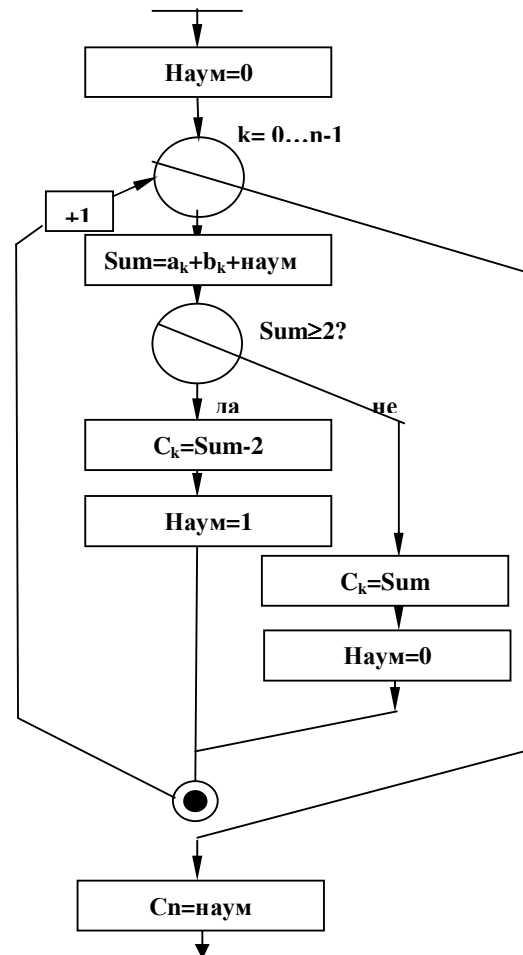
		0	1	1	1	<i>a</i>
		1	1	0	1	<i>b</i>
						<i>Sum</i>
						<i>c</i>
						<i>наум</i>
<i>K=4</i>	3	2	1	0		

Конструира се специална таблица за получаваните на всяка стъпка стойности, както е показано горе. Входът се “поставя” по съответните места в таблицата.

После всяко алгоритмично действие се имитира “на ръка”, стъпка по стъпка, по схемата на управление, като едновременно с това се изчисляват (на ръка) междинните резултати. Те се записват, като се попълва таблицата. Опитайте с този числов пример: вход – $a = 0111$, $b = 1101$.

Да подобрим сега принципа на алгоритъма за събиране на двоични вектори. Да припомним, че пренос се получава, когато получената сума надвишава базата на системата (например, надвишава десет при ДесБС). Това, което се записва в C_k е остатъкът (mod) от целочислено делене на получената мислено сума Sum и базата. Това, което се пренася “наум” е резултатът от прилагане на операцията div на Sum и базата.

Следващата схема на управление е съставена на този принцип. Проследяването на алгоритъма от схемата с примерни данни и попълване на таблиците долу е полезно не само за проверка на коректното изчисление, което той извършва, а и за опитно установяване на принципа на събиране в позиционна бройна система.

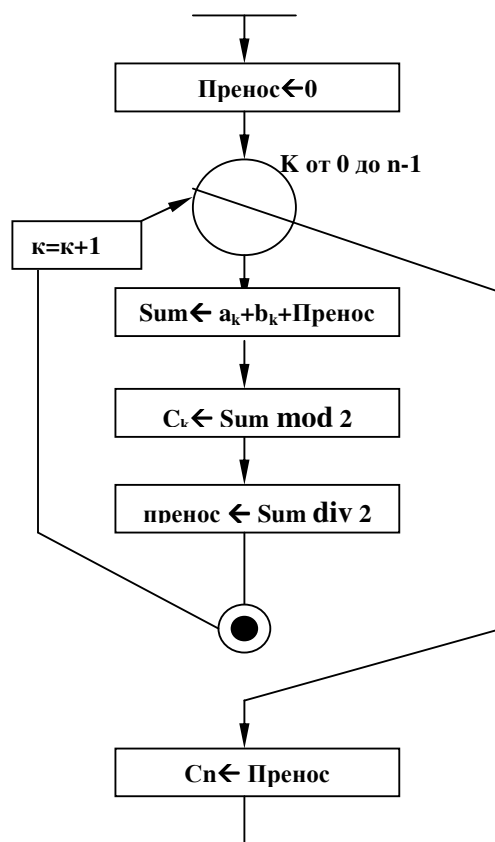


Пример 1 – ДвБС, точно по схемата

					<i>a</i>
					<i>b</i>
					<i>Sum</i>
					<i>c</i>
					<i>пренос</i>
<i>K=4</i>	3	2	1	0	

Пример 2 – ДесБС, (“mod 10” и “div 10”)

					<i>a</i>
					<i>b</i>
					<i>Sum</i>
					<i>c</i>
					<i>пренос</i>
<i>K=4</i>	3	2	1	0	



Този пример дава допълнителна представа за операцията целочислено деление и по-конкретно на ролята на операциите “колко пъти” (div) и “с остатък” (mod) в бройните системи. На този принцип може да бъде съставен общ алгоритъм за събиране в произволна q -ична позиционна бройна система. (q е основа на системата, $q \in \{2, 3, 4, \dots\}$)

2.5 Модел на множеството на целите числа в машината

Кодирането на едно естествено число става като числото се представи с двоичен вектор и се запише над определен фиксиран брой битове. Какъв именно е този брой зависи от системата, езика, версията му и т.н. Целите отрицателните числа се кодират също като двоични вектори. Съществуват различни методи за кодиране на цяло отрицателно число. Най-общо, всички те изискват да се съобрази симетрията (на събирането) относно *нулата*, т.е. изискват кодирането да е такова, че $A + (-A) = 0$. Независимо от метода на изобразяване на “симетричността”, *знакът* (образно казано) изисква още един допълнителен бит. Това е така, защото се изисква кодиране на такова количество отрицателни числа, каквото е това на кодираните положителни числа, т.е., налага се с кода да се “изобразят” два пъти повече обекти. Ако се анализира мощността на двоичните вектори, ще се види, че това изисква кодирането да става над вектор с един бит “по-дълъг”. Би могло да се каже, че над n бита се кодира така: стойността на числото заема $n-1$ бита, а един бит е (образно казано) за знак.



➤ **“Препълване”.**

Този феномен се изразява с факта, че при добавяне на единица към най-голямото положително цяло число, което може да бъде изобразено с кодовата система, се *получава отрицателно число*. (Това се получава при някои от съществуващите системи за кодиране.)

На базата на една примерна схема за кодиране, да се спрем на начина, по който става “препълване”. Разгледаната тук примерна система за кодиране се използва, без да е единствената. Според нея първият бит е запазен за знака и ако в него е записана единица, числото е отрицателно. В останалите $n-1$ бита е кодирана самата стойност.

Вдясно е илюстриран пример за това как би изглеждало кодирано числото минус две чрез двоичен вектор, от които първият е за знака (в примера е взет такъв от четири бита).

1	1	1	0
---	---	---	---

$$-1.2^3 + 1.2^2 + 1.2^1 + 0.1 = -2$$

Да проследим как става събирането при такава система на кодиране. Пример: за удобство при схематичното изобразяване, кодирано е над общо 4 бита, от които 3 са за числото:

The diagram shows two methods for adding 2 and -2. On the left, a number line from -5 to 5 is shown. A blue arrow starts at 0 and points right to 2. A red arrow starts at 2 and points left to 0. The final position is at 0. On the right, a table represents the same operation:

Знак	Число				
	0	0	1	0	2
+	1	1	1	0	-2
=	0	0	0	0	0

Below the table, the equation $2 + (-2) = 0$ is written.

Нека над $n-1$ бита се кодира най-голямото положително число (следователно кодът се състои от нула в първа позиция и останалите са само единици). Добавянето на едно води до появата на единица и в първия бит. Но това “означава” отрицателно число. Да проследим следващата схема:

Знак	Число				
	0	1	1	1	7
+	0	0	0	1	1
=	1	0	0	0	-8

$-1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 1 = -8$
 $7 + 1 = -8$

Този пример илюстрира защо понякога в някои системи, в процеса на нарастването си, една по идея положителна променлива може да стане изведнъж отрицателна.

Да представим едно обобщено сравнение между типовете-машинни изображения и техния първообраз, т.е. множеството на целите числа:

Множеството на целите числа	Множествата – машинни образи на цели числа (типовете)
Безкрайно	Крайни
Изброимо	Крайни
Наредено	Наредени

2.6 Множество на реалните числа. Пример с НОМ на отсечки

Множеството на реалните числа и операциите в него⁸ са обичайно изучавани в гимназиалния курс на обучение и нямат нужда от представяне. По-същественото тук е да се подчертае, че това множество има такива свойства, че за него не може да бъде съставен адекватен машинен образ. За да илюстрираме за какво става въпрос, ще се спрем на един вариант на алгоритъма на Евклид.

Нека процедурата на Евклид се прилага не над цели, а над реални числа и нека действието целочисленото деление е “обикновено” деление. Ще покажем, че това води до риск процедурата да се превърне в преобразование, което не може да бъде квалифицирано като алгоритъм, защото няма никаква гаранция за крайност.

И така, нека A и B са положителни реални числа, а X е цяло число. Тогава съществува познатото представяне:

$$A = X \text{ пъти } B + r, \quad r < B,$$

в което r е реално число. Това представяне е единствено за всеки $A \in \mathbb{R}^+$, $B \in \mathbb{R}^+$. Приложена над такова представяне, Евклидовата процедура съвпада със задачата за намиране на най-голяма обща мярка (НОМ) на две отсечки A и B , като на всяка отсечка се съпоставя реално положително число – дължината ѝ.

Ще приведем рекурсивна дефиниция на НОМ на две отсечки:

Най-голямата обща мярка на две отсечки – A и B ($A \geq B$), е отсечка, която е:

или B , ако B се нанася цяло число пъти в A

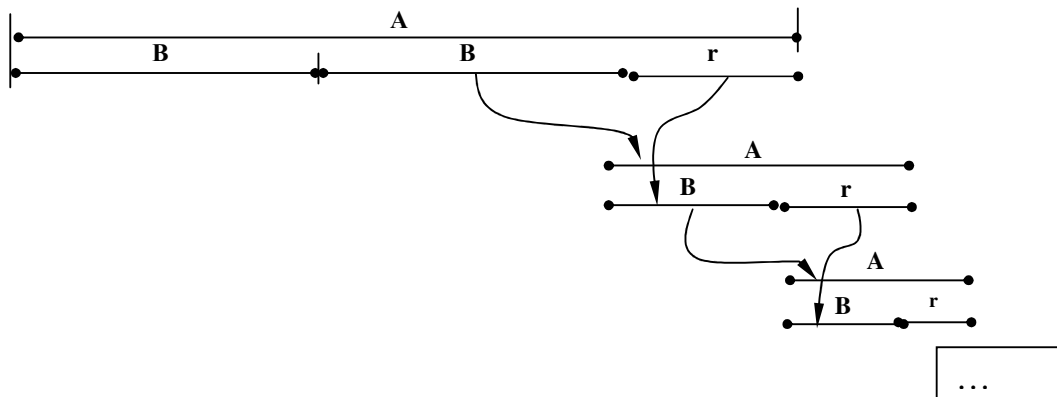
или е отсечката, която е

Най-голямата обща мярка на две отсечки – B и r ,

където r е отсечката-остатък от нанасянето на B върху A .

⁸ Множеството на реалните числа, с операциите събиране и умножение, образуват алгебричната структура $(\mathbb{R}, +, \cdot)$, наречена *поле на реалните числа*. Нейните свойства се различават съществено от тези на множеството на целите числа с операциите събиране и умножение.

Долу са показани няколко стъпки на последователните преобразувания над отсечките при търсене на тяхната НОМ.



Както се вижда от илюстрираните преобразувания за последователно получаване на отсечки-остатъци, тази схема е като алгоритъма на “Евклид”, но за реални числа. Както също веднага се вижда, процесът на делене и получаване на остатъци може за някои входни отсечки A и B да се окаже безкраен, за разлика от Евклидовия с целите числа. В случай, че процесът на делене и получаване на остатъци е безкраен, казва се, че двете входни отсечки нямат най-голяма обща мярка.

Ясен е паралелът между много, много малка отсечка и много, много малко реално число. Те са в някакъв смисъл “едно и също”. Фактът, че не съществува най-малка отсечка (реално число) теоретично обрича на неуспех всички замисли описаната процедура да се превърне в алгоритъм, защото тя просто няма гаранция за край.

За да бъде съставен алгоритъм, който да “работи” в множеството на реалните числа, трябва да се видоизмени по приемлив начин получаването на НОМ. Разбира се, тогава алгоритъмът не е за НОМ, а за нещо друго, различно от НОМ, макар и някакво негово “приближено”. Може например процесът да бъде прекратен след N пъти получаване на остатък различен от нула (N е голямо), като се приеме, че в този случай процесът е безкраен, т.е. входните отсечки нямат НОМ. Тогава, алгоритъмът има винаги край, но изходът “тези отсечки нямат НОМ” не съответства винаги на истината.

2.7 Типът “плаваща запетая”. Преходи между числовите типове

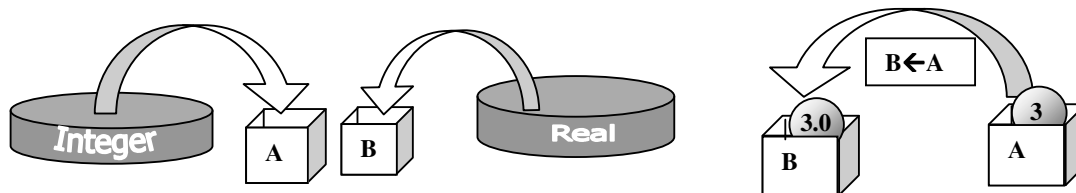
В съзнанието на потребителите, претенция да моделират в машината множеството на реалните числа имат типове като Real, float и техните подобни, Действително, не само наименованията, които са избрани за този базов тип, но и операциите с променливи от такъв тип пораждат “усещане” за работа с реални числа. Излишно е да изреждаме всички операции, позволени за променливите от този тип. Такива са например “деление”, “коренуване”, “логаритмуване”, тригонометричните функции и т.н., спрямо които типът-множество е затворен. При това някак се подразбира, че аритметични операции като “събиране” и “умножение”, приложени над числови операнди, поне един от които е “реално число”, получават резултат от същия тип.

Да обърнем внимание, че в машината образът на множеството на целите числа *не е подмножество* на множеството Real (float). Тези две множества са представени напълно независимо едно от друго и по различен начин. За нас например $3.6 / 1.8 = 2$ и 2 е цяло число. За машината този резултат е “реален”, например: 2.000000001 или 1.99999999 Тя оперира с него, извършва операции над него като с “реално”.

Да видим как да се преобрази една цяла стойност в реална и обратно, като се има предвид, че машината “мисли” за тях като за като за съвършено различни неща. Ще се позовем отново на парадигмата “кутия-съдържание”, като за множествата на числата ще използваме наименованията на английски, които впрочем са използвани за имена на типовете в Pascal.

➤ Преходи между числовите типове

Нека две променливи са съответно A от целочислен тип и B от реален тип. Нека в алгоритъма се налага от известно място нататък да се работи със стойността, получена в A както с реално число. Достатъчно е да се запише стойността на A в B и да се работи с получената в B стойност на A като с реално число.



Интерес представлява пренасяне от Real към Integer. За да бъде извършено това пренасяне на стойността от единия тип в другия, над нея трябва първо да се извърши някакво преобразование. Това се прави по два основни начина – “изрязване”, т.е. пренасяне само на цялата част и “закръгление”, т.е. намиране на най-близкото цяло число. На следващата илюстрация са показани тези преноси, използвани са английските съкращения Trunc и Round⁹, както се наричат съответните функции в Pascal.



Същите преобразувания на стойностите в C биха се изразили съответно с операторите $a = \text{floor}(b)$; и $a = \text{ceil}(b)$;

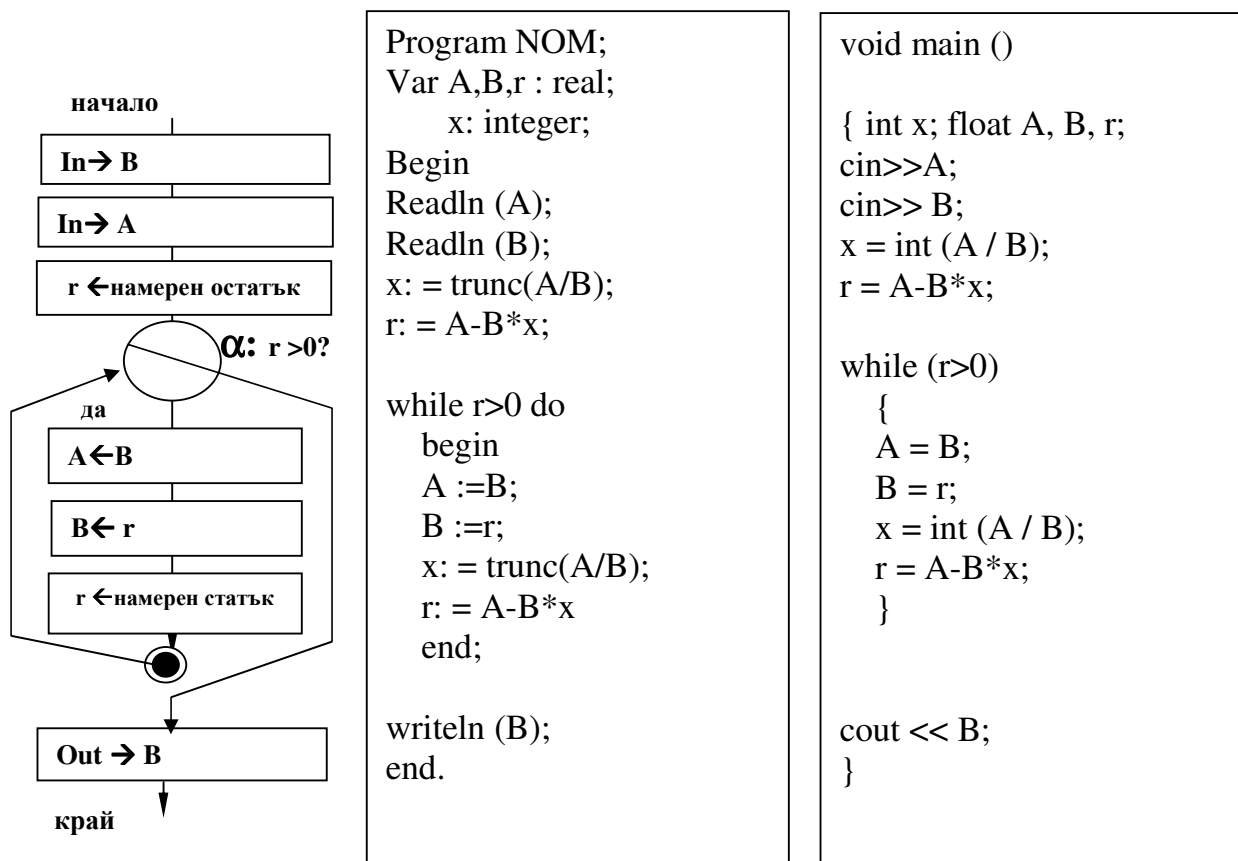
⁹ truncate (v)– изрязвам върха, взимам долна цяла част, round (v)– – закръглявам.

➤ Сравнения на стойности от различни числови типове

В множествата на числата, въпросите α от типа “е ли” ($<$, $>$, $=$ и т. н.) са разрешени за стойности на изрази от два различни числови типа. Например, нека a е целочислена променлива, а b е от тип “реална”. Въпросът:

$\alpha ? : a > b$ е разрешен.

Нека засега се ограничим с представата за Real като за машинен образ на множеството на реалните числа. Да се концентрираме над “префасонирането” на алгоритъма на Евклид в нещо, работещо с реални числа, за което искаме да е също алгоритъм, т.е. искаме то да е с гаранция за крайност. И така, работим в “среда” от реални стойности, и търсим НОМ на две отсечки.



И така, теоретично, задачата за намиране на НОМ на две отсечки (в множеството R), която е аналог на алгоритъма на Евклид, може да доведе до безкраен процес на получаване на все по-малка и по-малка отсечка (реално число). Горее е илюстрирана схема на управление с итеративен цикъл с предусловие, с оглед на съответствието ѝ със схемата за алгоритъма на Евклид от предходната тема. Иначе казано, “пренасяме” същия алгоритъм, но в друга среда – такава от “реални” променливи, формално, без да имаме гаранции за свойствата на алгоритъма в тази среда.

2.8 Итерация

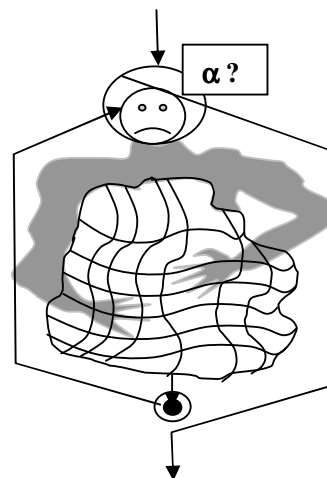
Много съществуващи процеси могат да се представят като повтарящо се изменение, водещо до постигане на определен резултат. Такива процеси се наричат итеративни и се моделират програмно чрез итеративен цикъл. Итеративният цикъл реализира повтарящи изменения, чийто брой не е предварително известен. Итерацията е широко разпространен подход в алгоритмите.

Например, намирането на НОМ на две отсечки може да се представи като итеративен процес. Той, обаче, понякога има край, понякога – не. Не всички итеративни процеси, които се представят с алгоритъм, са с гаранция за край. Тогава се налага да се въведе някакво допълнително правило от рода на: допустима грешка, брой повторения, обратна връзка и т.н.

Тъй като итеративните процеси не са винаги крайни, към тяхното моделиране трябва да се подхожда със съответни предпазни мерки. Нататък в изложението ще разсъждаваме по принцип над въпросите за “защита от зацикляне”. Ще разгледаме “зациклянето” от две гледни точки – зацикляне на програма по невнимание, и “циклене”, породено от естеството на самия моделиран процес.

Зацикляне на програма по невнимание

За да улесним разбирането на това какво представлява един итеративен процес, нека си представим итеративния цикъл като активно въздействие, което се повтаря над средата на алгоритъма (цялата или част от нея) дотогава, докато тя не добие “свойството” α , което всъщност е условието за прекратяване на итеративния цикъл. Всяко преминаване на алгоритъма през тялото на цикъла “се случва”, защото средата все още не е придобила свойството α . Илюстрацията вдясно, въпреки карикатурния си аспект, може да послужи успешно за опора при изясняването на феномена “зацикляне по вина на програмиста”.



Итеративните процеси често се моделират така, че при съставянето на самото условие α за прекратяване на цикъла е неизбежно прилагането на критерии от типа на : “удовлетворен съм от това състояние на средата, макар и да знам, че това не е точното решение”. От гледна точка на алгоритъма, тези критерии за допустима грешка следят за това, дали средата е “достатъчно добре” обработена и цикълът може вече да се прекрати, а задачата – да се приеме за решена. В този случай се говори за “точност на решението”, а това, че средата е “достатъчно” добре обработена от цикъла означава, че приемаме състоянието ѝ за удовлетворително с оглед изискванията на поставената задача. Този похват се използва много често в числените методи, а понятие за точност на решението ще изградим в следващите теми.

От краткото изложение за моделирането на итеративни процеси следва, че е трудно е да се даде някаква рецепта за това как те да бъдат моделирани

“правилно”. Един начин да се подсказе как принципно се достига до някакъв резултат е да се посочи какво не трябва да се прави. Ще наречем това, което не трябва да се прави с комичното название “достатъчни условия за зацикляне”.

➤ **Достатъчни условия за зацикляне, или още – сигурни методи за зацикляне на програма**

1. Достатъчно е условието α за край на цикъла да не се отнася до обработваната в тялото среда. Например:

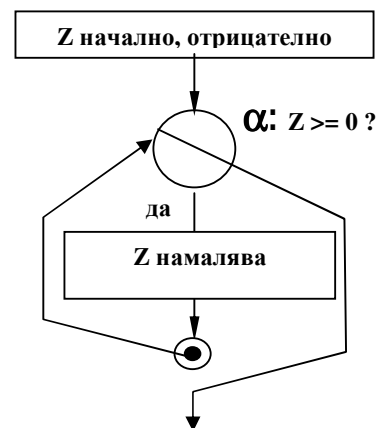
“Търси остатъка $r = A \bmod B$ и размествай A , B и r дотогава, докато сумата на D и F стане 7895218.”

Колкото и елементарен да изглежда този сигурен метод за зацикляне, той често е причина изпълнението на програми да се прекратява посредством електрическото захранване. Ще посочим и друг, по-фин метод на зацикляне:

2. Достатъчно е условието α , въпреки че се отнася до “обработваната” в тялото среда, да е съставено с гаранция, че никога няма да се изпълни.

Нека например в тялото на цикъла една отрицателна променлива Z намалява с всяко преминаване през цикъла. Достатъчно условието за край на цикъла да следи кога въпросната променлива ще стане най-после положителна.

Въпреки, че този втори метод е “по-тънък”, причината за циклене все пак лесно би се разпознала в програмния текст. Затова ще приведем един още по-ловък метод за зацикляне, наречен тук “Прескочи-кобила”.

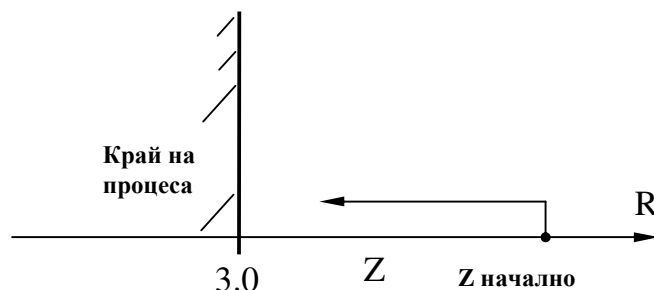
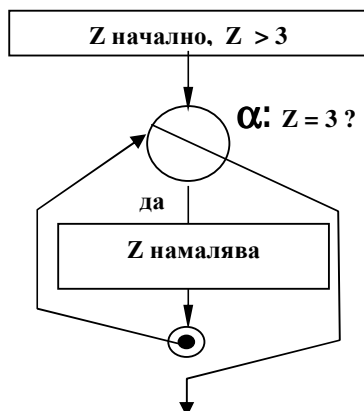


3. Прескочи-кобила

Със следващия пример ще илюстрираме принципната постановка на метода. Нека в тялото на цикъла една променлива Z намалява с всяко преминаване през цикъла. Например така :

$Z \leftarrow Z - 2 * a$, където Z е “реално”, а a е “реално” и положително.

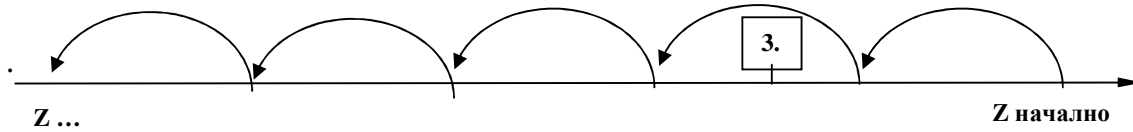
Нека естеството на процеса, който се моделира с този цикъл, е такова, че повторението трябва да спре, когато Z премине границата 3.00.



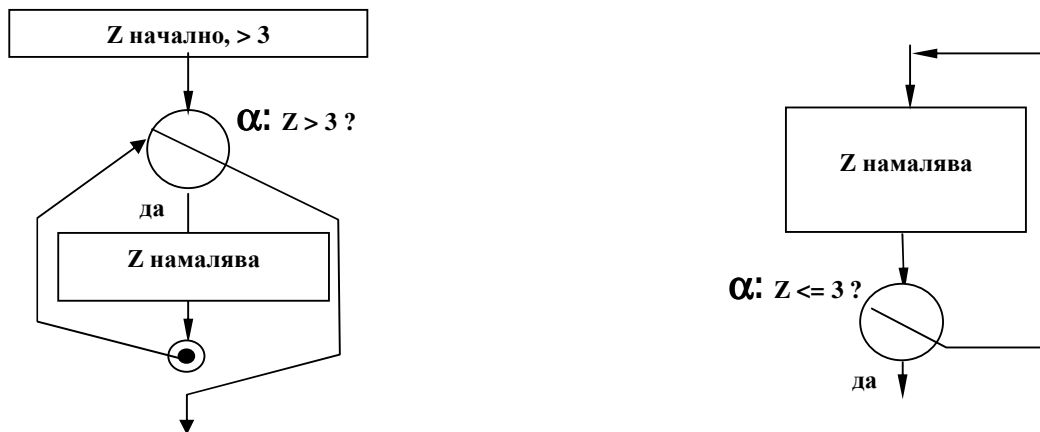
В тази примерна ситуация изглежда, че “Z намалява и все ще премине през тройката!” Условието:

α : $Z=3.00$?

може да се приеме за напълно обосновано. Именно то реализира игра на прескочи-кобила. Ето схемата на изменение на Z:



Съществуват и други ефективни методи да се зацikli програма. Засега ще се ограничим с тези три, от които най-сигурен е този с прескачането на кобилата. Той е в такава степен “мощен”, че понякога дори и опитни програмисти го прилагат по невнимание. Затова ще дадем един чисто практически съвет: Никога не поставяйте в условието α изискване за стриктно равенство, без да имате за това някаква много специална и добре обмислена причина. Ако няма такава, условието α за край на цикъла трябва да е съставено с : $<$, $>$, \leq , \geq .



Можем да обобщим изложеното за методите на зацикляне с едно изречение: “Няма нищо по-лесно от това да се зацikli една програма поради некоректно условие за край на итеративния цикъл.”

Много често α е “комплексен” въпрос, който не се отнася само до едно елементарно “свойство” на средата. Както вече нееднократно подчертахме, колкото и да е “сложен” въпросът α , той има само два отговора – Да или Не. Затова, преди да се запознаем с начините за “самозащита на програмиста” от безкрайни итеративни процеси, нека припомним и някои от правилата, по които се формулират и “по-сложни” въпроси α .

2.9 Логически тип. Логически операции

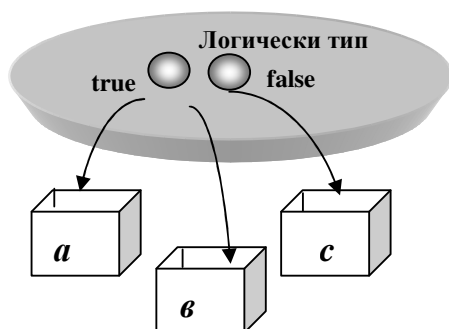
Математическото множество на логическите стойности има само два елемента – Истина (Да) и Неистина (Не). То се представя по много естествен

начин в компютър, където поначало информацията е представена при използване на бинарен принцип. Един елемент от това множество е представен просто с един бит.

Долу е приведена Таблица, която може да се види навсякъде. Дадени са две логически променливи *a* и *b*. В Таблицата са посочени стойностите на логическите изрази за *a* и *b*, образувани с логическите операции И, Или и Не:

Не <i>a</i>	Дадено		<i>a</i> И <i>b</i>	<i>a</i> Или <i>b</i>
	<i>a</i>	<i>b</i>		
False	True	True	True	True
False	True	False	False	True
True	False	True	False	True
True	False	False	False	False

Нека *a* и *b* са “елементарни” въпроси, всеки от които има един от двата възможни отговора. Следователно, можем да кажем, че *a* и *b* са логически променливи, чиято стойност може да бъде едно от двете: Да или Не (*true* или *false*). Според парадигмата кутия-съдържание, “заявката за елементи” на средата на алгоритъма изглежда така, както е показано на илюстрацията долу.

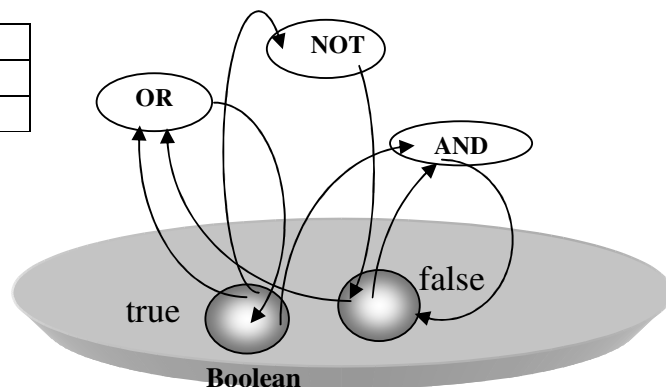


Такава заявка за променливи от логически (булев¹⁰) тип означава, че:

- 1) се създават променливи с име *a*, *b*, *c*, които са от логически тип;
- 2) “съдържанието” им може да бъде *само* от множеството на логическите стойности, т.е. *true* или *false*;
- 3) над тези променливи могат да се извършват само действия, “разрешени” в множеството на логическите стойности.

Операциите, относно които множеството на логическите (булевите) стойности е затворено, са:

Не	Not	Не (операнд1)
И	And	(операнд1) И (операнд2)
Или	Or	(операнд1) Или (операнд2)



¹⁰ На името на Дж. Бул (George Boole), английски математик и логик от XIX век, поставил основите на математическата логика.

Операндите на логическите операции са логически стойности. Долу са приведени примери за изрази, които биха могли да бъдат записани за променливите от булев тип a , b и c :

$c \leftarrow \text{true}$
 $c \leftarrow \text{Not } a$
 $c \leftarrow a \text{ And } b$
 $c \leftarrow a \text{ Or } b$

Ето и пример за по-сложен логически израз:

$c \leftarrow \text{Not}((b \text{ And } a) \text{ Or } (d \text{ And } g))$

където d и g са също от булев тип.

Каква стойност ще получи белевата променлива c при зададени стойности на променливите в израза от дясната страна, по-неопитните преценяват с помощта на дадената в началото Таблица. Затова ще дадем една методическа опора за Таблицата.

Нека въпросите a и b са дефинирани както следва:

$a?$	$a = \text{да}$	Означава да е отворен прозорецът
Отворен ли е прозорецът	$a = \text{не}$	Означава да е затворен прозорецът
$b?$	$b = \text{да}$	Означава да е отворена вратата
Отворена ли е вратата?	$b = \text{не}$	Означава да е затворена вратата

По следващата примерна таблица, резултатите от прилагане на логическите операции биха могли да бъдат възпроизведени, без да се помнят наизуст:

a : отворен ли е прозорецът	b : отворена ли е вратата?	$c=a$ и b (и a и b едновременно) става ли течение?	$d=a$ или b (поне едното) проветрява ли се?
True	True		
True	False		
False	True		
False	False		
a	b	$a \text{ AND } b$	$a \text{ OR } b$

Условията, които означаваме тук с “ $\alpha?$ ”, са логически стойности, изчислявани по посочените дотук правила. (Елементарните сравнения от типа “Е ли равно” имат логически стойности.) Формулирането на условия “ $\alpha?$ ” изисква съставяне на логически изрази (с операциите And, Or и Not) в които операнди са елементарни сравнения, логически променливи, или и двете. Например, нека a е логическа променлива, а x и y са променливи от числов тип. Тогава условието $\alpha?$ би могло да се формулира например така:

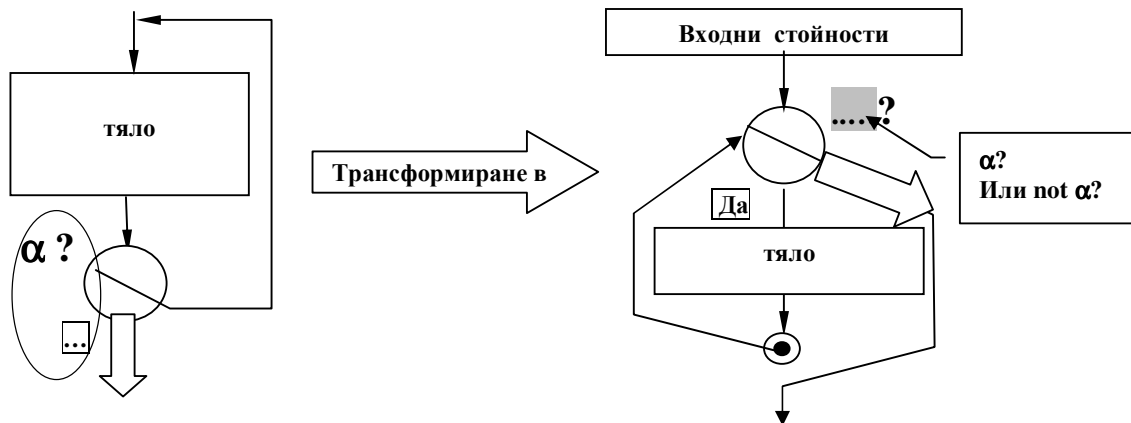
$\alpha? : (\text{Not } a) \text{ Or } (x > y)$

Тъй като, както споменахме, програмите циклят често именно заради неправилно съставяне на условието α , програмистите научават Таблицата наизуст.

➤ **Някои правила, свързани с итеративните цикли**

Понякога се налага един итеративен процес, моделиран посредством цикъл със следусловие, да се представи чрез цикъл с предусловие, или обратно. Наложително при такова преработване да се внимава за променливите в тялото, преди тялото и т. н. Съществува и една важна особеност, която по принцип трябва да се има предвид, въпреки че се отнася до конкретни езикови реализации на фразите за цикли.

При разглеждането на циклите с предусловие и следусловие, представихме разликата в езиковите формулировки на конструктите в термините на “Щом като” и “Докато”. Всъщност, ставаше въпрос за това в какъв случай се излиза от цикъла – при изпълнено или при неизпълнено условие $\alpha?$ от програмния текст. Ако “пренасянето” на цикъл с предусловие към такъв със следусловие (или обратно) става при използване на фраза, конструирана “обратно” на фразата от изходния вариант на цикъла, налага се и смяна на записаното условие – от $\alpha?$ в $\text{not } \alpha?$.



Ако се проследят предходните схеми на управление от гледна точка на това “кога свършва итерацията”, може лесно да се прецени дали при смяната на използвания конструкт, записът на условието $\alpha?$ трябва да се “обърне” или не. В Pascal например това е наложително, защото от цикъла със следусловие се излиза при $\alpha?$ – Да, а от този с предусловие се излиза при $\alpha?$ – Не. Помощният въпрос “кога се излиза от цикъла” дава добър ориентир за това дали при промяната на “формата” на цикъла трябва и условието $\alpha?$ да бъде обърнато от $\alpha?$ в $\text{not } \alpha?$.

2.10 Предпазване на итеративен цикъл от зацикляне

➤ **Всяко зацикляне на програма е по вина на информатика – програмист**

В предходната точка за “зациклянето” показвахме как може да бъде зациклен итеративен цикъл от небрежност. Тук ще разгледаме някои похвати, които трябва да се прилагат като израз на специално внимание. Разглеждаме случая, в който е възможно да се получи “циклене” на програма не поради невнимание, а защото такъв е моделираният процес. Такава ситуация се наблюдава много по-

често, отколкото ни се иска. Най-общо, тя се изразява с това: моделираният процес е такъв, че за някои входни данни итеративният цикъл е краен, а за други – не. Неприятността идва от това, че често не е предварително известно за кои входни данни процесът свършва (е сходящ) и за кои – не. Както казахме, пример за такъв процес е намирането на НОМ на две отсечки. В такива случаи програмата трябва специално да се “защити”.

➤ *Аварийен изход от тялото на цикъл*

Нека вътре в тялото на цикъла има поставен един “аварийен изход” – логическо условие β , което, ако е изпълнено, цикълът се прекратява “насилствено”.

Може да се прекрати направо изпълнението на програмата, както е например при следното използване на оператора Halt в Pascal:

If β then Halt;

Към такава схема със спиране на изпълнението на програмата не е добре да се прибегва, освен при тестване на програмата. Използването на подобно спиране, образно казано, означава, че програмата е ... халтава.

“Аварийното” излизане от цикъла (не по α , а по β) може да се “насочи” и към следващия изпълним оператор след цикъла с goto, exit, break, и други, в зависимост от езика, версията и т.н.

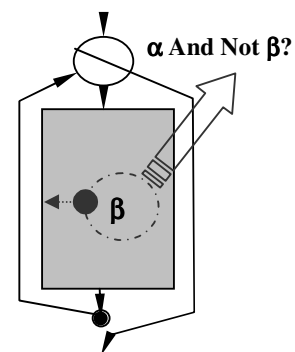
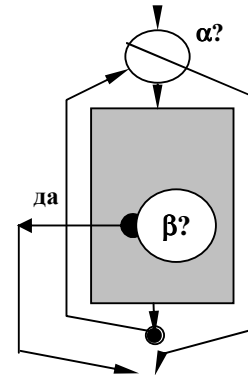
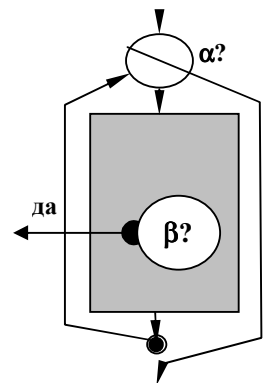
Наличието на “насилствено” прекратяване на цикъл е обичайно признак за това, че при тестването на програмата с различни входни данни програмистът е бил силно затруднен и се опитва “пътем” да подобри нещата.

Независимо от конкретната ситуация, в тези случаи е препоръчително тялото на цикъла да се направи “самодокладващо се”, за да може да се проследи какво “се случва” с участниците в α , такова, че програмата понякога цикли. Добре е по принцип да се избягват “насилствените” прекратявания на цикли.

В езиците, за които няма специално предвидени фрази от типа на exit и break в C++, не се препоръчва прекратяването на цикъла с използване на безусловен преход.

Принципно това става, като условието за “аварийно” прекратяване се постави на “полагащото му се място”, а именно при условието за край на цикъла, както е илюстрирано вдясно. На схемата вдясно :

α е условието за “нормален” вход/изход от цикъла, следващо от самия моделиран процес (например при НОМ $\alpha: r > 0$?)



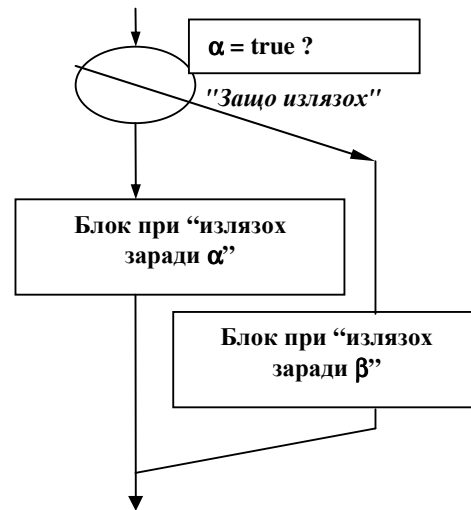
β е аварийното условие за прекратяване на цикъла, което програмистът е изградил въз основа на някакви съображения.

При спазване на горната схема на съчетаване на двете условия в едно, програмата е конструирана “по правилата” с вградените в езика фрази за управление. Така тя е защитена от грешки по небрежност и по-лесно се проследява или модифицира.

В тази “комбинирана” схема, условието за край на итерацията е такова, че изпълнението на цикъла ще спре в следните два случая:

1. Когато итеративният процес спира, защото “така си е редно” и
2. Когато цикълът е спрял “аварийно”.

При прилагане на такава схема, след изхода от цикъла трябва да се прави проверка със смисъл “защо излязох от цикъла?”. Схемата на такова управление е показана вдясно.



Във всеки от двата клона на показания горе двуклон се поставя (поотделно) това, което трябва да направи алгоритъмът, ако итеративният цикъл е завършил “нормално” и това, което трябва да направи, ако не завършил нормално.

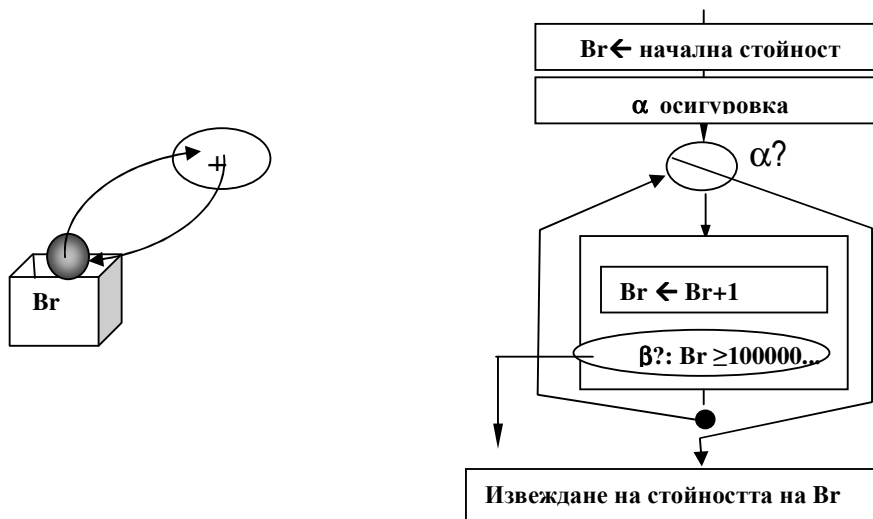
Схемата “защо излязох” се прилага не само при аварийни изходи.

➤ Съставяне на условията за “аварийно” прекратяване на цикъла

Тук можем да изложим само общи съображения и методи за съставяне на условия за аварийен изход от цикъл. Най-простата методика е да се брои.

А. Броячи на програмиста

Броячите на програмиста са нещо подобно на “датчици”, поставени на избрани от програмиста места в схемата на управление.



Целта е да се разбере колко пъти изпълнението на алгоритъма преминава през даден клон на схемата. За да се реализира това, достатъчно е в средата на алгоритъма да се въведе променлива-бройч и да се увеличава нейното съдържание с единица при всяко преминаване през дадения клон. На схемата е илюстрирано използването на брояч в тялото на итеративен цикъл.

Резултатите от това преброяване могат да служат както следва:

- 1) Ако α се изпълни, броячът показва след колко итерации това е станало.
- 2) Ако α не се изпълни много пъти (все не се, и не се изпълнява!), прекратяването на цикъла може да стане по съставения от програмиста аварийен изход (условие β), в който се следи за стойността на брояча. Следователно, в β се проверява колко пъти алгоритъмът е преминал през тялото на цикъла и ако този брой стане “обезпокоителен”, изпълнението на цикъла се спира аварийно. Например:

β : $Br \geq 10\,000\,000\,000\,000$?

Ами ако α щеше да се изпълни на $10\,000\,000\,000\,001$ - вия път? Много рядко броят преминавания през тялото на цикъла е смислен източник на информация за това, дали процесът е краен (цикълът ще спре “нормално” при тези входни данни) или не. Прилагането на такава схема с преброяване е непрепоръчителна. Все пак, тя е един вариант на защита от продължително изпълнение на цикъл.

Б. По-разумни допълнителни условия β
Добре би било, ако, както е илюстрирано вдясно, условието β за “аварийно” излизане от цикъла се отнася до средата, обработвана в тялото на цикъла. Почти винаги е възможно да се открият тези елементи от средата, по стойностите на които може да се съди за това дали при изпълненията с различни набори от входни данни има опасност процесът да е безкраен (разходящ).



Когато се казва, че “цикълът върви към ...циклене”, се има предвид, че той не довежда средата до състояние, което предварително е определено като “приемливо”, т. е. резултатът от процеса далеч от решението на задачата. Тогава това се извежда на екрана подходящ коментар, например: “При тези входни данни вашият алгоритъм не работи, както ви се иска.”. Това е напълно допустимо.

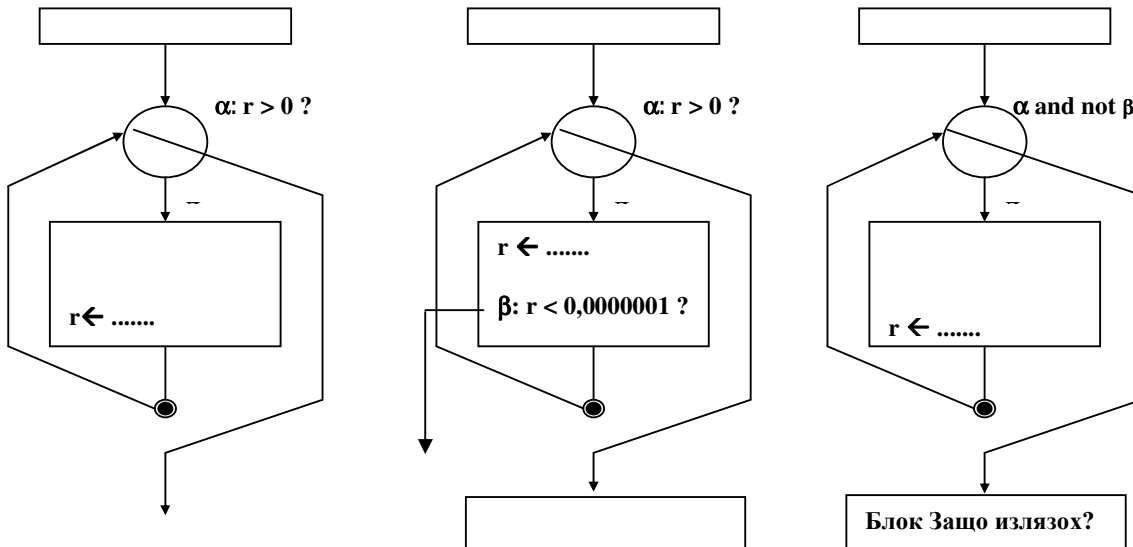
Търсенето на НОМ може да се алгоритмизира по схемата “точност на решението”, в него не може да се следи по никакъв начин дали процесът ще завърши при конкретни входни данни. При НОД интуитивно приехме, че “ограничението отдолу” е в най-лошия случай най-малкото естествено число, т.е. 1. Нещо такова би се получило и с НОМ, ако приемем, че най-малката отсечка-остатък, с която въобще ще се занимаваме, е $0,0000001$. При резултати, по-малките от този “граничен” остатък ще приемаме, че двете входни отсечки нямат смислена НОМ. Все едно приемаме, че най-малката отсечка е $0,0000001$,

знаейки, че това ни най-малко не е вярно. Следователно, с цел превръщане на процеса в краен, достатъчно е да формулираме:

$\alpha: r > 0 ?$

$\beta: r < 0,0000001 ?$

Схемите долу показват различни варианти на реализация на итеративното намиране на НОМ.

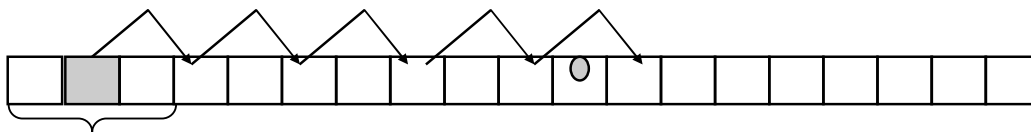


В последната, която е най-добра, ако при изпълнение “излизането” е заради β , т. е. r не е станало 0, може да се приеме едно от двете:

- НОМ на двете отсечки е коя да е отсечка с дължина по-малка от 0.0000001;
- Двете отсечки нямат НОМ.

♦ Примерна задача за упражнение

Игра на “скрит предмет”



Нека следната игра се играе от двама играчи – Играч 1 и Играч 2:

Играч 1 “скрива предмет” в една от стаите на безкраен коридор, но без първата, втората или третата. Стаите са номерирани с естествените числа.

Играч 2 “търси” предмета в стаите, като “прескача врати” и има право да “влиза” с определена стъпка, избрана от него в началото на играта. Стъпката не може да е 1. Играч 2 тръгва от първа, от втора или от трета стая.

Играта завършва така:

- ако Играч 2 попадне на предмета (намери го) – Печели.
- ако го пропусне – Губи.

Да се състави алгоритъм и програма за имитиране на тази игра. Играта се играе от двама души, на компютър. Играч 1 задава “скритото” естествено число x . Играч 2 избира стъпка на проверяваните номера – $step$ и начално число (стая): 1, 2 или 3.

Задачата е да се имитира играта, а не да се състави формула за пресмятане на резултата.

Препоръки.

Използвайте цикли и защиты: В началото разсъждавайте така: Приемете, че условието α е дали играч 2 е намерил предмета.

Съставете схема с “авариен изход” по β : “прескочено” е x .

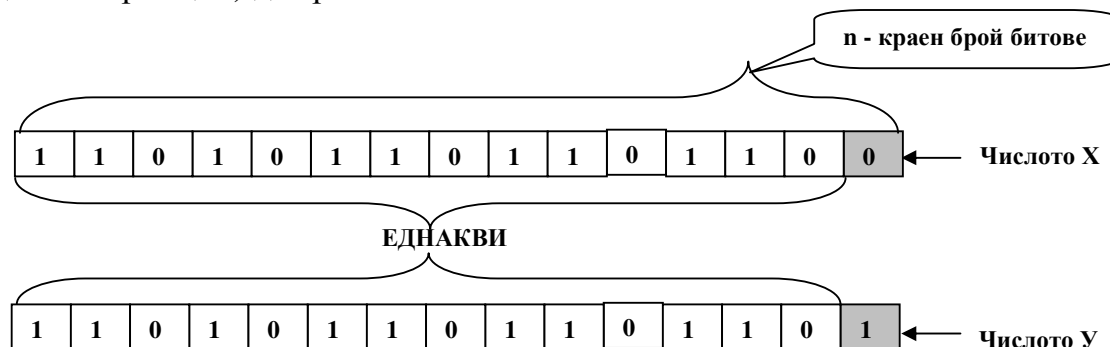
Съставете схема с обобщено условие за изход от цикъла, в което участват α и β . Анализирайте полученото условие за изход от цикъла. Използвате ли “е ли равно”? Оформете по подходящ начин управлението на “защо излязох от цикъла”, така, че програмата да уведомява Играч 2 дали е спечелил.

2.11 Типът “Плаваща запетая” и представянето на рационални числа

Видяхме някои особености на множеството на реалните числа при разглеждането на примера за намиране на НОМ на отсечки. Без да се навлиза в подробности от математиката, важно е да се подчертае едно свойство на това множество, а именно, че в произволен интервал от множеството на реалните числа, съществуват безкрайно много негови елементи. (Дори интервалът да е много, много малък...).

При програмно реализиране на предложения алгоритъм за намиране на НОМ може да се установи, че дори да не е защитена от безкраен итеративен процес, програмата винаги завършва (при произволни допустими входни данни). Фактически с програмата не се реализира процес, съответстващ на този в множеството на реалните числа.

Нека разгледаме един напълно необвързан с конкретна реализация пример, който цели само да накара читателя да се замисли над кодирането и реалните числа. На какъвто и принцип да е кодирано едно число, това става над краен брой битове. Да предположим, че в една система над n бита са кодирани (по еднакъв принцип) две реални числа – X и Y .



Нека, както е илюстрирано, двата кода – този на числото X и този на числото Y , са такива, че всички битове с изключение на n -тия са еднакви. В множеството на реалните числа в интервала $[X, Y]$ “между” X и Y има безбройно много реални числа. Очевидно, всички те няма как да бъдат представени със съответен им код, съставен на същия принцип.

Най-общо, реалните числа не могат да се представят по удобен за машинна обработка начин. Например, реалното число “корен от две” няма дори крайно означение в позиционна бройна система. Следователно, то не може да бъде представено с краен запис в паметта на машината.

Да помислим тогава за представянето на рационалните числа. Ако те се представят например така:

(числител – цяло число) / (знаменател – цяло число, различно от нула),

това би довело до сравнително точно представяне на самите рационални числа (в рамките на представянето на целите, което, както видяхме, не е съвсем перфектно). Но как би изглеждал тогава алгоритъмът за събиране на две рационални числа например? Дали наистина си заслужава за всяко събиране да се извършва привеждане под общ знаменател?

Това са въпроси, които са решени по някакъв сравнително приемлив начин, но не трябва да се забравя, че те са сложни, че за тяхното решаване има различни подходи и че нито един от тях не е съвършен.

В машината няма тип, съответстващ на множеството на реалните числа. Съществуващите типове `Real`, `float` и техните подобни, са образ на рационални числа. При това този машинен образ е далеч от съвършенството.

➤ *Едно правило при сравнения*

Въпросите от вида “Е ли $\boxed{\dots}$ равно на $\boxed{\dots}$?” се задават за аргументи от множествата на числата. Изложеното за машинното представяне на рационални числа на практика води до необходимостта от спазване на следното програмистко правило:

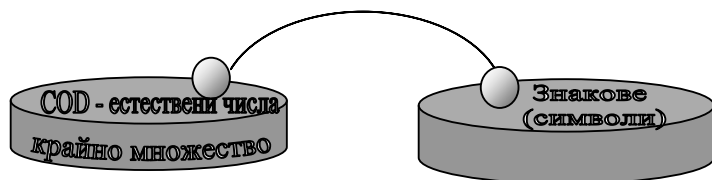
Променливи от тип “плаваща запетая” не трябва да се сравняват с “е ли равно”!

Дори и теоретично да е сигурно, че двете променливи се получават с равни стойности, възможно е два еднакви резултата или резултат и рационална константа да са представени като две различни стойности – и едната с грешка и другата с грешка, според възможностите за точност на представянето. Това би довело един цикъл например до вариант на разгледаната игра на “прескочи кобила”. За да се избегне това, използва се сравняване на *разликата* между двата аргумента с много малко число (“малко” по обоснована преценка). Ето пример:

$$x = y \quad ? \qquad \text{се превръща в} \qquad \text{abs}(x-y) \leq 0.000001 \quad ?$$

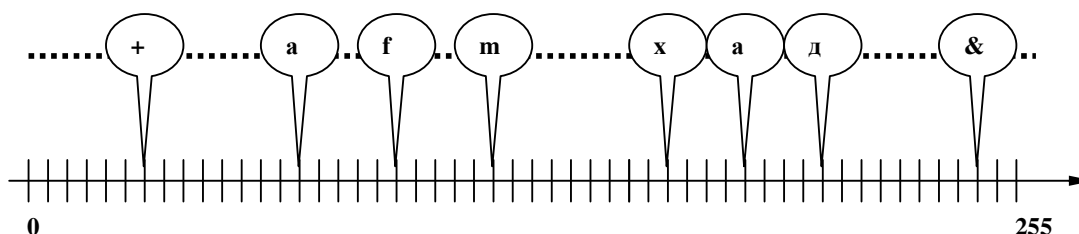
2.12 Типът “Печатен символ”

Ще разгледаме накратко принципа, на който се основава машинното изображение на множеството на знаковете (печатните символи). Разгледахме какъв е принципът, на който са представени целите числа в машината. Всяко от кодираните числа е изобразено по единствен начин, при това този начин позволява над числото да се извършват аритметични операции по правилата въобще, но в друга бройна система. Като се отчете фактът, че в машината достатъчно добре се представят естествени числа, логично изглежда да се направи някакво съответствие между използваните писмени знакове и естествените числа. На илюстрацията долу е показано каква е “задачата”: на едно крайно множество от естествени числа да се съпоставят знакове.



Мощността (броят на елементите) на множеството от естествени числа COD предопределя мощността на множеството от символи, които могат да се свържат биективно с него. По исторически причини, най-често употребяваното множество COD е множеството от целите числа от нула (включително) до 255, или, иначе казано, множеството на осем местните двоични вектори, кодирани с един байт.

И така, съществува биекция (взаимно еднозначно съответствие) между множеството COD и печатните символи.



Самите печатни символи се съхраняват като изображения за визуално възприятие, но всеки си има *номер*. Съответствието (биекцията) “номер-символ” се нарича *кодова таблица*. Долу е показано как принципно изглежда тя:

Кодова таблица		
....	
128	←→	‘А’
129	←→	‘Б’
130	←→	‘В’
...		...

(например)

Редно е да възникне въпросът за това кои именно са включените в тази биекция печатни символи? (Дали сред възможните 256 символа е включено множеството от печатни символи на японски?) Включена е латиницата, десетте цифри, знаците + – . , ; ! ?... Една груба сметка показва, че до 256 има още доста свободни позиции в таблицата.

Най-широко разпространената кодова таблица е ASCII¹¹ (по исторически причини, свързани с развитието на телетайпа като средство за комуникация).

Да дефинираме множеството на печатните символи като тип, според въведеното преди понятие за тип. На схемата долу е илюстрирано значението на примерна заявка за променливите M и N от знаков тип, в парадигмата “кутия-съдържание” :



Тази заявка означава, че две променливи с имена M и N могат да имат съдържание от знаков тип и над тях могат да се извършват само действия, позволени за този тип. Ето няколко примерни оператора:

$M \leftarrow 'a';$

$M \leftarrow '+';$

$M \leftarrow '!';$

Всички действия в типа “Печатен символ” са свързани с биекцията между множеството на знаковете и кодовете. Нека разгледаме примери на Pascal, защото в този език типът “Char” и множеството от кодове са отделени, а биективното съответствие $\text{Char} \leftrightarrow \text{COD}$ се “вижда” с функциите, прилагани над елементите на Char и на COD.

Поредният номер на елемент от Char се намира с функцията:

Ord (аргумент от Char),

Съответстващият на даден код (0 – 255) елемент Char се намира с функцията:

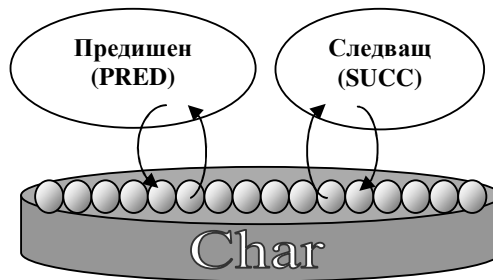
Chr (аргумент от COD),

Например:

Chr (248) е символът ‘ ° ’, а Chr (129) е символът ‘ Б ’ и т.н.

Множеството COD е наредено, следователно множеството Char е също наредено. За всеки знак (освен за първия и за последния) съществуват точно определени “предишен” знак и “следващ” знак.

¹¹ American Standard Code for Information Interchange



Проверката за еднакво съдържание на две знакови променливи всъщност е проверка за еднаквост на кодовете им. Наличието на наредба в Char има значение и при сравненията от вида “>”, “<” и т.н. В много езици е възможно с кода на знака да се работи по подразбиране. Ето пример на Pascal:

If ($M < N$) then ... ,

където M и N са променливи от тип Char.

Може дори с помощта на променливи от знаков тип да се организират цикли по брояч, като всъщност се отброява по кода:

For $M := 'A'$ to $'Z'$ do ...

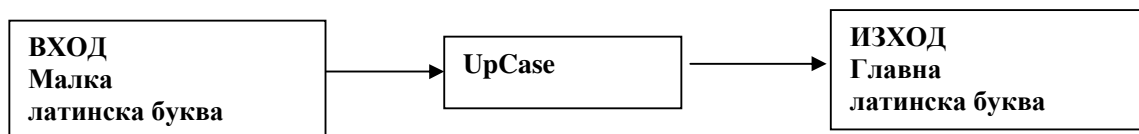
Кодът на “брояча” – променливата M (от тип Char), нараства от кода на A (ord ('A')) до кода на Z (ord ('Z')), вследствие на което записаният в M символ се сменя последователно от A до Z

В езика C, множеството на печатните символи и множеството COD са представени с един обединен стандартен тип **char**, а на променлива от този тип може да се “гледа” и по двата начина: и като печатен символ, и като код – *char*(-128,127) и *unsigned char* (0,255).

Всички особености, свързани с наличието на наредба и с използването на множеството {0,..., 255} за “номериране”, са в сила за всички “ординални” типове в езиците за програмиране. Това са типовете, за които има биекция между елементите им и множеството {0,..., 255}.

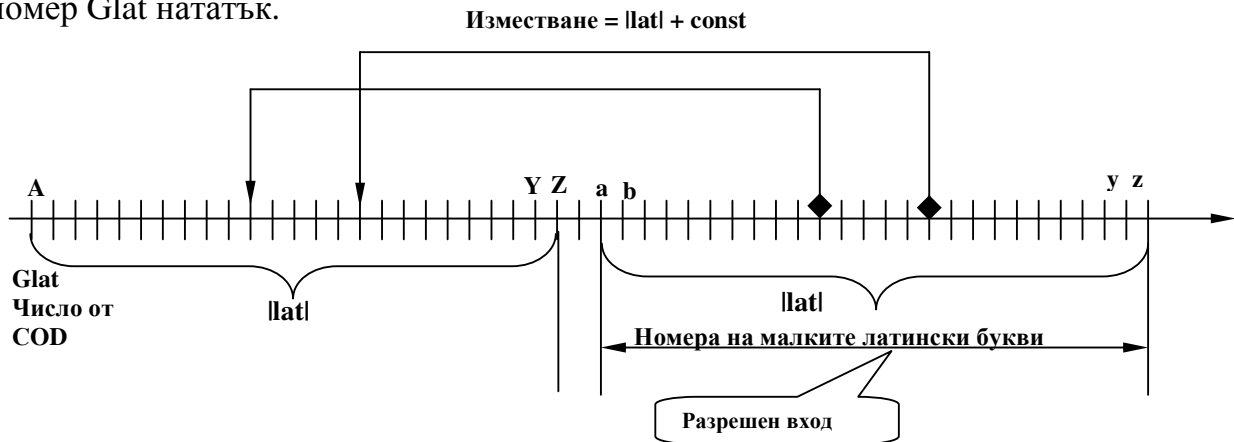
➤ Някои примери за функции над символи

Казаното дотук важи по принцип за всички версии на Pascal, C и сродните им езици. Да разгледаме как работи една нововъведена в някои версии на Pascal функция за работа над Char, която се нарича “UpCase¹²” и “работи” както е илюстрирано на следващата схема.



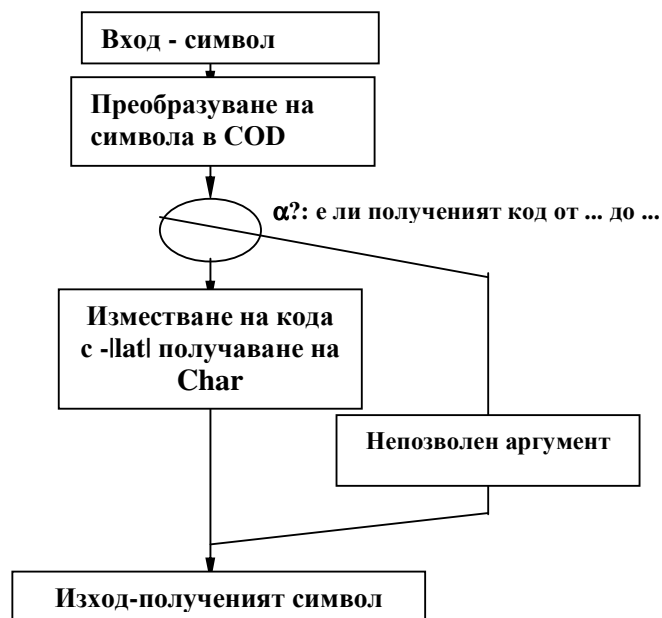
¹² Upper Case означава “главна” буква.

Нека, както е показано долу на схемата за примерно разположение на печатните символи, в кодовата таблица главните латински букви започват от номер Glat нататък.



Нека мощността на латиницата е числото $|\text{lat}|$ ($\text{lat}:\{a, b, c, d, e, f, g, h \dots x, y, z\}$). Нека след края на главните са “залепени” малките, от номер $\text{Glat} + |\text{lat}| + \text{const}$ нататък, т.е. с някакво изместване от определен брой const позиции. Важно е и двете поредици от букви – малките и главните, да са в един и същи ред. В ASCII е прието това да е азбучният ред.

Помислете как да се състави програма по следната схема на управление, реализираща UpCase:



➤ **Едно надграждане над типа “Печатен символ” – тип “Символен низ”**

“Символен низ” е тип, т.е. множеството от разрешени съдържания за една променлива. Един елемент от това множество представлява например:

Променлива S:	A l g o r i t h m i c s
	1 2 3 4 5 6 7 8 9 10 11 12

Всяка стойност на такава променлива представлява наредени (имат свое поредно място) и конкатенирани (залепени един до друг) символи. Символният низ е вектор с елементи от множеството Char. В C променлива от този тип се задава именно така – като едномерен масив, т.е. вектор с елементи – променливи от тип *char*.

Основните операции, които се извършват над променливи от този тип, са:

Проверка за еднаквост на два символни низа;

Проверка за наличие на определен символ в даден низ;

Конкатениране на два низа.

Да дадем примерни оператори за работа с този тип променливи в Pascal, където той съществува като самостоятелен тип с име **String**:

S := 'algorithmics'; – присвояване стойност

length (S) – показва каква е дължината на символния низ (низовата променлива) S, в случая е 12.

S[3] е 'g' – изолира символ от променливата, по номера на мястото му.

Writeln (S); – директна разпечатка на низа.

По подразбиране, дължината на променливите от тип низ е 256 последователни знака. В повечето системи първият знак е “служебно” запазен за броя на заетите от дадената променлива позиции.

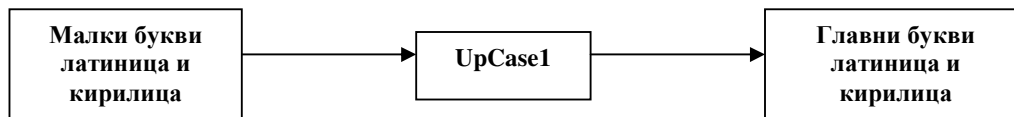
Var S: string “означава” Var S : string [255]

Дължината на String може и да се фиксира така: Var S : string [18];

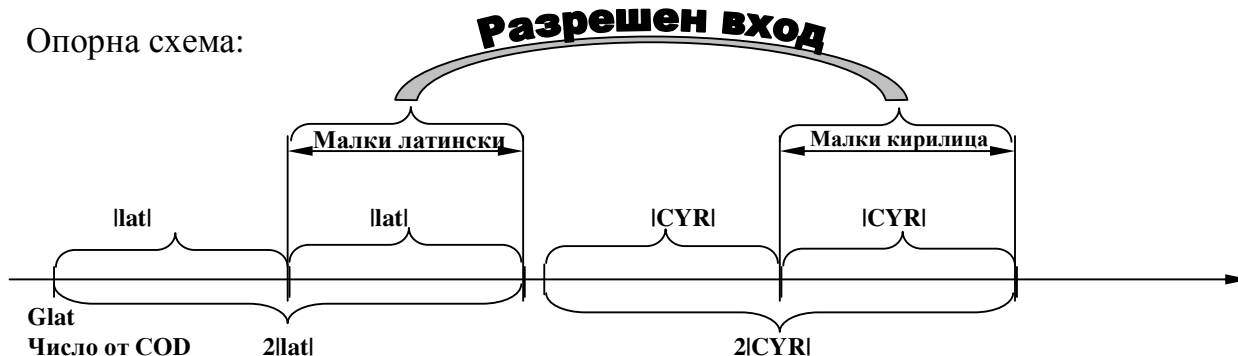
Тогав за променливата се запазват 18 места в паметта, всяко място за по един Char (ако специално не е посочена дължина, се подразбира максималната дължина на този тип, а именно: 255 места).

♦ Примерна задача за упражнение

Да се подготви схема, аналогична на UpCase, която да работи и за кирилицата (“Всесилен” UpCase). Приемете, че малките и главните букви са разположени в кирилицата на същия принцип, на който това е направено за латиницата. Множеството CYR : {а, б, в ... ю, я} има мощност 32:



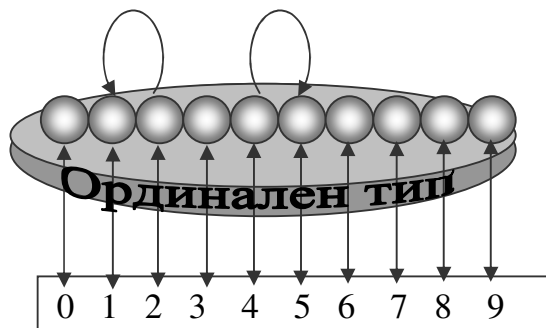
Опорна схема:



В зависимост от това дали е постъпила малка буква от едното “малко” подмножество или от другото, трябва да се направи едно (от двете възможни) измествания наляво.

2.13 Ординални (скаларни) типове

В предходните точки от изложението се запознахме със стандартните типове на променливите, като наблегнахме на това, че типовете са множества с дефинирани в тях операции. Ще се запознаем с едно разширение на стандартните типове, наричано “ординален” тип (*ordinal*¹³ – с номерация на елементите).



Във всяко крайно множество може да се въведе номерация на елементите, т.е. биективно съответствие между елементите на множеството и естествените числа. В машината такава номерация се прави обикновено при “използване” на номера от множеството $\{0, 1, 2 \dots 253, 254, 255\}$. Типове, които съответстват на това представяне, се наричат ординални.

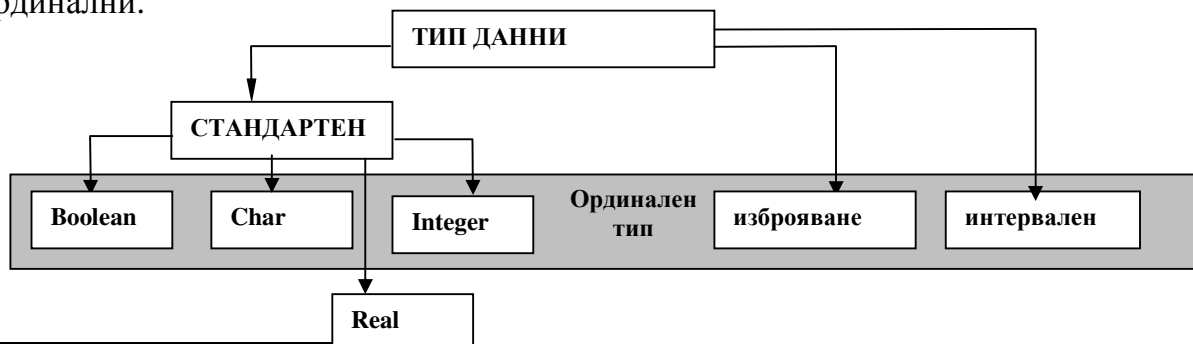
Всеки елемент от ординалния тип има номер. В повечето езици е възможно да се работи с поредния номер на елемента, като той се “извлече” със съответната вградена функция. Например “*Ord (елемент)*” в Pascal.

Всички елементи, освен първия имат предишен: “*Pred (елемент)*”

Всички елементи освен последния имат следващ: “*Succ (елемент)*”

От представените дотук типове с такива свойства са типовете-образи на множеството на целите числа и на множеството на печатните символи.

Например в Pascal, ординални са типовете Char, Integer, Longint, Byte, Word. Множеството Boolean е също наредено, но е толкова маломощно, че този факт не върши никаква съществена работа. Схемата долу е за типове в Pascal, но служи достатъчно добре за общо представяне най-вече на това кои типове не са ординални.



¹³ *ordo* (лат.) – ред

Ординални типове по принцип могат да бъдат “конструирани” при използване на стандартните наредени типове в качество на “градивен материал”. Така се дефинира нов тип, който също е наредено, номерирано множество. Това се прави обичайно по два начина:

- 1) С изброяване на елементите на типа в тяхната последователност.
- 2) С посочване на интервал – подмножество на някой от стандартните ординални типове.

Да припомним, че когато е необходимо програмата да работи с елементи, които не са от стандартен тип, налага се новият тип да бъде “обяснен” предварително (в началото на програмния текст) със специална “заявка за нестандартен тип”. В тази заявка, на нововъдения тип се задава избрано от програмиста име.

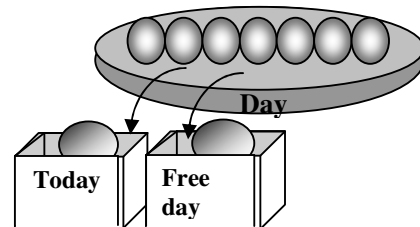
➤ *Конструиране на ординален тип с изброяване на елементите*

В оператора за заявяване на типа, елементите му трябва да бъдат изредени един след друг в реда, в който програмистът иска те да бъдат номерирани. Първият “придобива” автоматично номер нула (това не се забелязва, освен при изпълнение на функцията “покажи поредния номер”). Долу е приведен пример, в който е образуван типът `day`, съдържащ седем елемента, избрани за означаване на дните на седмицата. В примера е използван Pascal, обявени са и две променливи от ново конструирания тип.

```
Type day =(pn, vt, sr, ch, pe, sb, ne);
```

```
Var
```

```
Today , Freeday : day;
```



Според разгледаните вече правила за работа, след обявяването типа и променливите от този тип с променливите могат да се извършват само позволените за този тип операции. Характерна особеност на всички ординални типове е, че в тях са позволени операции, свързани с наредбата на елементите. Долу са посочени примерни оператори за действия над съдържанието на променливите от горния пример. Проследете операторите и обърнете внимание на цикъла по вграден брояч. В Pascal е възможно да се организира такъв цикъл с брояч – променлива от “новообявен” ординален тип. Нарастването на стойността на брояча става автоматично и без проблеми, защото номерата на елементите на типа са еднозначно дефинирани още при заявяването му. Очевидно, необходимо е долната и горната граница на изменение на брояча да са от “неговия” тип.

```
Today := pn; Freeday := ne;
```

```
If succ (today) = freeday then writeln ('приятна почивка!');
```

```
...
```

```
For today:=pn to sb do
```

```
Writeln ('През',today,'се решават задачи.');
```

➤ Конструирание на ординален тип с посочване на интервал

Очевидно един интервал (“верига”) от множество, в което има наредба е също наредено множество. Това е причината в някои езици да е осигурена възможност за обявяване на нов ординален тип като подмножество на друг съществуващ ординален тип посредством посочване на *интервала от...до*.



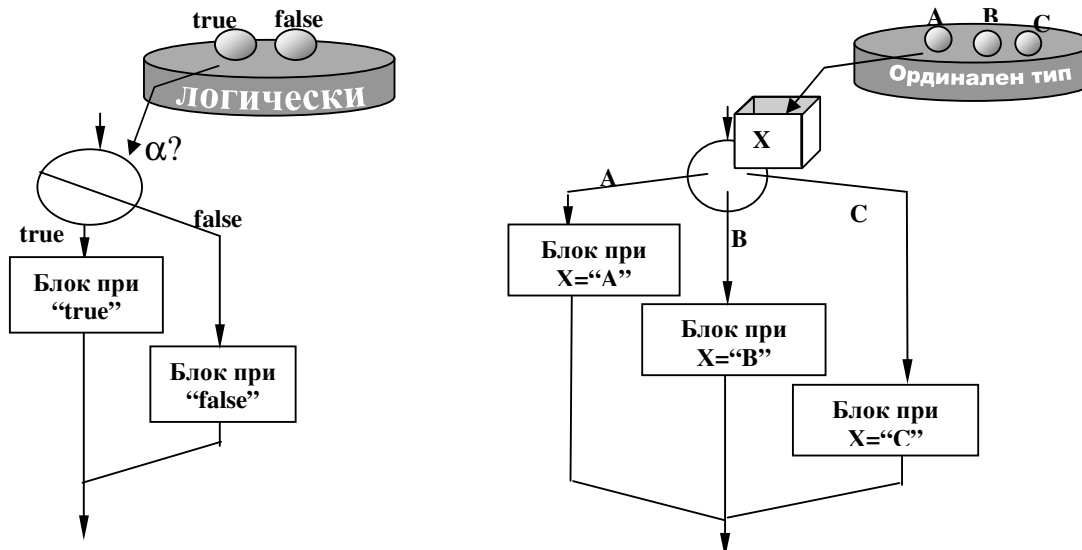
Типът-интервал е “нов” нареден тип, но номерацията на елементите в него се наследява от основния ординален тип.

Тази възможност е полезна и приложима. Например датите в месеца са цели числа, но не бива да заемат стойности по-големи от 31 и по-малки от 1. Ако в дадена програма се работи с променлива, в която се записва дата, то тази променлива е цяло число, над нея могат да се извършват онези операции, които са “позволени” в множеството на целите числа, но стойността ѝ трябва да е само в интервала от 1 до 31. Разумно е за яснота и дори като своеобразна логическа защита, в програмата да се обяви тип “дата” и всички променливи, с които се изразяват дати да бъдат от този тип. Долу е приведен пример за това как се обявява интервален тип и съответни променливи в Pascal:

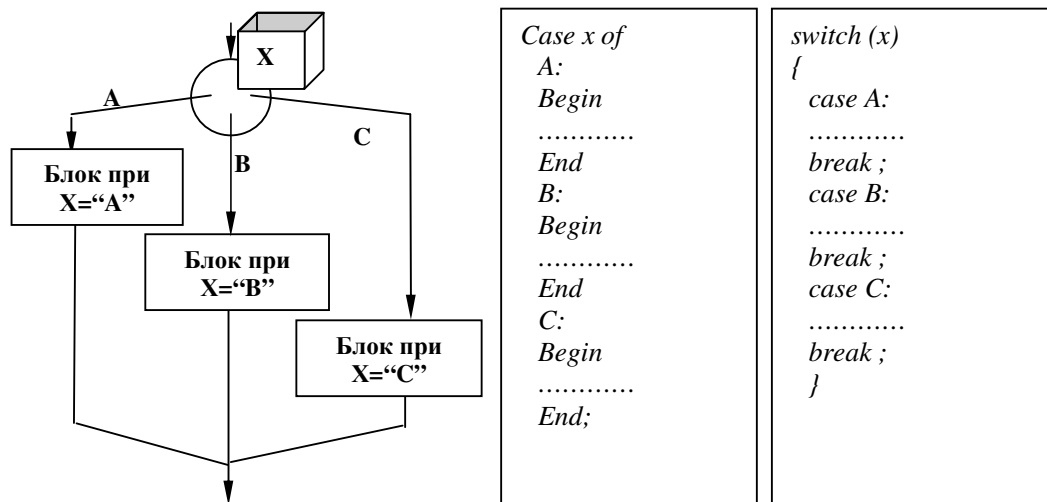
```
Type Date = 1 .. 31 ;
Var
TodayDate, ExamDate : Date ;
```

➤ Конструкт за управление “многоклон”

Това управление “разклонява” на *краен* брой клонове пътя, по който алгоритъмът може да премине. За да улесним разбирането на това защо в настоящото изложение този вид управление се обвързва така непосредствено с ординалните типове, ще направим паралел с конструкта *двуклон*.



При конструкта “двуклон” управлението се ръководи от стойността, която една логическа променлива α (или логически израз) получава (по време на изпълнението на алгоритъма). На всяка от двете възможни стойности на α съответства един клон от по-нататъшното изпълнение на алгоритъма. Да разгледаме многоклона на базата на аналогия с двуклона. Нека приемем, че за всяка една от възможните стойности на *ординална променлива* алгоритъмът трябва да преминава по различни клонове на схемата за управление. Променливата, от стойността на която зависи по кой от клоновете да премине изпълнението, се нарича “селектор”.



В примерната схема, селектор е променливата X . Селекторът е израз от ординален тип (т.е. израз, резултатът от чието изчисляване е от ординален тип). *Етикетите* на разклоненията (A , B , C на примерната схема) са константи (конкретни стойности) от същия ординален тип, от който е селекторът. В Pascal е възможно етикетите да бъдат и списъци от константи (например $(A,B), (A,C)$), интервали на ординалния тип от който е X (например “ $A .. C$ ”) или списъци от такива интервали. В C етикетите могат да бъдат само константи, а за да се получи списък от константи, при които се преминава през един и същи клон, в текста на програмата те трябва да се изброят една под друга без прекъсванията “*break*;”.

Този конструкт за управление е осигурен с готови фрази в повечето използвани понастоящем процедурни езици за програмиране. Той би могъл да се състави от програмиста с използване на повече двуклони.

Използването на многоклон изисква специално внимание, особено при избора на селектор и съставянето на възможните клонове. Ако е трудно възможните стойности на селектора да се “разпределят” във вид на клонове на “поведение” на алгоритъма, т.е. няма предвиден клон за всяка стойност, е препоръчително използването на още един успореден клон (*else*, респ. *default*), с оглед яснотата на начина на преминаване на алгоритъма през многоклоновата структура. При всички положения, когато се използва многоклон, желателно е селекторът да е с добре “обозряна” от програмиста мощност.

2.14 Натрупване на суми и произведения

Да разгледаме най-напред крайни суми от вида:

$$a_1 + a_2 + a_3 + \dots + a_n = \sum_{i=1}^n a_i$$

При това общоприето означение в сила е следното:

- 1) Индексите i са първите n естествени числа : $\{1, 2, 3, \dots, n\}$
- 2) Членовете a_i на сумата се изразяват посредством i , т.е. са функция на поредния си номер.

Горните две разсъждения дават основание изчисляването на крайни суми от числа да става с използване на цикли с вграден брояч i , като се спазват следните правила:

1) В една променлива Sum се добавя всеки нов член a_i на сумата. Типът на променливата Sum е като този на общия член сумата.

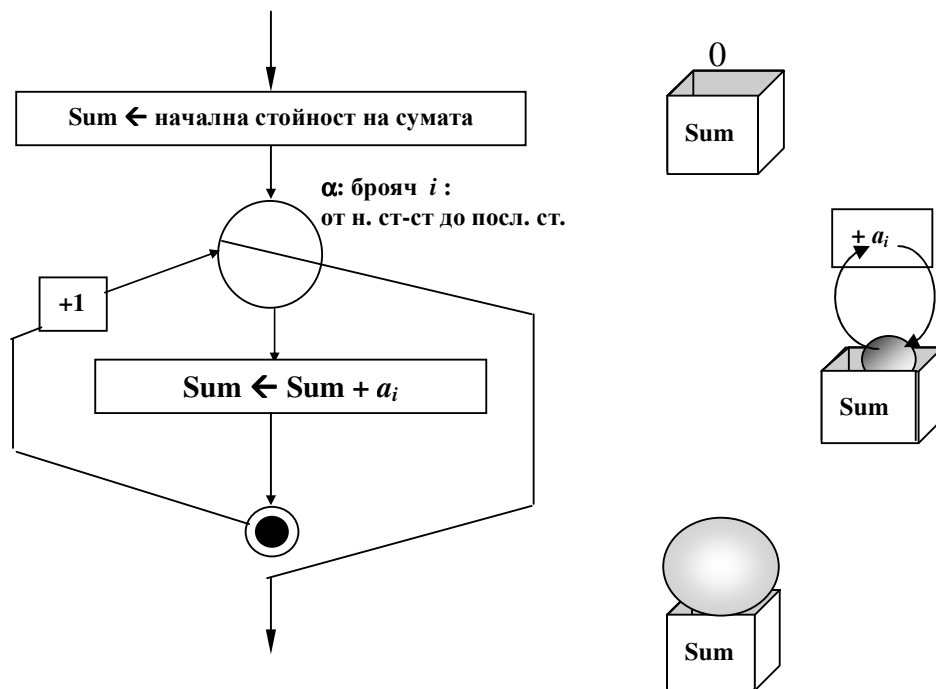
2) В тялото на цикъла с вграден брояч i се пресмята всеки пореден член a_i и се добавя към сумата при всяко преминаване през тялото:

$$\text{Sum} \leftarrow \text{Sum} + a_i ;$$

3) Преди да започне “натрупването” на членовете a_i към сумата, тя има някакво *начално съдържание*. Това се осигурява с подходящ оператор преди цикъла. (Най-често това начално съдържание е 0.)

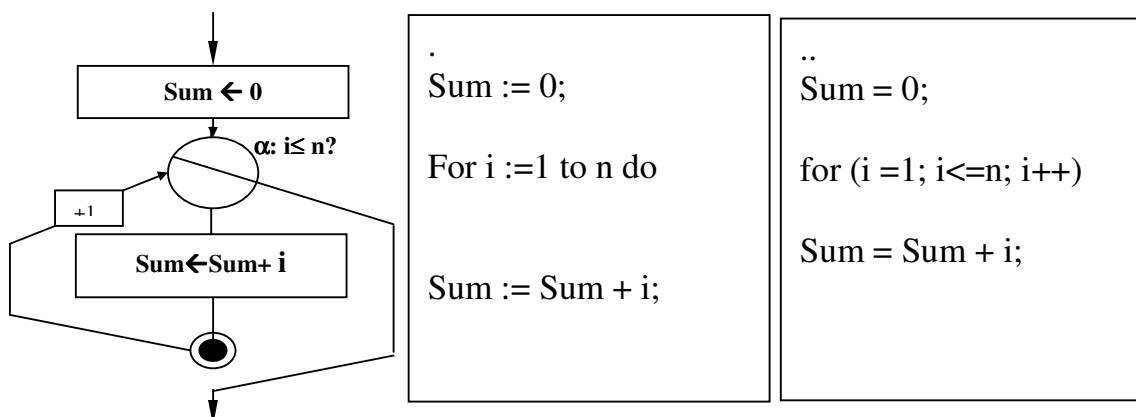
Долу е илюстрирана общата схема за натрупване на сумата

$$\text{Sum} = \sum_{i=\text{н. ст-ст}}^{\text{посл. ст.}} a_i :$$



Пример:

Натрупване на сумата на първите n естествени числа.



Може за проверка на програмата за натрупване на сума, след натрупването в променливата Sum , да се провери с подходящ оператор дали $Sum = n \cdot (n+1) / 2$. (по формулата на Гаус $\sum_{i=1}^n i = \frac{n(n+1)}{2}$). В много случаи съществуват формули от типа на Гаусовата. В още по-голям брой случаи *няма такива формули* и сумите програмно се “натрупват”.

➤ *Натрупване на суми при алтернативна¹⁴ смяна на знака*

В примерите дотук разгледахме крайни суми от типа $\sum_{i=1}^n a_i$. Понятията *ред*, *сума на ред*, *граница на безкраен ред* са предмет на математическия анализ и няма да се спираме на тях тук. За нашите цели, ще представяме сумата на ред така:

$$\sum_{i=1}^{\infty} a_i = \boxed{+} \boxed{} \boxed{+} \boxed{} \boxed{+} \boxed{} \boxed{+} \boxed{} \boxed{+} \boxed{} \boxed{+} \boxed{} \boxed{+} \boxed{} \boxed{+} \boxed{} \boxed{+} \boxed{} \boxed{+} \dots$$

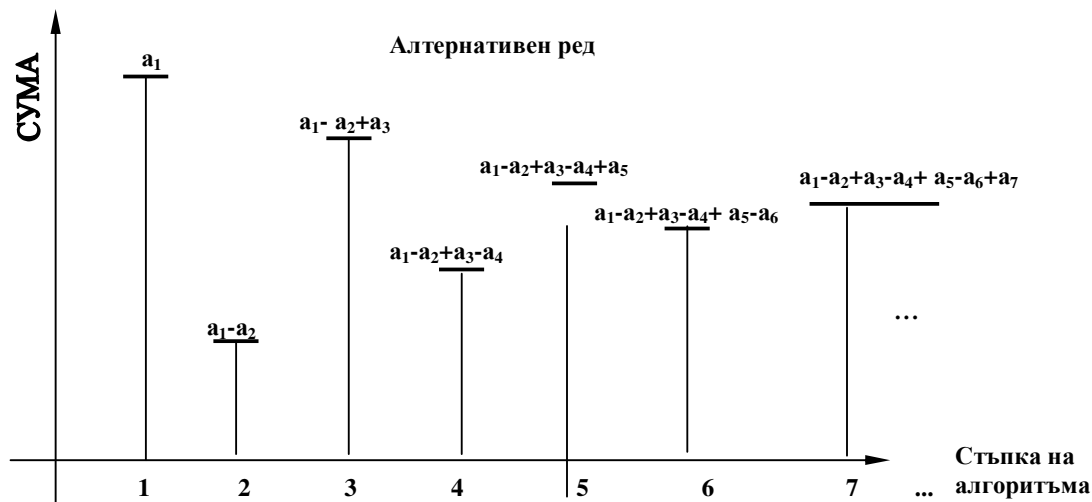
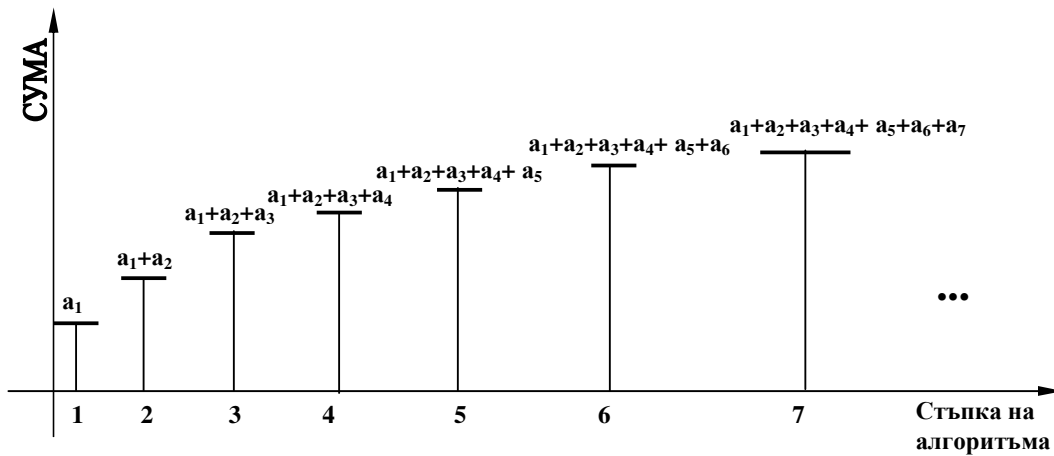
Не е възможно да бъдат сумирани безкрайно членовете на един ред с алгоритъм, предвид свойството *крайност* на алгоритъма. Затова безкрайните суми се сумират “донякъде”. Въпросът “докъде” е друг, нелесен въпрос, с който няма да се занимаваме тук.

Друг вид суми, които представляват интерес, са тези на *алтернативните* редове.

$$\sum_{i=1}^{\infty} a_i = \boxed{+} \boxed{} \boxed{-} \boxed{} \boxed{+} \boxed{} \boxed{-} \boxed{} \boxed{+} \boxed{} \boxed{-} \boxed{} \boxed{+} \boxed{} \boxed{-} \boxed{} \boxed{+} \dots$$

¹⁴ Думата “алтернативен” е с латински корен *Alter*, което означава “един от два”.

Нека си представяме сумирането на ред и на алтернативен ред, както е показано на следващите схеми:



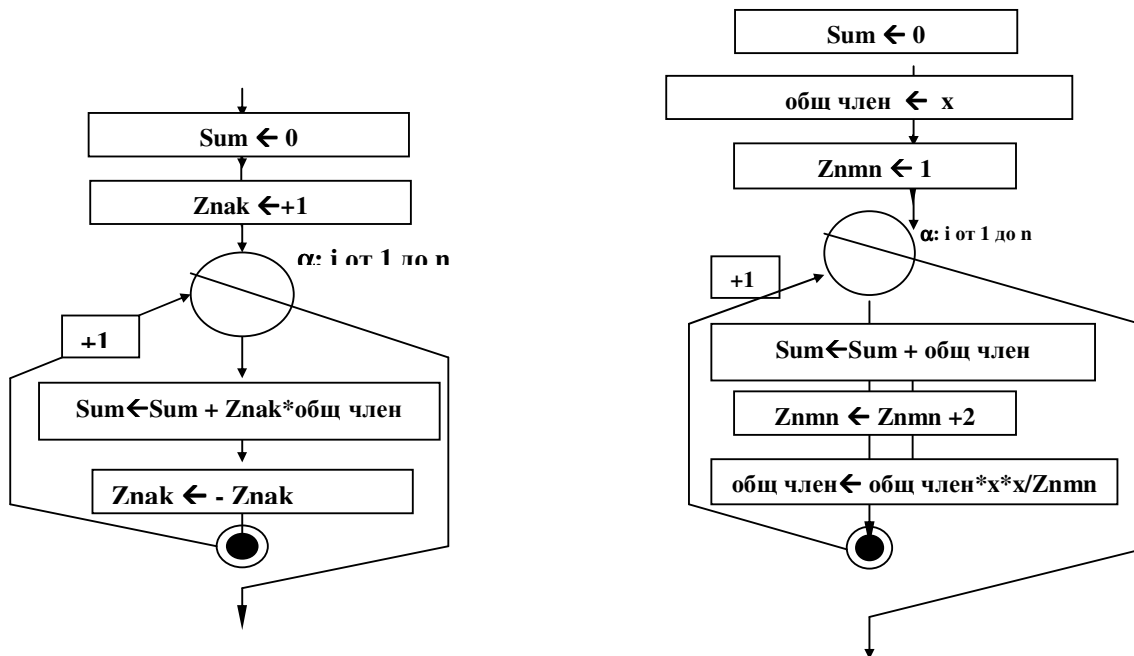
Изобразените редове $\sum_{i=1}^{\infty} a_i$ са безкрайни и сумата им не може да бъде изчислена с машина, освен с някаква точност т.е. при сумиране “донякъде”.

Много математически обекти могат да се представят като суми на безкрайни редове. Наличието на такова представяне много активно се използва в машинните алгоритми, защото машината, както се убедихме, може да сумира. Примери за това са функциите $\sin(x)$ и $\cos(x)$, които могат да се развият в алтернативни редове и да се представят като суми. Изчисляването на стойността на тези функции става именно чрез сумиране. Разбира се, има и редица тънкости, свързани с точността при изчисленията и с въпроса докъде да се “отреже” безкрайният ред, но засега ще се запознаем със сумирането на алтернативни редове по принцип. Дали сумирането на такива редове да става по брояч или итеративно по условие α (α : вече се получават много близки стойности на сумата за поредната стъпка на алгоритъма) е отделен въпрос.

Нека за конкретност да си представим, че сумираме по брояч. За алтернативния ред ще приложим следното представяне:

$$(+1).a_1 + (-1).a_2 + (+1).a_3 + (-1).a_4 + (+1).a_5 + \dots + (+1).a_n \dots +$$

Тази “тънкост” : $(+1)$ се сменя с (-1) и обратно, може да се реализира с една променлива, например “znak”, която се “обръща” при всяко преминаване през тялото на цикъла, в който става натрупването на сумата:



Друга тънкост е това, че поредното събираемо в много случаи може да бъде изразено по-ефективно не като общ член, зависещ от параметъра на цикъла (i), а с използване на стойността на предходно изчисленото събираемо. Например функцията $\sin(x)$ се представя с реда:

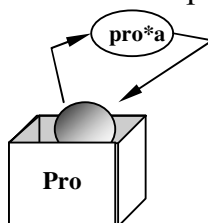
$$\sin(x) = +\frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots = \sum_1^{\infty} (-1)^{i+1} \frac{x^{(2i-1)}}{(2i-1)!}$$

Проследете схемата на управление за сумиране на първите n члена на този ред, показана горе вдясно.

➤ *Натрупване на произведения*

Подобно на сумите, могат да се съставят и произведения:

$$a_1 . a_2 . a_3 . a_4 . a_5 \dots a_n = \prod_{i=1}^n a_i$$

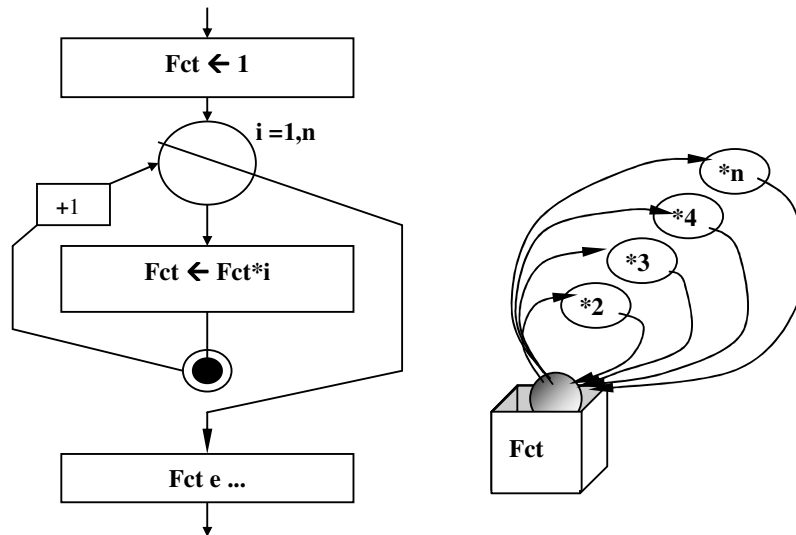


По напълно аналогичен на сумите начин, преди да се премине към натрупване на произведението в цикъл, на променливата, в която става натрупването, трябва да се зададе начална стойност.

Пример: факториел.

$$1*2*3*4*5*6*...*n = n!$$

Факториелът е произведението $\prod_{i=1}^n i$ на първите n естествени числа. Алгоритъмът за натрупване на произведението “факториел” може да се състави почти като този за сумата на първите n естествени числа.



Начините за пресмятане на факториел не се ограничават само с тази примерна схема. Например, функция за изчисляване на факториел може да се представи рекурсивно.

♦ Примерни задачи за упражнение

1. Да се съставят програми за натрупване на следните суми:

$$1) \sum_{i=1}^n i^2 \quad 2) \sum_{i=2}^n (i^2 - 2i) \quad 3) \sum_{i=3}^n (i^3 - 2i)$$

2. Да се състави програма за пресмятане на π , като сума на следния алтернативен ред (ред на Лайбниц):

$$\frac{\pi}{4} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - \dots = \sum_{i=1}^{\infty} (-1)^{i+1} \frac{1}{2i-1}$$

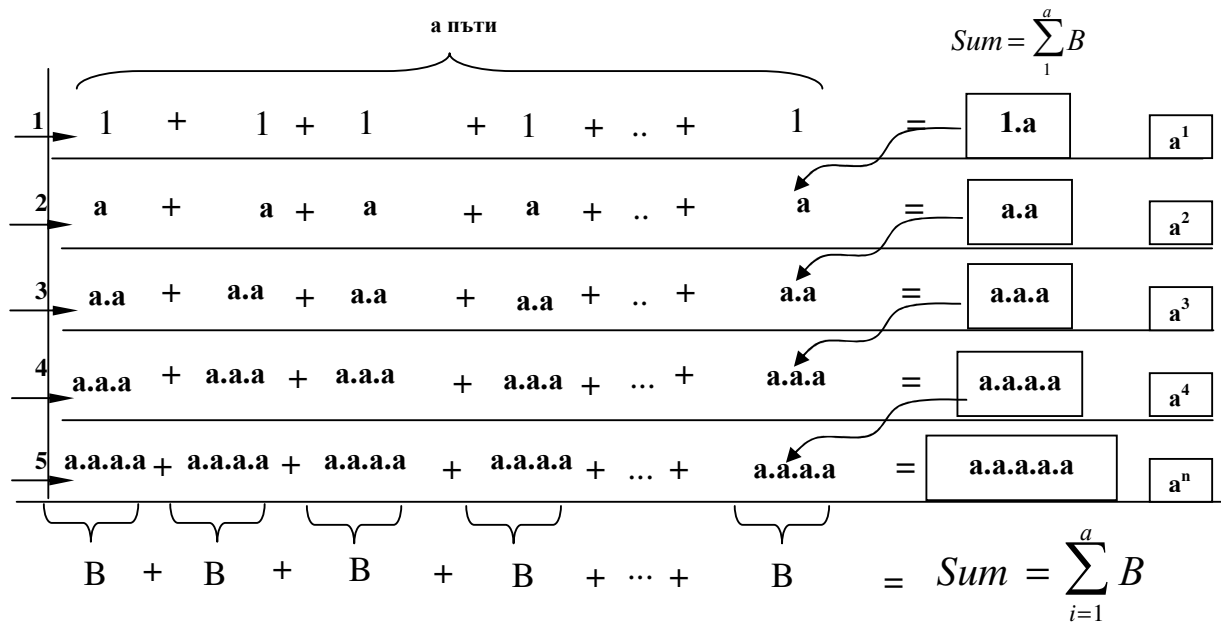
3. Да се състави алгоритъм и програма за повдигане на цяло положително число a на цяла положителна степен n , при използване само на адитивни операции.

Препоръки

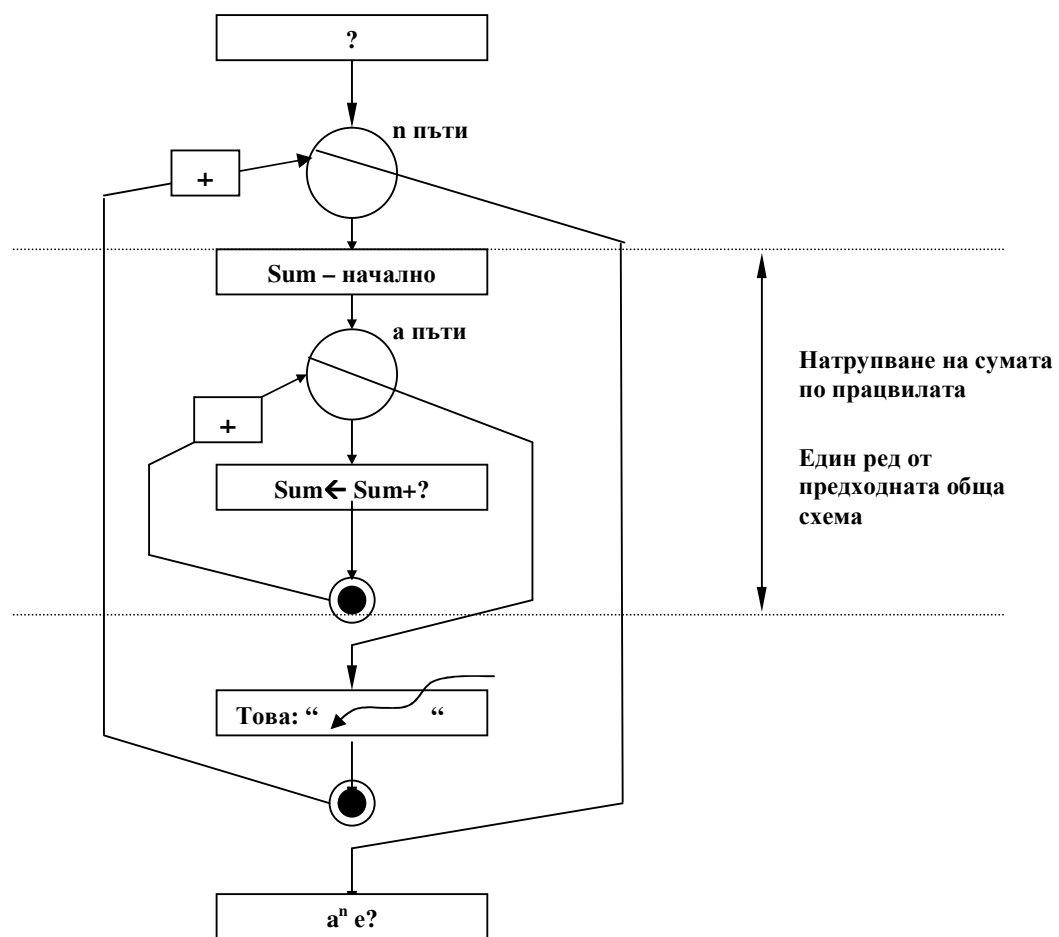
Да уточним, че предлаганото решение на задачата не е нито единственото, нито най-доброто. Съществуват други, по-ефективни алгоритми за пресмятане на степен на число. За нашите цели, ще използваме наученото за натрупване на суми. Да използваме следното представяне:

$$\begin{aligned}
 &\underbrace{1 + 1 + 1 + 1 + 1 + \dots + 1}_{\text{a пъти}} = a \\
 &\underbrace{a + a + a + a + a + \dots + a}_{\text{a пъти}} = a \cdot a = a^2 \\
 &\underbrace{a^2 + a^2 + a^2 + a^2 + \dots + a^2}_{\text{a пъти}} = a^2 \cdot a = a^3
 \end{aligned}$$

Ето една опорна схема за алгоритъм:



Като се има предвид дадената горе опорна схема, трябва да се допълни схемата на управление:



3 БАЗОВИ АЛГОРИТМИ И ПРОГРАМИ

➤ *Обща постановка*

В предходните теми показахме примерни алгоритми за елементарни задачи, за съставянето на които е достатъчно да се познава начинът на работа на конструктите за управление и особеностите на използване на променливи от стандартен тип. Очевидно, в практиката се налага решаване на по-сложни от разгледаните дотук задачи и в много случаи това може да стане с използване на алгоритми и програмиране.

От изложеното в предходните теми става интуитивно ясно, че за работата на един машинен алгоритъм не могат да бъдат съставени произволни среда и управление. Очаква ли се да има някакви строги формални правила, по които средата и управлението на по-сложен алгоритъм могат да се изградят?

Ще бъдат изложени някои методически похвати за структуриране на алгоритъм, работа със структурирани елементи в средата и принципи за обвързване на управлението със средата при съставяне на алгоритми и програми за основни алгоритмични задачи, подкрепени с примери от числените методи и базовите структури данни. Ще започнем от управлението, което е “най-гъвкавата” съставна част на алгоритъма.

3.1 Структурно програмиране

3.1.1 Изграждане на алгоритъм по блокове

Да въведем някои термини:

Алгоритмичен блок – смислово обособена част от алгоритъм (градивен елемент), чието *управление има единствена входна точка и единствена изходна точка*.

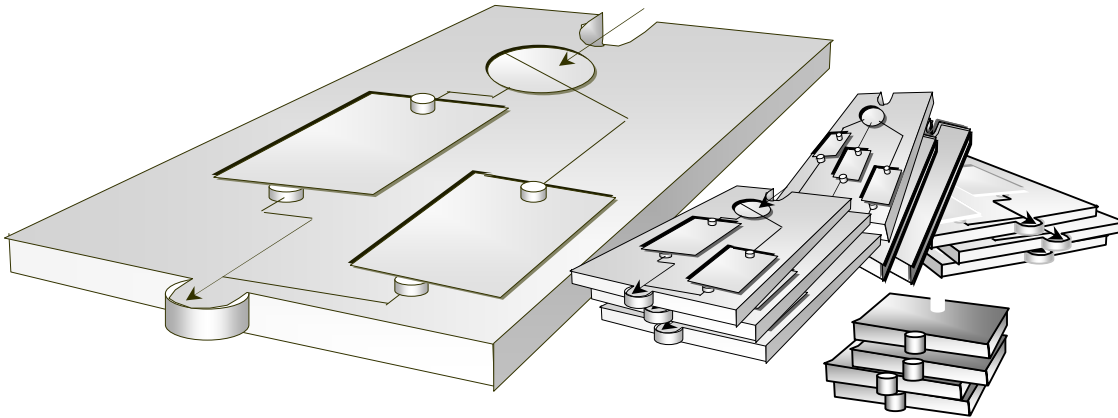
Горната дефиниция посочва, че блокът е “смислово обособена част”, т.е. алгоритмичният блок съответства на решаване на добре дефинирана подзадача. От тази гледна точка алгоритмичният блок е логически завършена, обособена стъпка. За да обвържем изграждането на логически смислените единици от алгоритъма със съществуващите в езиците за програмиране градивни единици, най-напред ще разгледаме алгоритмични блокове, които съответстват на блокове от схема за управление, както те бяха дефинирани отначало, а именно:

- конструкти за управление;
- инструкции за действия над средата.

И двете отговарят на въведения термин “Алгоритмичен блок”.

На следващата фигура е илюстрирана идеята на изграждането на алгоритъм по блокове. Дефиницията и илюстрацията ни карат да направим аналогия между *изграждането на алгоритъм по блокове* и игрите “млад конструктор” и “пъзел”, взети заедно. Изграждането на “конструкцията” на алгоритъма става със съединяване на алгоритмични блокове при “напасване” – съединяване на

блоковете по единствената им входна и единствената им изходна точка и/или влагане на блокове един в друг.



За да дефинираме процеса на формалното изграждане (синтез) на несложен алгоритъм като процес на *съединяване, вграждане и обхващане* на алгоритмични блокове, ще дадем следната (работна) рекурсивна дефиниция на алгоритмичен блок:

<p>Алгоритмичният блок е :</p> <p>конструкт за управление, имащ <i>тяло</i> (краен брой тела)</p>	<p>1. най-малкото тяло е инструкция за действие (оператор в дадения език)</p> <hr/> <p>2. тялото е Алгоритмичен блок.</p>
--	--

Тази дефиниция ни дава основание да говорим за *вграждане и обхващане* на алгоритмични блокове, като дефинира какво представлява най-малката единица, с която се работи при “конструирането”. Важно свойство на този формален подход е, че изградената по този начин логическа структура на алгоритъма се свежда до схема на управление и може да бъде еднозначно преведена на езика за програмиране, чийто конструкти са използвани в блоковете.

От така въведения модел на изграждане на алгоритъм се вижда, че с краен брой блокове могат да бъдат получени безкрайно много различни алгоритми. Освен това, не трябва да се забравя, че е възможно една и съща задача да се решава с повече от една различни схеми на управление.

На следващата фигура е показана една примерна схема на управление, изградена от блокове по посочения принцип.

Да напомним, че в началния момент от процеса на създаване на алгоритъм се изяснява постановката на алгоритмичната задача. Въпросът, който ни интересува по отношение на задачата е: може ли въобще решаването на задачата да се представи като алгоритъм? Как да разберем това? Съществуват теоретични подходи за това, но един практически начин да отговорим на този въпрос е да опитаме да изградим алгоритъм.

Нека тръгнем в разсъжденията си “отзад напред” – нека алгоритмът от схемата вдясно е вече съставен за решаването на конкретна алгоритмична задача. В резултат на какъв процес на разсъждения схемата на управление се е получила именно такава?

Можем да предположим, че това е станало чрез последователно разлагане (декомпозиране, анализ) на поставената задача с употреба на конструктите за управление. Ако наличните в схемата блокове са изпълними оператори, това е алгоритъм. Ако някой блок е неизяснена подзадача, алгоритмично решение все още не е намерено. Ако за някой блок докажем, че не може да се представи с алгоритъм, цялата задача няма алгоритмично решение при тази декомпозиция.

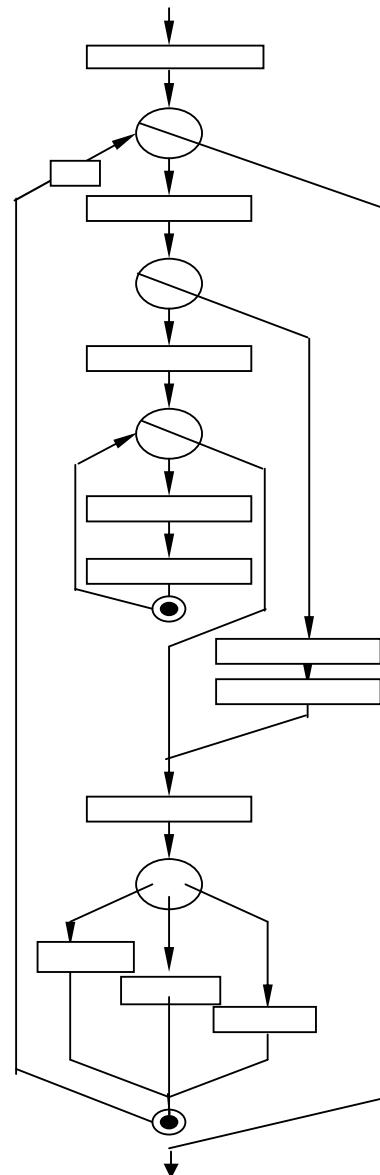
Да обърнем внимание на това, че формализирането на алгоритмичната задача е свързано с избор на подходяща среда. Алгоритмът от схемата работи над конкретна среда (на схемата тя не е изобразена). Средата може да бъде нестандартна, изцяло измислена от програмиста, абстрактна.

На този етап ще се запознаем с един подход при съставянето на управление, като предполагаме, че подходящата среда може да бъде съставена.

Въпросът е по какви методики може да се състави алгоритъм. Не бива никога да се забравя, че една и съща алгоритмична задача може да се решава от множество по-добри или по-лоши “пъзели”. За жалост – няма правила за конструиране на алгоритъм. Почти както когато се нарежда пъзел, при построяването на алгоритъм може да се подхожда по различни *стратегии*. Например:

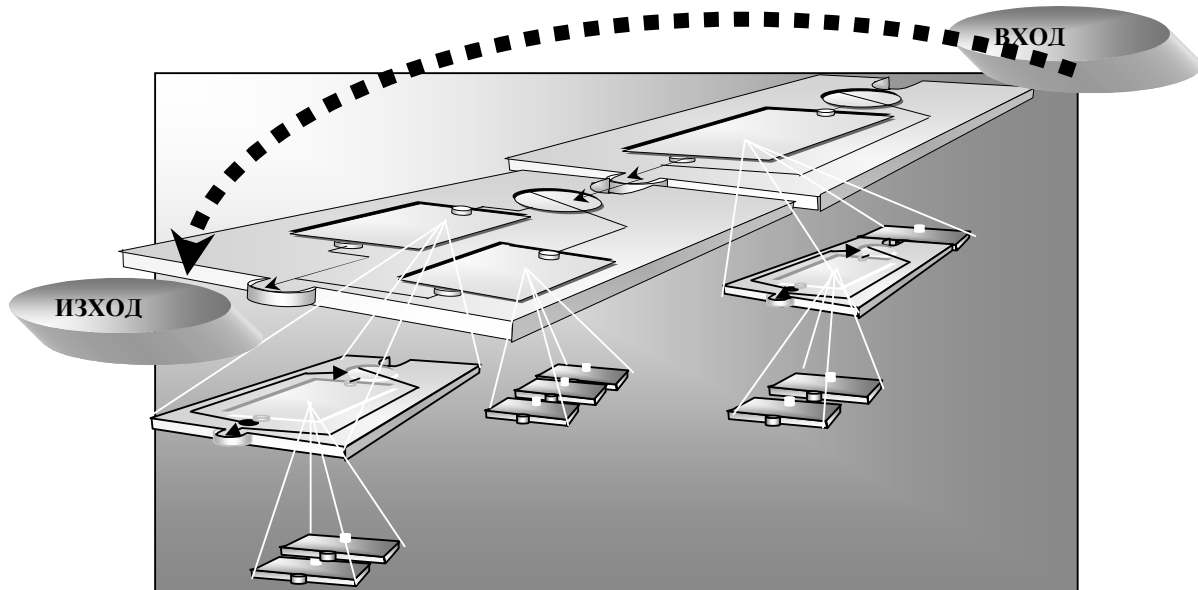
А. Най-напред се изгражда най-обща картина. После тя се детайлизира като постепенно картината се “избистря” и уточнява във все по-прецизни нива на детайлност. В термините на нашата аналогия с играта “млад конструктор” това означава, че ако например трябва да конструираме автомобил, най-напред *решаваме*, че той се състои от купе и колела. После гледаме от какви налични части те се състоят или могат да бъдат изградени.

Б. Най-напред в задачата се разпознават известните подзадачи и се сглобяват тези части, за които се знае (отнякъде) каква е вярната им конструкция. После така сглобените части се “нагаждат” – съединяват, вграждат и надграждат, така, че да се получи общата картина.



Ще се опитаме да илюстрираме една методика за изграждане на алгоритми, наречена “Top-Down”. Тя ползва и двете изложени горе стратегии.

3.1.2 Top – Down методика за изграждане на алгоритъм



Начинът на прилагане на тази методика е илюстриран на фигурата горе: целта е от най-горното “ниво”, където стои алгоритмичната задача, да се достигне до най-ниското ниво с прилагане на определени правила за декомпозиция. Блокчетата от най-ниското ниво са изпълними оператори, а блоковете от по-високите нива съответстват на алгоритмични блокове, т.е. те са *дефинирани подзадачи*.

Така създаването на алгоритъма се свежда до последователна детайлизация по нива. Това разграждане (детайлизация) става по определени общи правила, в съответствие със следното принципното ограничение:

- Смесовите единици, които се използват при разграждането са *алгоритмични блокове*. Това е необходимо условие за това алгоритъмът да може да бъде изпълнен от машина.

Този метод предлага и едно голямо удобство при формализирането на задачата, а именно:

- За описание на действието и предназначението на подзадачите-блокове от нивата, по-високи от крайното ниво (програмата), се използва естествен език.

Ето някои основни *правила* при разграждането:

- Разграждането става само “отгоре-надолу”.
- По-горното ниво е *описано изцяло* от по-долното ниво.
- Всеки блок от ниво n е съставна част на *единствен* блок от ниво $n-1$.
- Разграждането продължава, докато получените блокове не се окажат инструкции за действие – изпълними оператори.

Практически правила при прилагане на методиката “отгоре – надолу”

1. При разграждането се преминава последователно от ниво “задача” към по-ниското, първо ниво, като се използват “прости” алгоритмични блокове, т.е. такива, които съответстват на конструкти за управление. Предназначението и действието на простите блокове *се изписва в тях на български език*. Същата технология се прилага при по-нататъшното разграждане.

2. Всеки от получените при разграждането блокове е добре дефинирана подзадача, която има единствена входна точка. Желателно е да се отбележи изрично с текст *входното условие*, т.е. в какъв случай този блок се изпълнява. Добре е също при разграждане на блок от ниво n е да се запишано на кои блокове от по-високо ниво той е съставна част.

➤ Пример за прилагане на Top–Down методика

Описаната методика се усвоява достатъчно добре след многократно упражняване с конкретни задачи. Нека приведем първия пример, за който е избрана една много “ясна” задача:

Дадено:	Търси се:
Уравнението $ax^2 + bx + c = 0$ в което a , b и c (вход на задачата) са известни коефициенти	Решението на уравнението. (изход на задачата)

Уравнението $ax^2 + bx + c = 0$ ($a \neq 0$) има *аналитично решение*. Това означава, че е възможно корените $x_{1,2}$ да бъдат изразени във функция на зададените a , b и c посредством “директен” израз:
$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

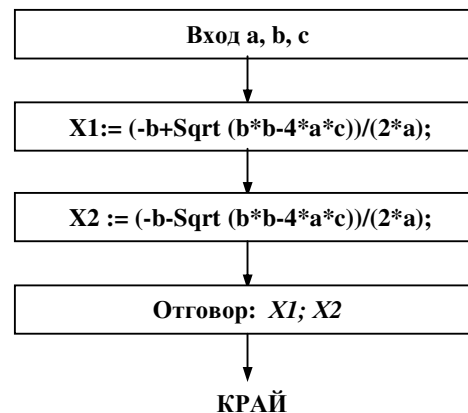
Задачата е да се състави алгоритъм, по който се намира решение при използване на добре известната формула.

На пръв поглед задачата изглежда много проста, защото изразите, с които се пресмятат стойностите на неизвестните, могат да бъдат представени с два изпълними оператора (долу е използвано означение на аритметичните операции в Pascal):

$X1 := (-b + \text{SQRT}(b*b - 4*a*c)) / (2*a);$

$X2 := (-b - \text{SQRT}(b*b - 4*a*c)) / (2*a);$

Но какво би станало, ако алгоритъмът изглежда както е показано вдясно? Този алгоритъм не решава поставената алгоритмична задача.



Ако по него се състави програма, тя ще работи *само понякога*. Например при въведено: $a = 0$ или a , b и c , такива, че $(b^2 - 4ac) < 0$, ще настъпва прекъсване на изпълнението с евентуално съобщение от типа на “divide by 0 on 48683584588”.

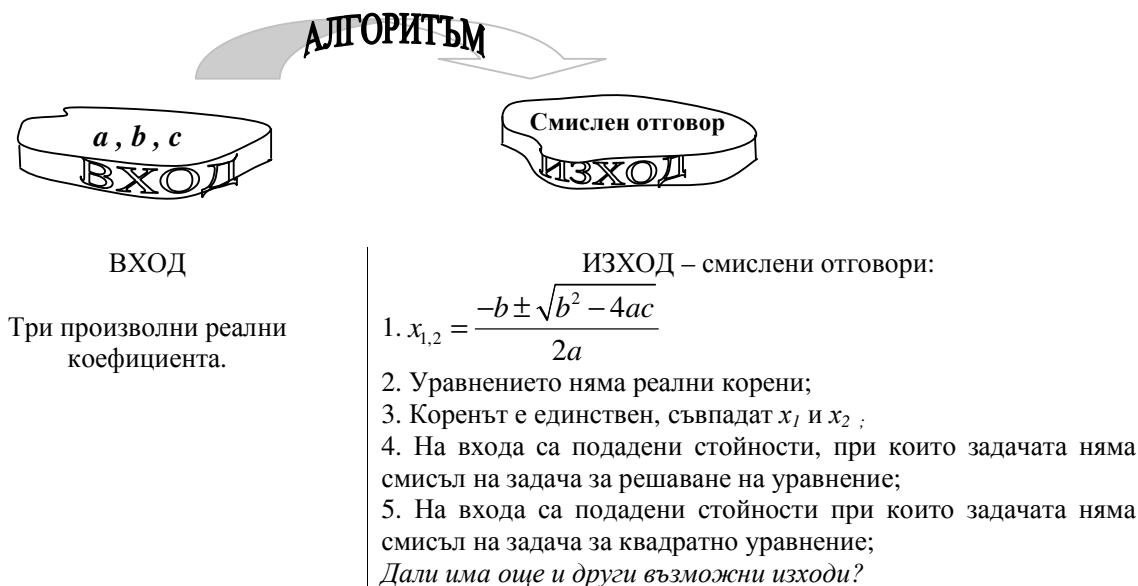
При това явно не е допустимо да се предвижда защита на входа от рода на: “Драги потребителю, задай такива a , b и c , че $(b^2 - 4ac) \geq 0$, а a не трябва е нула моля. ... Пак не ги зададе такива!”.

Да се опитаме да “организираме” алгоритъма така, че той да решава поставената задача и да дава смислен отговор за всички възможни стойности на входа – a , b , c . При това, принципно е възможно входните коефициенти да не са задавани от клавиатура, а да са и подадени на входа като резултат от изчисление, извършено от друга програма.

➤ Съставяне на алгоритъма по Top–Down методика

На първо място трябва да се уточни поставената задача, като се дефинират двете множества – това на входа и това на изхода на задачата. В нашия случай става въпрос за решаване на квадратно уравнение и това е лесно. Целта е да се “обозрат” всички възможни входове и всички очаквани смислени отговори, които биха се получили при решаване на квадратно уравнение.

0. Ниво “нула” – Алгоритмична задача



На това “нулево” ниво логическа декомпозиция, алгоритъмът изглежда както е показано на фигурата вдясно. Състои се от три последователни блока, двата от които са изразими направо с оператори на език за програмиране и следователно няма да бъдат разграждани. Трябва да се разгради блок 0.



Разграждане на блок 0

Решението, т.е. отговорът, следва да се получи след “програмно реализиран” анализ на получените на входа данни за коефициентите.

Въпросът, който се задава на най-първично равнище на анализ е дали за всички различни случаи на входни данни за a , b и c , заместени в израз

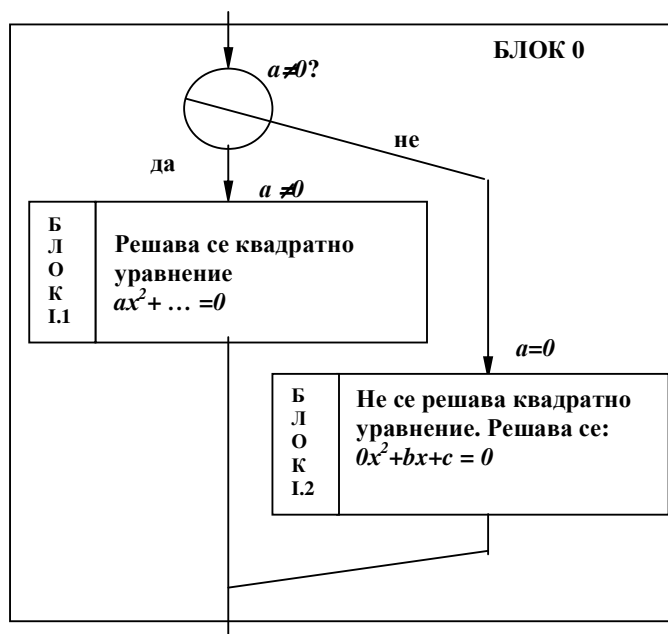
$$ax^2 + bx + c = 0, \quad (1)$$

следва той да се квалифицира като “квадратно уравнение” и да бъдат прилагани добре известните формули:

$$x_{1,2} = \frac{-b \pm \sqrt{D}}{2a}; D = b^2 - 4ac \quad (2)$$

Да видим стойността на кой входен параметър определя уравнението (1) като “квадратно”, за което са валидни формулите (2)? Очевидно, това е стойността на a . *Необходимо и достатъчно* е a да не е нула, за да бъде изразът (1) квадратно уравнение.

Според прилаганата стратегия, трябва да се разгради тялото на блок 0 до алгоритмичен блок, който изразява направеното разсъждение за входния коефициент a . На фигурата вдясно е показан двуклон, с който се описва изцяло ситуацията. Забележете, че освен фактът, че използваната структура за управление е познатият двуклон, нищо друго на схемата не е изразено с програмни средства.



Текстът вътре в получените блокове I.1 и I.2 е описание, на български език, на изложените горе разсъждения. На входа на получените блокове е отбелязана входната им клауза. Този двуклон описва напълно разграждания блок 0.

Блоковете I.1 и I.2 не са разградени до степен да са пряко изразими с оператори на език за програмиране. Следователно разграждането им трябва да продължи.

Разграждане на ниво I

Да започнем с разграждането на блок I.2 от началната декомпозиция на задачата.

Ще приложим разсъждения, напълно аналогични на тези при разграждането на предишното ниво. Разграждането е показано на следващата фигура.

Въпрос: Кой определя израза $bх + c$ като линейен израз за x ?

Отговор: b

Ако $b=0$, не се получава израз за x , това е особен случай!

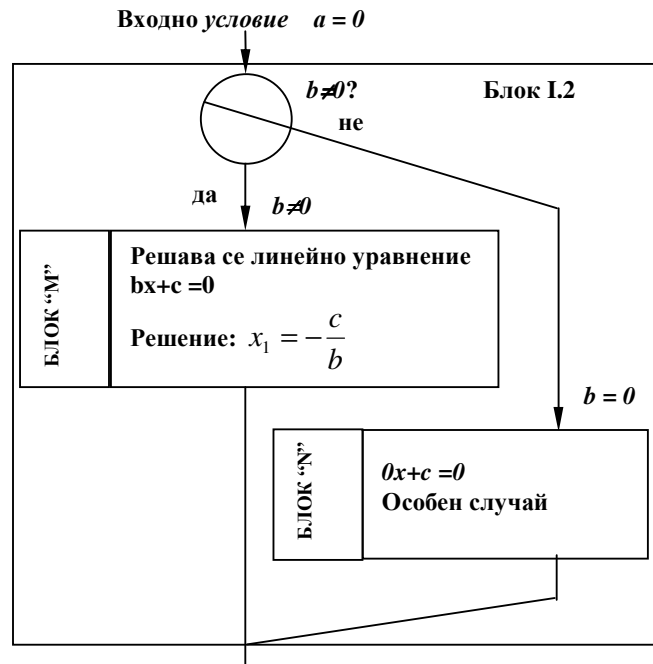
Блокът I.2 е разграден на два блока – М и N.

М: $X1 \leftarrow -c/b$;

N: Out ('Особен случай!');

Блоковете М и N могат да бъдат изразени на език за програмиране.

С това блокът е разграден докрай и тази част от алгоритъма е готова за “превод” на програмен език.



Да направим и един допълнителен анализ:

Блок М : $x_1 = -\frac{c}{b}$. Алгоритъмът преминава през този клон само когато $b \neq 0$ и няма опасност от делене на нула. Ако $c=0$, т.е. уравнението е $b.x+0 = 0$, отговорът се получава “сам”: $X1 \leftarrow 0$, защото $X1 \leftarrow -0/b$

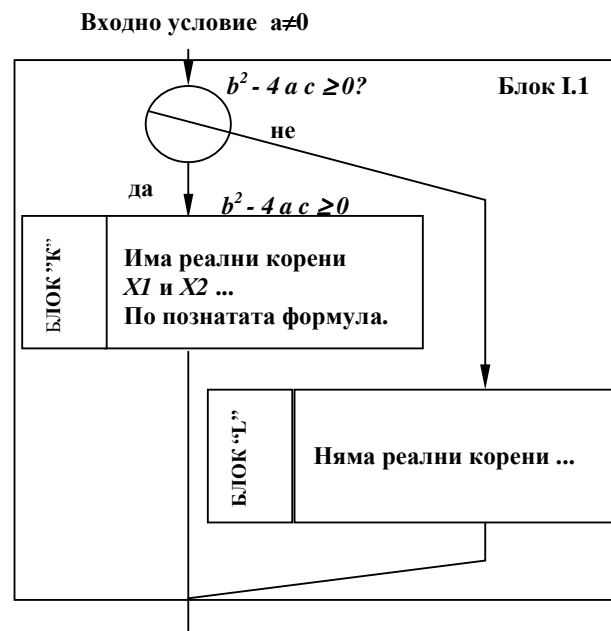
Разграждане на блок I.1

Прилагането на разсъждения, напълно аналогични на тези при разграждането на предишното ниво, води до получаване на блока, показан на фигурата вдясно.

Въпрос: Кога едно квадратно уравнение има реални корени, та да са приложими формули (2) ?

Отговор: Когато $b^2 - 4ac \geq 0$

Ако $D < 0$, уравнението няма реални корени и формулите не са приложими.



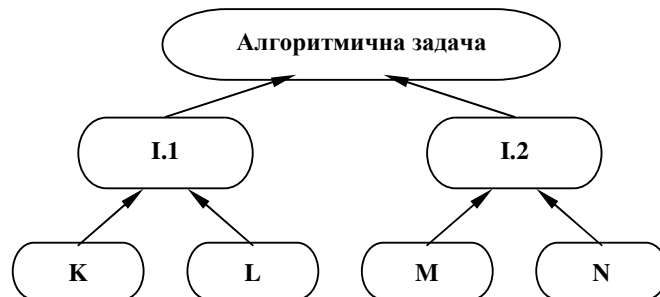
Блокът I.1 е разграден на два блока – К и L. Блоковете К и L могат да бъдат изразени на език за програмиране.

Да направим и тук един допълнителен анализ.

Блок К : Алгоритъмът преминава през този клон само когато $a \neq 0$ и няма опасност от делене на нула. Не е необходимо да се разглежда отделно случаят “ $D = 0$ ”, защото, ако

$b - 4ac = 0$, в блок К програмата “сама” ще получи $X1$ и $X2$ равни.

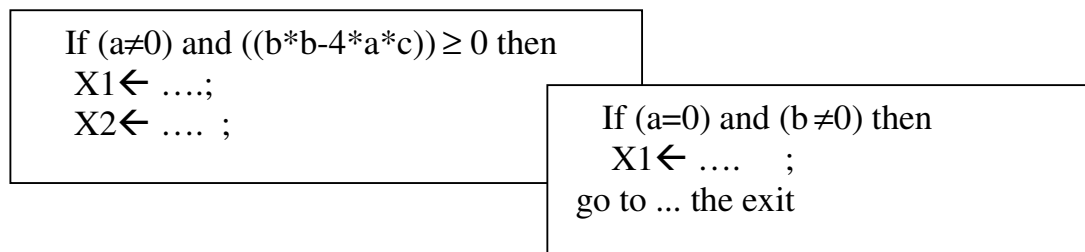
Ето резултатът от разграждането:



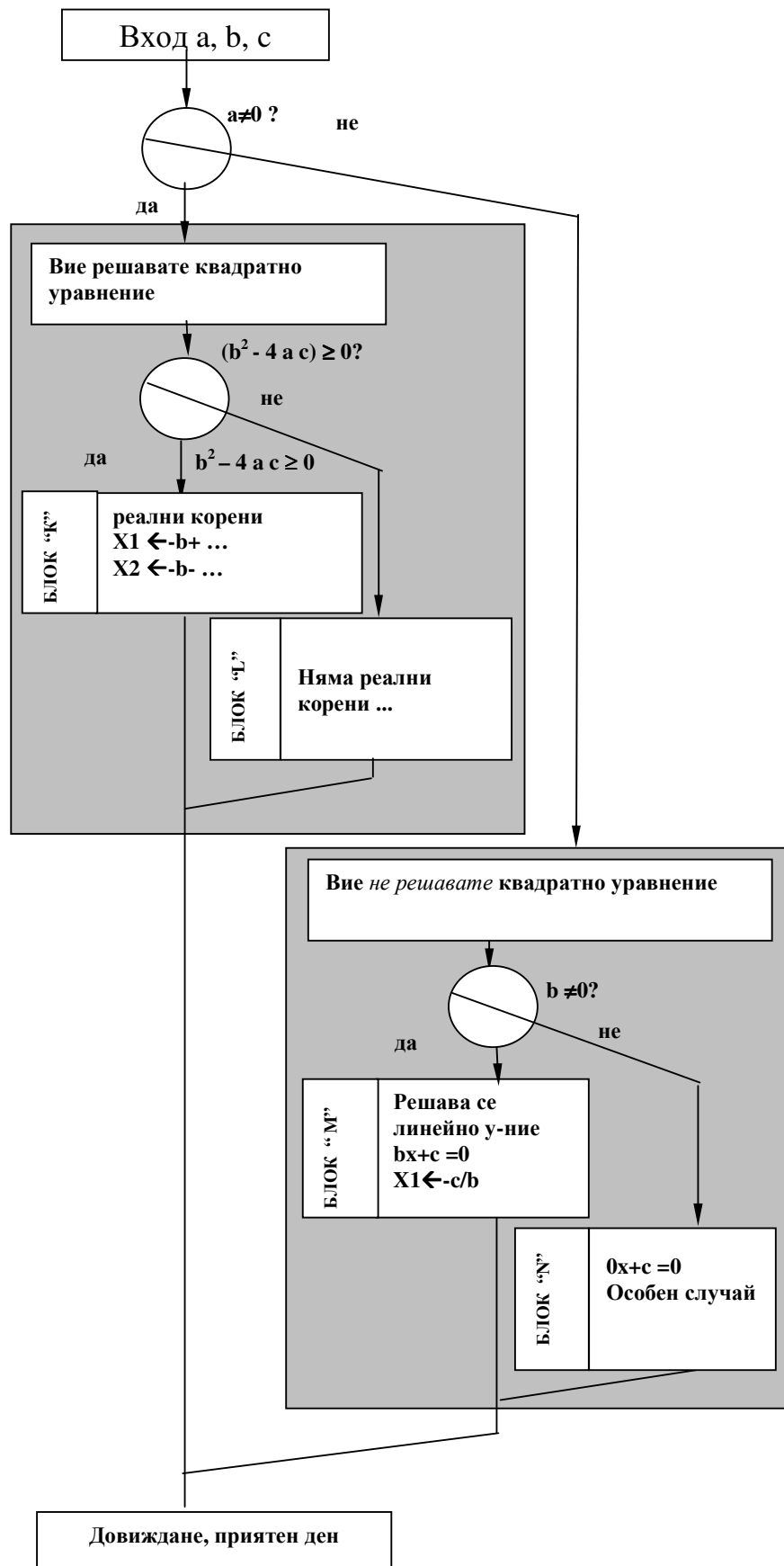
Твърдим, че алгоритмичната задача е решена пълно, че няма пропуснати “варианти на входа”, че няма рискове от делене на 0 и че се получават всички възможни отговори. Схемата на управление се получи след разграждане на алгоритмичната задача на най-общи, по-детайлни и още по-малки алгоритмични блокове, както е илюстрирано на фигурата горе.

На следващата страница е показана пълната схема на управление на получения по стратегията “отгоре-надолу” алгоритъм. Схемата съдържа множество вградени блокове, при това на повече от едно ниво на влагане. Преминете мислено през нея за различни входни данни, за да се убедите, че така получената схема се “държи разумно”.

Същата задача може да бъде решавана и при използване на “Bottom-up”, т.е. “отдолу нагоре” логика на построяване. Решението би съдържало програмни фрази, съответстващи на логически парченца от дъното на приложения от нас процес на съдържателно разграждане. Например (псевдоезик):



Опасността от подобен подход се състои в неорганизираното формулиране на множество частни случаи, което може да не бъде пълно или по-късно да бъде неправилно обобщено или организирано за “по-високите нива” на управлението. По тази причина изграждане по принципа “отдолу нагоре” не се прилага, а с термина “Bottom-up” обичайно се означава избирателното разработване на отделни блокове от планирания по методиката “Top-Down” алгоритъм.



Важно е да се подчертае, че прилагането на тази методика води до получаване на много ясна структура както на схемата на управление, така и на съответстващия ѝ програмен текст. На фигурата вляво е показана блокова структура на програма, получена по този начин. Тази структура отговаря и на нашата “конструкторско-пъзелска” аналогия.

Може да се твърди, че програми, съставени по този начин, са по-лесни за разчитане от друг програмист.

Подходът за това разгадаване включва “разпределянето” на програмния текст на блокове така, както е показано на фигурата горе. Графичното разграничаване на блоковете на програмни текстове се използва често при разчитане на алгоритъма на чужди програми, т.е. при извличане на смисъла им.

Да обърнем внимание на практическите правила и на дисциплината на работа при съставяне на алгоритми и да дадем някои напътствия за писането на програми :

1. Добре е винаги да се пише първо на лист хартия.
2. Добре е винаги да се съставя първо схемата на управление.
3. Добре е винаги да се ограждат блоковете на написания програмен текст.
4. Добре е да се преминава към набиране на програмния текст едва след като програмата съществува на лист, съответства на схемата на управление и е “оградена” от съставни блокове.

Спазването на тези правила не е задължително, но дори напредналите програмисти ги спазват, когато решават по-сложни задачи. По тези правила не само че не се губи време, но се печели време. По-важното е, че програмите, съставени така, сработват по-лесно и се коригират по-бързо от тези, съставени по метода “проба-грешка”.

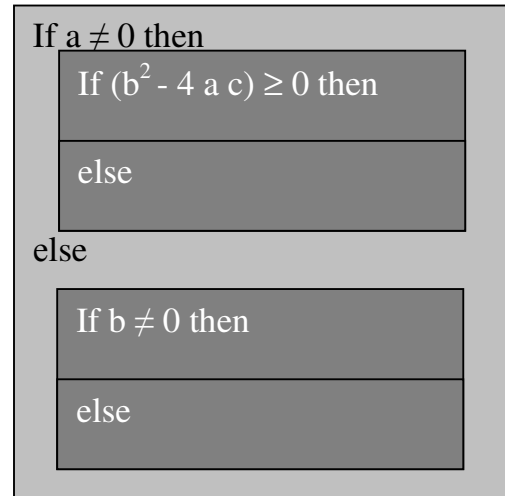
♦ Примерна задача за упражнение

1. В примера за квадратното уравнение, помислете как още да се разградят блоковете L и N.

2. Дадени са две точки A и B, и двете в първи квадрант (без да лежат на осите). Точките са зададени с техните координати (X_A, Y_A) и (X_B, Y_B) , реални числа. Координатите се задават от клавиатурата. (Помислете за защита на входа).

Без да се използват тригонометрични функции, само чрез сравняване на стойностите на координатите на точките, да се установи дали правата, преминаваща през двете точки е:

1. Успоредна на ос X
2. Успоредна на ос Y

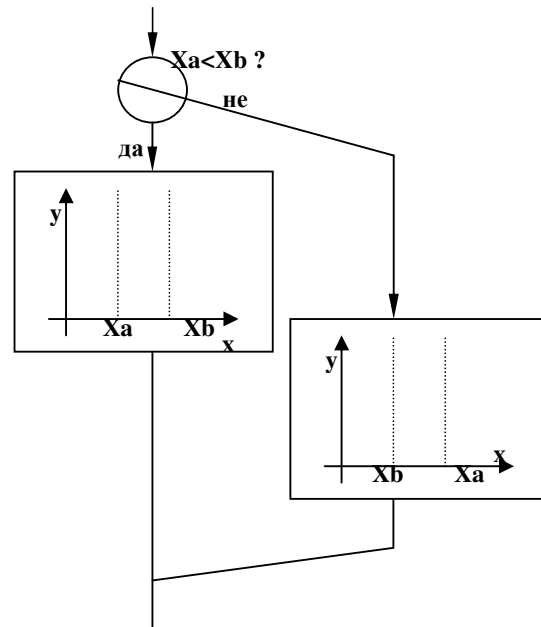


3. Наклонена наляво
4. Наклонена надясно

Да се състави алгоритъм, решаващ горната задача, като се приложи “Top-Down” методика.

Съвети:

1. Спазвайте стриктно методиката стъпка по стъпка. Изясняването на началната постановка на задачата е задължителната първа стъпка.
2. При разграждането, за описание на блоковете можете да използвате не само естествен език, но и схеми, поставени вътре в блока, например така, както е показано на фигурата вдясно.



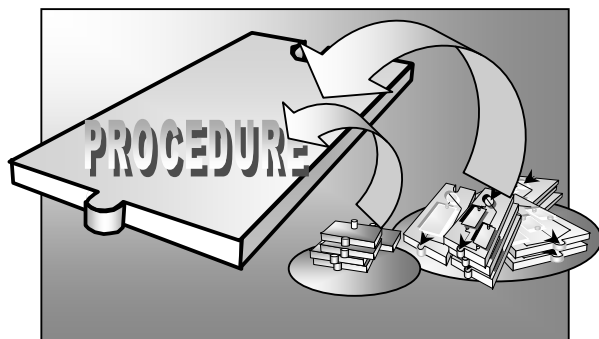
3.2 Процедури

Ще се спрем на изграждането на управление, несъвпадащо, надграждащо и допълващо съществуващите конструкти за управление. Става дума за изграждане на отделен, нов блок за управление, който може да се използва при изграждането на алгоритъма. Това съставено от програмиста “собствено блокче” се нарича **процедура**.

3.2.1 Изграждане на процедура

Процедурата е алгоритъм, който е изграден така, както се изгражда всеки друг алгоритъм.

Принципно процедурата се изгражда от алгоритмични блокове, например с конструкти за управление и инструкции за действия. Процедурата е алгоритмичен блок и има единствен вход и единствен изход. Веднъж съставена, процедурата може да бъде вграждана в друг алгоритъм по начина, по който се вграждат алгоритмичните блокове въобще.

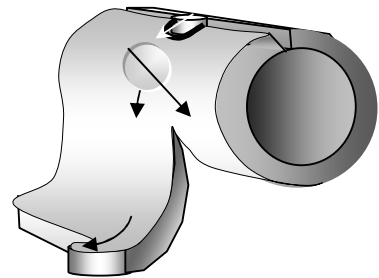


Процедурата може да бъде използвана като “градивен материал” на друг алгоритъм, но и самата процедура би могла да използва други процедури. Един алгоритъм може да бъде изграден изцяло от блокове, които са процедури. За езиците от процедурен тип е характерно именно това – в основата им е залегнала концепцията, че всеки алгоритъм е процедура и всеки блок на

алгоритъма може да бъде дефиниран като самостоятелна процедура. В тези езици е заложен разглежданият тук подход на изграждане на алгоритъм от блокове.

Възниква въпросът за това дали една процедура може да използва като градивен материал самата себе си. Отговорът е – да, и такава концепция има. В много езици за програмиране тя е реализирана – процедурата използва сама себе си като алгоритмичен блок. Такива процедури се наричат *“рекурсивни”*.

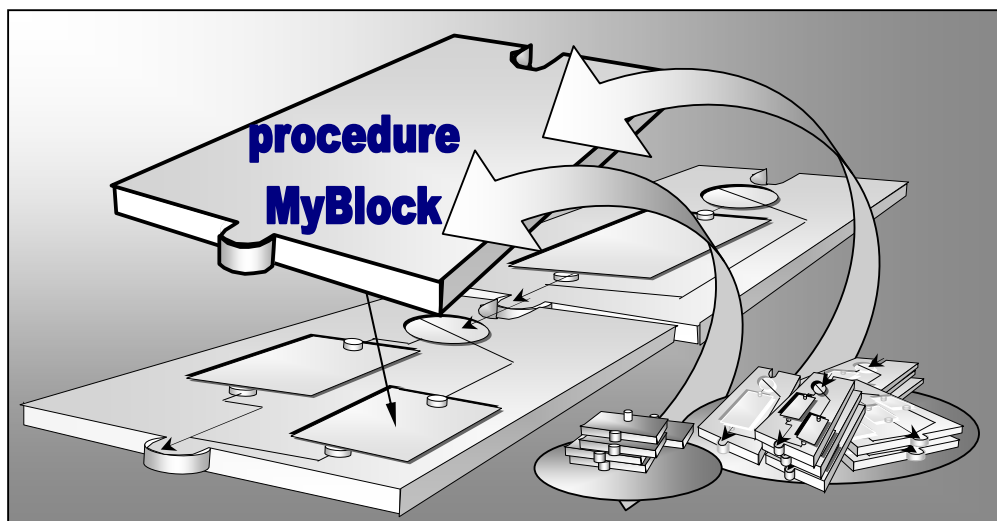
На фигурата вдясно е образно илюстриран алгоритмичен блок, процедура, която се “затваря” в самата себе си. Вижда се, че от гледна точка на управлението такова “самопозоваване” може най-общо да се опише на работата на цикъл. Рекурсията реализира нещо като “самозатваряне” на процеса на управление и може да стане причина изпълнението да се превърне в безкраен процес. Очевидно до подобна ситуация не трябва да се стига.



Използването на рекурсия е удачно, когато в средата на алгоритъма се работи с рекурсивно дефинирани обекти. По принцип, при съставянето на алгоритми рекурсията може да бъде избегната. Въпросът е дали това е най-удобно и най-ефективно за конкретен алгоритъм. Използването на рекурсивни процедури изисква определено равнище на алгоритмично мислене и няма да бъде разглеждано тук.

➤ *Процедурата от гледна точка на управлението*

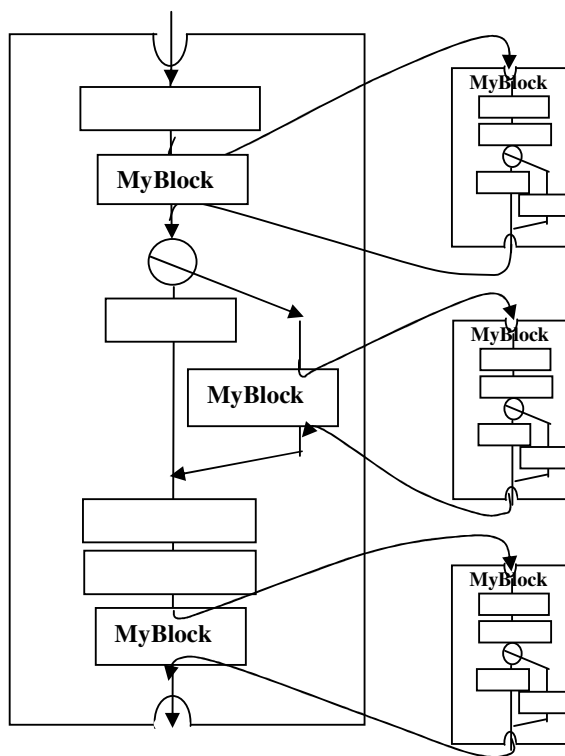
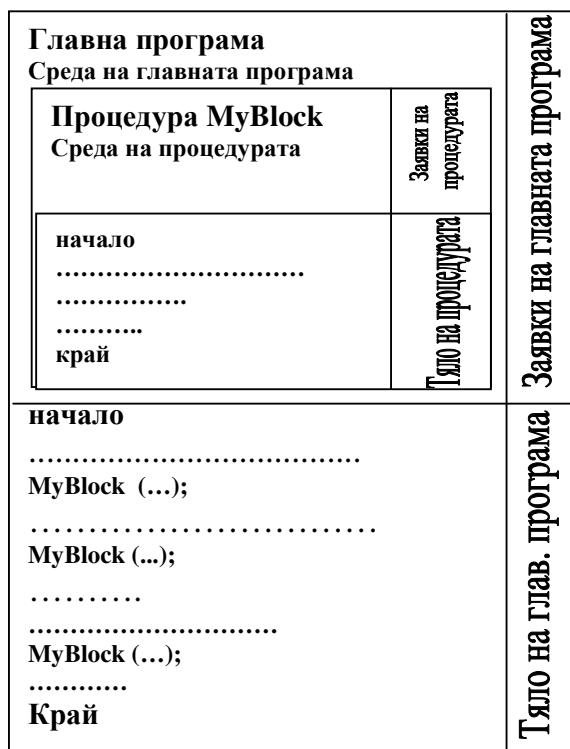
Процедурата е обособен алгоритмичен блок, който се използва при “конструирането” на алгоритъма като “готово блокче”. Независимо от това колко са вградените в един алгоритъм процедури и на колко нива на вграждане са организирани, те в крайна сметка са част от един общ цялостен алгоритъм, който съществува като отделна единица на управление. Тази единица се реализира като независима програма, която се нарича *“главна програма”*.



При вграждането в обхващащ алгоритъм (наричан по-общо “повикваща програма”) процедурата се използва като елементарен (неразградим) алгоритмичен блок, т.е като всеки друг оператор – инструкция за действие. За да може да се реализира на програмно равнище процесът на дефиниране на обособени блокове-процедури и впоследствие – вграждането им в обхващащ алгоритъм, необходимо е:

- Да се зададе *символично име* (идентификатор) на процедурата (според правилата на използвания език за програмиране).
- Да се напише алгоритъмът на процедурата на съответния програмен език, като текстът за описание на процедурата се включва еднократно в текста, предназначен за компилация.
- За изпълнение на процедурата, или както често се казва за “повикването” на процедурата от повикващата програма, се използва зададеното за процедурата име. Името на процедурата се използва по определени правила навсякъде, където това е необходимо.

На следващата схема вляво е илюстрирано как принципно се организира текстът на главна програма и процедура в повечето процедурни езици.



В частта “заявки” на главната програма могат да бъдат описани една или повече процедури. Както се вижда от схемата, структурата на програмния текст на самата процедура е много подобен на този на една програма въобще.

Може да се каже, че при компилиране на главната програма машината “приема за сведение” какво да “разбира” под името “MyBlock”, запомня го и го изпълнява всеки път, когато в хода на изпълнение на главната програма управлението “срещне” това име. Както е илюстрирано горе вдясно, при

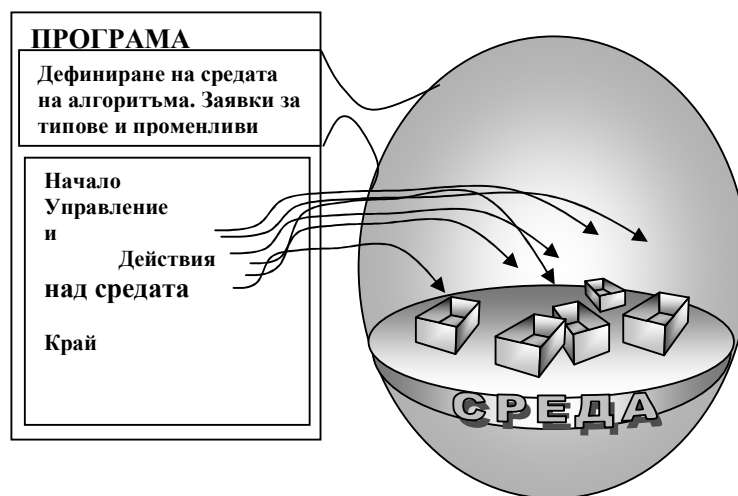
стартирането на главната програма, управлението се поема от нея. Когато изпълнението достигне до повикване на процедура, управлението се предава на процедурата и изпълнението следва съответно нейната схема на управление. При завършване на процедурата, управлението се връща на повикващата я програма, в точката от алгоритмичната схема, намираща се след “повикването”. В тялото на всяка процедура може да се “извиква” друга процедура, обявена с име и описана предварително в същата главна програма. Когато изпълнението на главната програма приключи, всички заявки, включително тези за обявяване на процедурите, престават да се “помнят”.

Използването на процедури съкращава *текста* на главната програма. За горния пример, ако процедурата не беше обособена и дефинирана като отделен блок, на местата, където стои нейното име, трябваше да е написан изцяло (или почти) нейният текст. Използването на процедури прави главната програма по-компактна. Това се използва за съставяне на по-ясни програмни текстове. Не бива да се забравя, че на процедурите се задават добре подбрани имена.

3.2.2 Процедури и обмен на данни

➤ Глобални и локални променливи

Да припомним, че разглеждаме алгоритъма като взаимодействие между среда, управление и действия над средата. При ползването на процедури възниква въпросът за това какъв е начинът, по който си взаимодействат стойности в среда, предоставена на множество относително самостоятелни алгоритми.



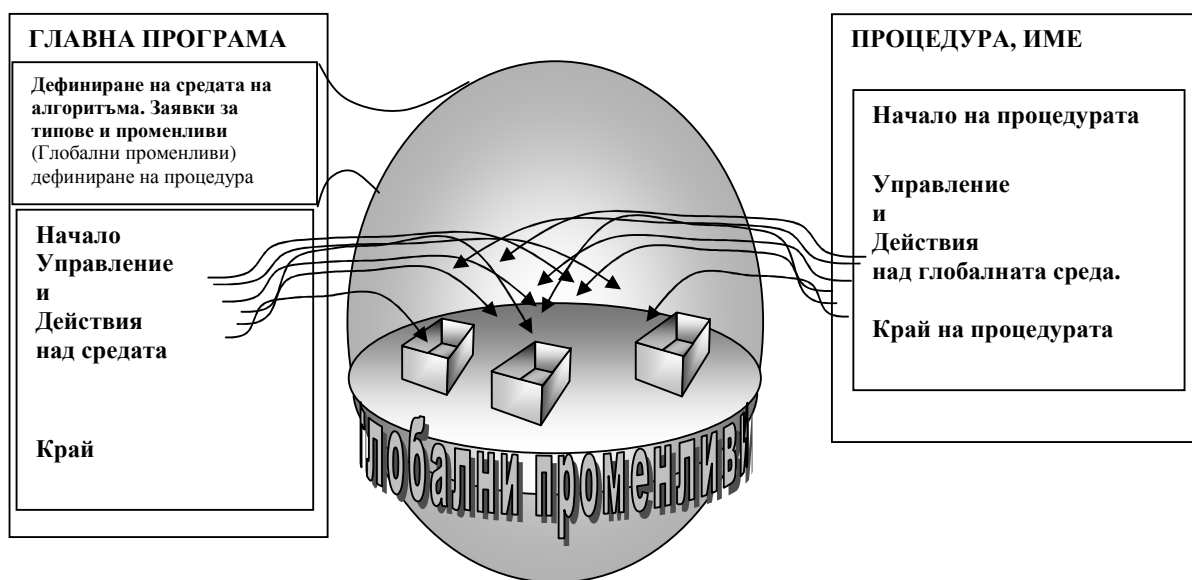
На илюстрацията горе е изобразена абстрактна аналогия, която ще използваме нататък, за да се улесним разбирането на процеса на обмен на данни. Показано е образно как една отделна програма “упражнява въздействие” над собствената си, дефинирана в самата нея среда. Да припомним: всеки елемент на средата има име и е от даден, дефиниран тип. Използването на това име в някой оператор е работа със стойността, записана в този елемент. Програмата “разполага” със “собствената си” среда и всеки потенциално

изпълним оператор в нея има достъп до стойностите, записани в средата, би могъл да ги променя, въвежда, сравнява и т.н.

➤ Глобални променливи

Ако една променлива е “заявена” в главната програма, тя се нарича *глобална променлива*.

Очевидно, самата главна програма може да извършва над средата (си) всички позволени действия. Както е показано на следващата илюстрация, това може да прави и всяка процедура, обявена в главната програма. Всяка процедура може да има достъп за работа с глобалните променливи. По този начин, от гледна точка на получаваните по време на работата на двата алгоритъма резултати, процедурата не работи “изолирано” от главната програма.



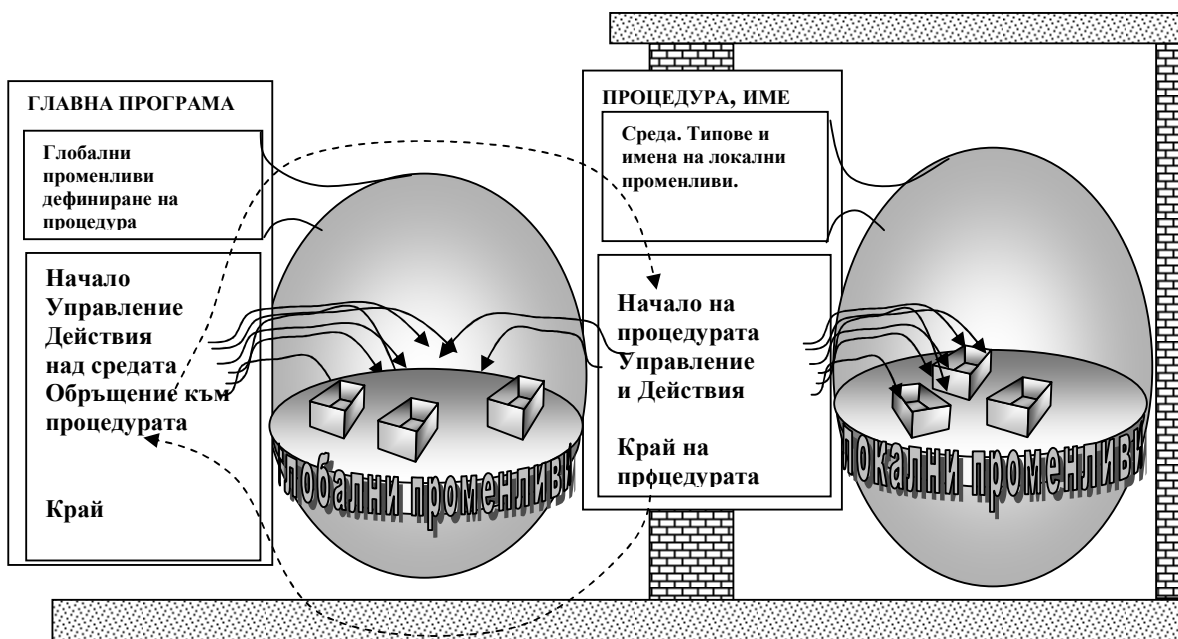
Глобалните променливи могат да послужат за предаване на стойности между главната програма и процедурите. Главната програма предава управлението на процедурата. Процедурата извършва действия направо над глобални променливи и когато управлението се върне отново на главната програма, в глобалните променливи, над които е работила процедурата, има записани нови, променени от процедурата стойности.

Организирането на такъв обмен на стойности – с използване на глобални променливи над които процедурата да работи, е възможност, която не се използва много често. Това е така, защото подобно “безрезервно споделяне” на променливите крие известни опасности.

Променливи, предоставени за “общ” достъп и работа над стойностите им се използват понякога за “предаване на информация” по посока от процедурите към главната програма. Прилага се следната схема: процедурата работи в собствена, независима среда и след като получи резултат, “прехвърля го” на главната програма, като го записва в глобална променлива.

➤ Локални променливи

Процедурата може да има и своя “собствена” среда. Променливите, обявени съответно с име и тип (със съответни заявки) в рамките на процедурата, се наричат *локални променливи*. Достъп до стойностите на локалните променливи има само процедурата.



➤ Обмен при използване само на глобалната и локалната среда

При компилирането на програма с процедури, в машинната памет се организират по определени правила местата и начинът за достъп до всички променливи – и глобалните и локалните. При това локалните променливи “съществуват” само за процедурата, която ги е заявила, а до глобалните имат достъп всички.

Когато процедурата започне изпълнението си, тя извършва действия над обявената за нея локална среда. Тя може да работи и с глобалните променливи, ако съдържа оператори за действия с тях. Така процедурата може да “пренася” направо стойности *от* и *към* глобалната среда.

Когато изпълнението на процедурата завърши, стойностите, получени от нея в локалните променливи се губят (както това става със стойностите от всяка програма след завършването ѝ). Процедурата “пази” само имената и типа (следователно и възможността за работа с тях) на обявените локални променливи до завършването на изпълнението на главната програма.

Имена и типове на локалните променливи

Локалните променливи могат да бъдат от всеки от стандартните типове, или от нестандартен тип, обявен в главната програма. В процедурата могат да се обявяват собствени за процедурата типове, валидни само за нейните локални променливи.

В процедурите могат да се задават имена на локални променливи по достатъчно свободен начин, включително и имена, с които вече са “кръстени” глобални променливи.

➤ Област на валидност на идентификаторите

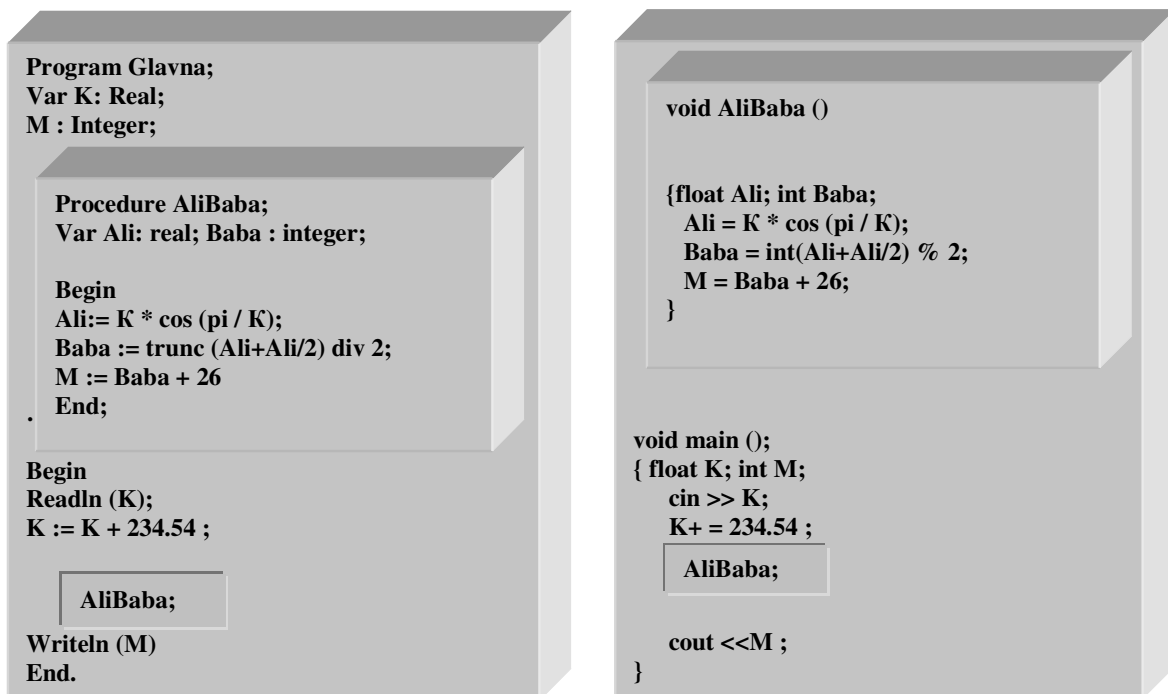
Образно казано, името на променливата служи на програмата като “идентификатор” на порцията памет, в която се записва стойността на променливата. Програмните единици имат достъп до различни обособени среди и могат да работят със стойностите в тях по определени правила. Това разпределяне на достъпа предопределя и каква е областта на валидност на имената – “идентификатори”. Съществува ясна организация за това коя програмна единица какво “подразбира” под дадено име.

Нека една глобална променлива се казва “ a ”, но и една локална променлива се казва също “ a ”. Това означава наличие на две променливи, които носят едно и също име. В такъв случай операторът $a \leftarrow 0$; (например) ще доведе до нулиране на стойността на глобалната a , ако се намира в главната програма. Същият оператор ще доведе до нулиране на локалната a , ако той е в процедурата.

Образно казано, подобно използване на “променливи-адаши” няма да обърка работата нито главната програма (тя работи само с глобалната среда), нито процедурата (тя пък “предпочита” да работи със своето си “ a ”). Объркване би могло да настъпи у програмиста и у всеки, който се опитва да разбере как работи алгоритъма, четейки програмния текст. Поради това, използването на еднакви имена за глобални и локални променливи не е препоръчително.

Долу е даден пример за организация на текст на програма с процедура.

Нека е съставена процедурата “AliBaba”, която има две локални променливи, а променливите K и M са глобални.



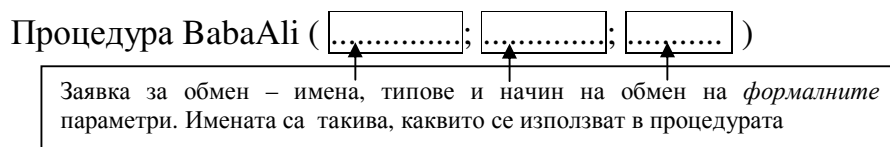
При стартирането си процедурата “взема” текущата стойност на променливата К, извършва посочените в нея действия и записва стойност в глобалната променлива М. Ако в променливата М е имало записана някаква стойност от главната програма (в примера такава стойност няма), тя би била заличена. В този пример смислен алгоритъм всъщност няма, примерът се дава само за илюстрация на организацията на текста, на средата и на обмена на стойности.

Разгледаният дотук начин на обмен на данни между главната програма и нейните процедури може да се организира винаги, дори в някои случаи е наложителен (за икономия). Все пак, при него се забелязва известна липса на организация. Съществуват и по-организирани и дисциплинирани начини на обмен. Най-общо, те са два вида : “*по стойност*” и “*по адрес*”.

И двата “организирани” начина на обмен са реализирани в езиците от процедурен тип по сходен начин. Най-общо, те изискват в повикващата програма и в процедурата видът на обмена и неговите параметри да се задават по определени синтактически правила.

Обявяване на параметрите на обмен става на принцип, илюстриран със следващата примерна процедура BabaAli:

1. При самото описание на процедурата се посочват параметрите на обмен, с техните имена – *локални* за процедурата и техния тип.

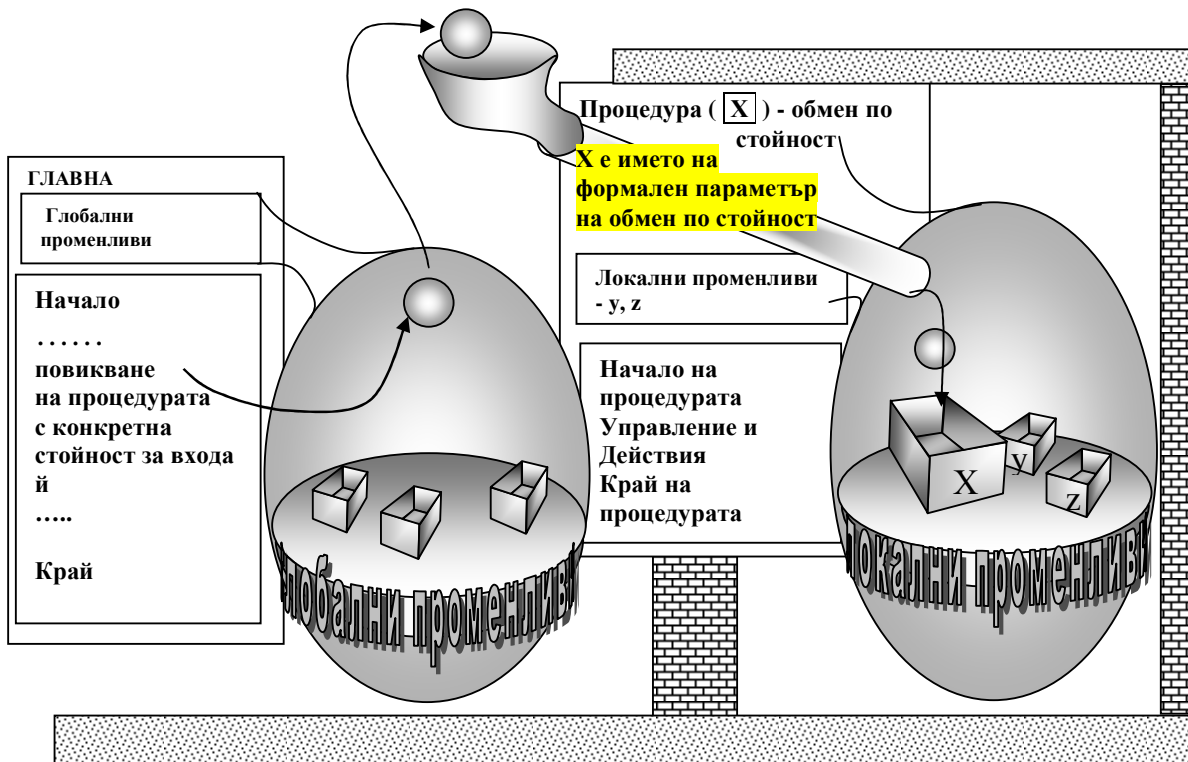


2. За повикване на процедурата, в текста на повикващата я програма, в повикващия оператор се изреждат *фактическите* (за даденото повикване) параметри на обмена, като се спазва съответно реда, в който те са в заглавието на процедурата.

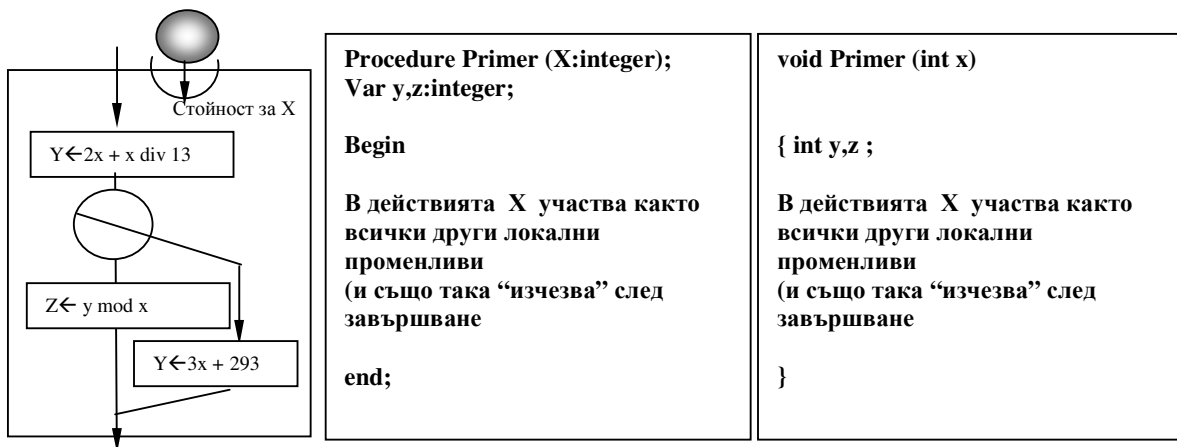


➤ **Обмен по стойност**

Както е показано образно на следващата илюстрацията, обменът по стойност създава нещо като “организиран вход” на стойности към локалната среда на процедурата. Би могло да се каже, че до някои локални променливи е предоставен “канал” за задаване на стойност отвън.



Както личи от фигурата, този вид “внасяне” на данни в средата за работа на процедурата е принципно “по-организиран”, отколкото ползването на глобални променливи. Той е по-безопасен по отношение на евентуални грешки от страна на програмиста. Процедурата не “бърка” по глобалните променливи, а на входа й се задава желаната стойност за X.



Задаването на входната стойност се прави при повикването на процедурата, като в съответния “повикващ” оператор освен името на процедурата се посочва с каква фактическа стойност на входния си параметър тя да работи. При обмяна по стойност, на входа на процедурата се подава израз от съответния тип. Всъщност, предава се стойността – резултат от изчислението на изказа. Ето примери за оператори за “повикване”:

Primer (3);

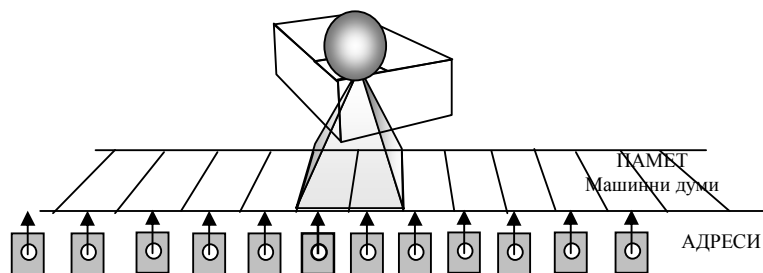
Primer (A); (A е глобална променлива от типа на X, чиято стойност се копира в X)

Primer (A + 2);

От илюстрацията – схема за обмен по стойност се вижда, че този обмен е еднопосочен. От “повикващата” програма се задават стойности за работа на процедурата. Резултатите от работата на процедурата биха могли да бъдат “върнати”, “съобщени” на главната програма (и с това “доведени до знанието” на всички интересувачи се процедури), например като се запишат в глобални променливи. Съществува и друг начин за реализиране на “обратната връзка” – от процедурата към повикващата я програма.

➤ Обмен по адрес

Нека засега се ограничим с една начална представа за организацията на работата на програмата с машинната памет. Стойностите на променливите се записват в машинната памет, която е организирана в единици памет – машинни думи. Всяка стойност на променлива се записва в определен брой машинни думи, в зависимост от типа променливата. За да се работи със стойността на дадена променлива, се помни адресът на началната дума, в която “е отредено” да се записва променливата. Всеки път, когато трябва да се извършат промени над стойността, машината “намира” адреса, на който е записана променливата, и работи с намиращия се там кодиран запис.



При използването в това учебно помагало програмни средства, определянето на това коя променлива на кой адрес да бъде записана става от компилатора без участието на програмиста. Съществуват различни схеми за “отреждане на адреси за променливите”, с които ще се запознаваме постепенно. При всички положения, адресирането и достъпът до адресите са организирани по точно определени правила, които не се виждат непосредствено в текста на програмата.

На следващата фигура е илюстриран принципът на обмен между главна програма и процедура чрез използване на адреса, на който стойността на една променлива е записана.

Този обмен работи на базата на следната схема на адресиране: Мястото на всяка глобална променлива е адрес, който се определя при компилирането и се помни до завършването на изпълнението на главната програма. Що се отнася до

процедурата, при всяко нейно повикване локалната ѝ среда се “разполага” на ново място в паметта. При това, всяка локална променлива получава временен адрес. Извикващата програма предава данни на процедурата, като записва стойности в нейната локална среда. Процедурата може да разглежда получените стойности по два начина. Единия от тях вече разгледахме – това е непосредствената обработка на получените данни, т.е. предаване по стойност.

При втория начин (предаването по адрес) процедурата приема полученото не като стойност за непосредствена обработка, а като стойност на адрес, т.е. място от паметта, със съдържанието на което да оперира.

Важно за тази схема на пренос е, че тя предоставя възможност да “се върне” информация от процедурата към повикващата програма. От илюстрацията е показано, че обменът по адрес може да бъде използван в двете посоки: от главната програма към процедурата и от процедурата към главната програма.

Да напомним – след прекратяване на изпълнението на процедурата, стойностите на локалните променливи се губят. Преносът на необходимата за главната програма информация може да стане или чрез записване на определени стойности в глобални променливи, или чрез организиран обмен по адрес.

Ето как изглежда на Pascal и C++ обявяването на процедура с формален параметър X за обмен по адрес:

Procedure Prim (**var** X: integer);

void Prim (**int&** X)

Използването на горният синтаксис при описанието на формалния параметър X е указание към компилатора да генерира изпълним код, в който променливата X от текста на процедурата се разглежда не като локална стойност от съответния тип, а като съдържание на адрес на такава променлива. По тази причина, фактическият параметър при повикване би могъл да бъде единствено и само *променлива* от същия тип. Например:

Prim (G);

където G е име на променлива от целочислен тип.

Но не Prim (2); !

В заключение трябва да кажем, че една процедура може да е замислена да използва всички изброени начини на обмен на данни.

1. Явно организирано – чрез обмен по адрес или чрез обмен на стойности.
2. ”Неявно” (но организирано от програмиста) – чрез използване на глобални променливи.

Най-общо:

- Извикващата програма предава на извикваната процедура стойности.
- Извикваната процедура ползва получените стойности непосредствено (предаване по стойност) или ги разглежда като адреси на стойности (предаване по адрес).
- Текстът на процедурата може да съдържа имена на глобални променливи, описани преди описанието на самата процедура. Тогава компилаторът

вгражда в изпълнимия код непосредствени действия с тези променливи. Това е още един начин за обмен на данни с процедурата.

- Предаването на данни по стойност е в посока от викащата програма към процедурата.
- Предаването на данни от процедурата към главната програма става чрез използване на глобални променливи или с обмен по адрес.

От изложеното дотук може да се заключи, че процедурите дават възможност за конструиране на компактни програмни текстове и са достатъчно гъвкави по отношение на взаимовръзките си със средата, което ги превръща в едно добро средство за структуриране на програми.

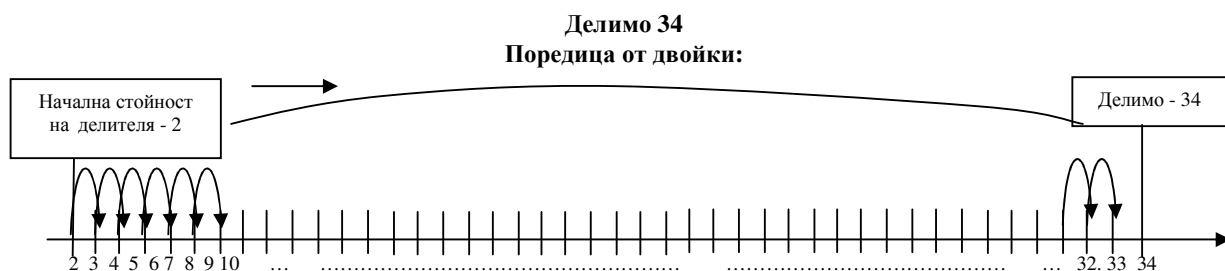
◆ Примерна задача за упражнение

Да се спрем на задача, в която алгоритъмът на Евклид се използва като процедура.

Задачата е да се установят “експериментално” *двойки числа* – делимо и делител, за които алгоритъмът на Евклид прави *най-много стъпки* при намиране на техния НОД.

За установяването на такива двойки може да се приложи следната схема: за едно и също делимо се избира начален делител, намира се НОД, след това се увеличава делителят с единица, намира се НОД и така нататък, докато делителят не достигне до стойността на делимото. За всяка получена по тази схема двойка “делимо-делител”, трябва да се разпечатват стойността на НОД и броят стъпки, за които този НОД е бил намерен. Така двойките с най-голям брой стъпки се виждат от разпечатката¹⁵.

По-конкретно, нека този изчислителен експеримент да работи по описаната обща схема, но последователно за две различни делими, както е илюстрирано на следващите две фигури:



Образуват се последователно двойките : (2, 34); (3, 34); (4, 34); ... (32, 34); (33, 34) и за всяка се извежда на екран НОД и броят стъпки, за който НОД е намерен.

¹⁵ При анализ на сложността на алгоритъма на Евклид се доказва, че изпълнението му е най-дълго за двойки – съседни числа от редицата на Фибоначи.

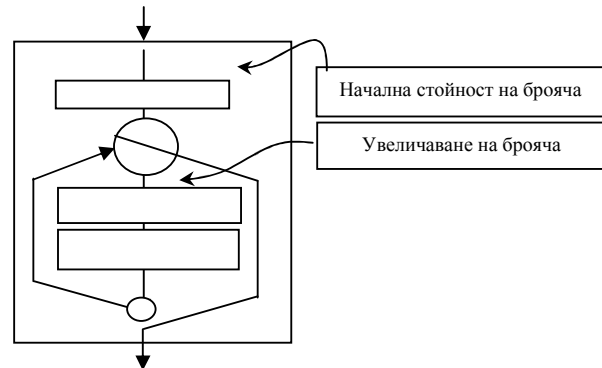


Образуват се последователно двойките : (2, 55); (3, 55); (4, 55); ... (53, 55); (54, 55) и за всяка се извежда на екран НОД и броят стъпки, за който НОД е намерен. Помислете дали е необходимо стойността на делителя да расте до стойността на делимото.

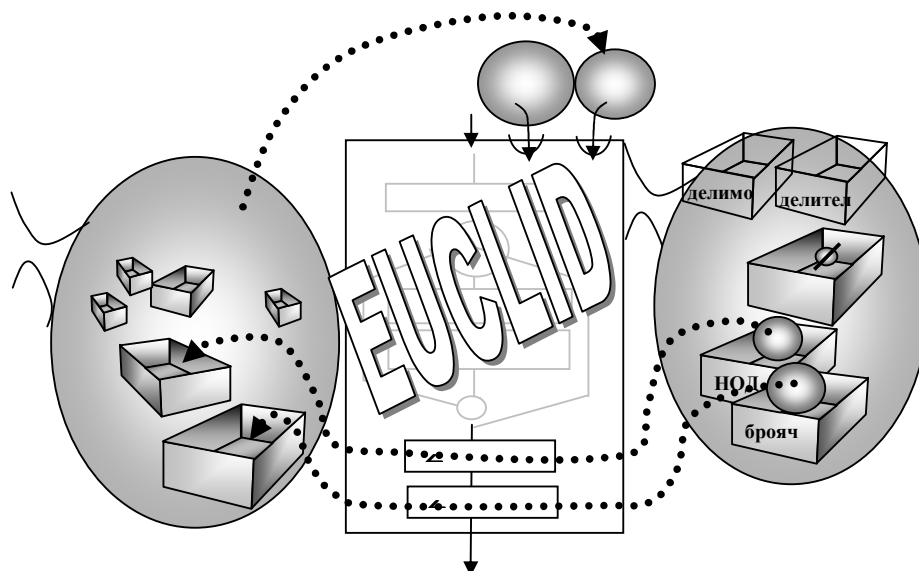
Използването на алгоритъма на Евклид като процедура за целите на решаването на тази конкретна задача може да стане по различни начини.

Ще дадем помощ под формата на схеми, над които може да се разсъждава, като подчертаваме, че тези схеми са съставени за примерен начин за решаване.

Първата схема дава помощ за това как в алгоритъма на Евклид да се вгради брояч, който да отброява колко пъти се изпълнява основният цикъл – този за намиране на остатъка от делението (докато не се окаже, че остатъкът е нула).



Втората схема дава идея за това как биха могли да се пренесат в главната програма получените стойности за локални променливи, без да се ползва обмен по адрес.



3.2.3 Функция. Основни понятия

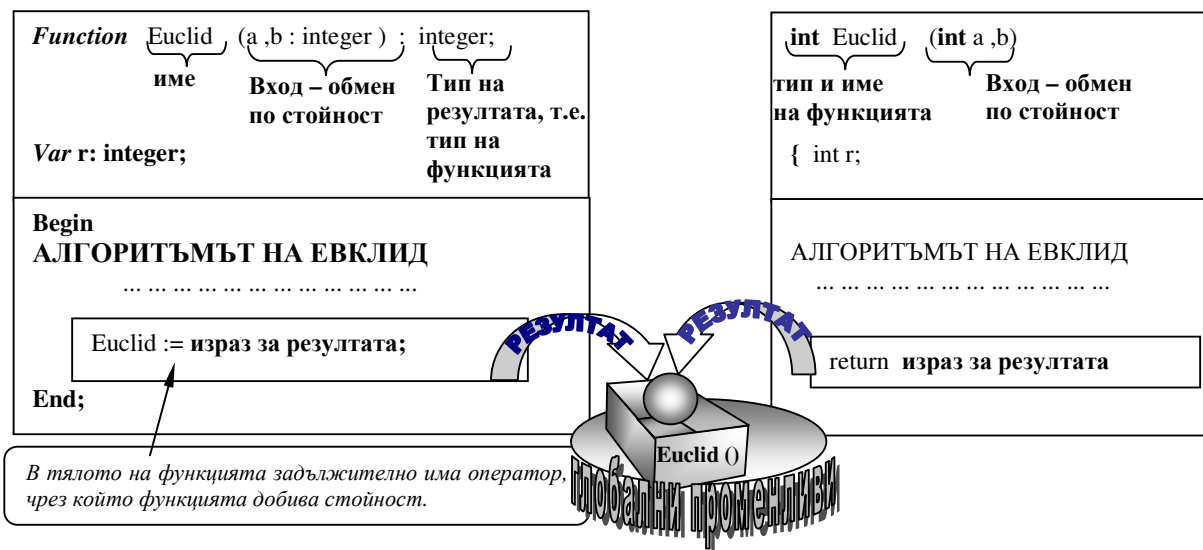
В синтаксиса на различните езици за програмиране за означаване на факта, че един програмен блок е организиран като процедура, се използват различни “елементарни думи” – *procedure*, *function*, *subroutine* и т.н. Това не променя принципните положения, изложени в предходната точка. В много от езиците съществува и един специфичен вид процедури, които тук ще наричаме “функции”. В математиката под функция се разбира преобразуване, което от един или повече аргументи получава *единствен резултат* – *стойност на функцията*. Процедурите работят по подобен начин – за работата им се организира *вход* и *изход*. Ако една процедура е замислена и съставена така, че от работата ѝ да съществува единствен резултат, заявен като тип и “връщан” по организиран начин в главната програма, ще я наричаме “функция”.

На фигурата долу е илюстрирана принципната постановка, според която на изхода на функцията задължително има и един резултат – стойност, която е третирана по специфичен начин при обмена с главната програма.

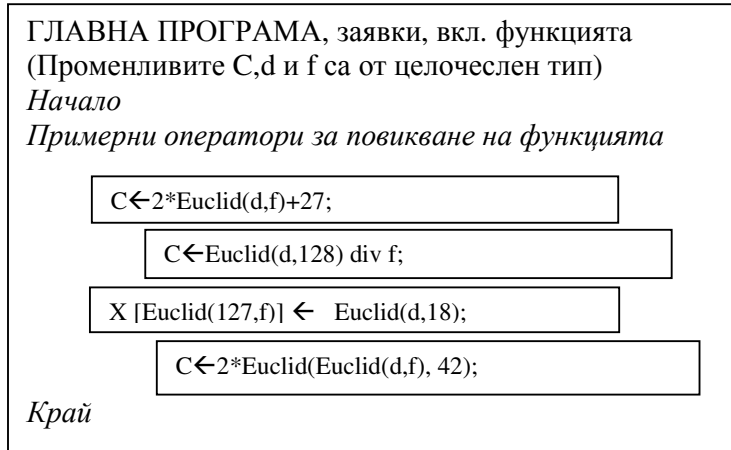


Тази получена от работата на функцията стойност има тип, който се нарича “*тип на функцията*”. Всъщност става въпрос за типа на резултата от работата на функцията, който е “връщан” като стойност в главната програма и е “на разположение” в глобалната среда.

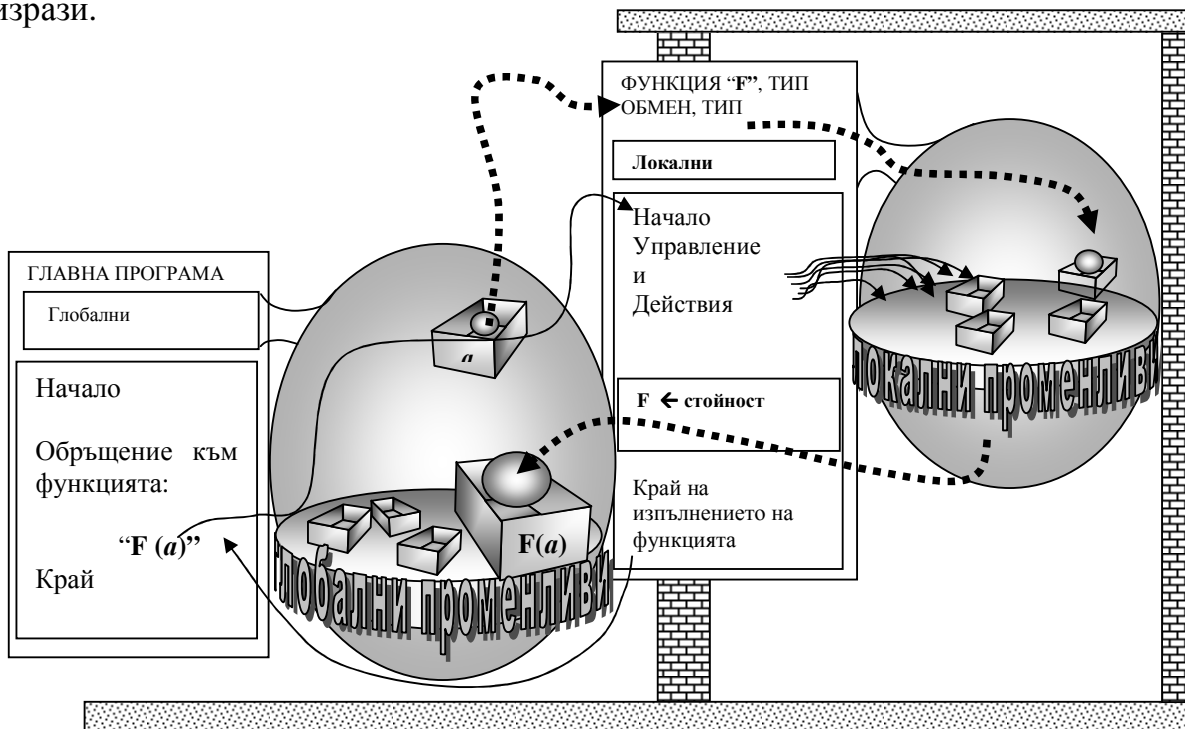
“Същественият” резултат от работата на функцията се заявява като *тип на функцията* още при нейното дефиниране, както е илюстрирано долу с обичайно използваните тук две “езикови схеми”. Примерът е с познатия алгоритъм на Евклид, организиран като функция с вход две цели числа и единствен резултат – най-големият общ делител на двете числа



Функциите са вид процедури и всичко, което беше изложено дотук за процедурите, се отнася и за функциите. Предвид факта, че функцията е вид процедура, тя би могла да ползва паралелно всички други начини на обмен с глобалната среда. Тук ще се спрем само на допълнителните особености на функциите.



Спецификата на обмена с повикващата програма е в основата на една характерна особеност на функциите, а именно – начинът на тяхното повикване от главната програма. Функциите се повикват с името си, както всички процедури, но те могат да бъдат “вградени” в израз. Това изисква обръщението към функцията, т.е. резултатът от нейното изпълнение да стои от дясната страна на оператор за присвояване на стойност или да участва в какъвто и да е израз. Това е илюстрирано на фигурата горе с четири примерни оператора за повикване на функцията Euclid, където обръщението към функцията участва в изрази.



На предходната фигура е илюстрирано повикване на функцията F , която има единствен параметър на обмен по стойност и е повикана с фактически параметър – стойността на глобалната променлива a . Можем да си представяме, че в глобалната среда за стойността на F е отредена една отделна променлива, която носи името на самата функция.

Когато при изпълнение на главната програма се достигне до обръщение към функцията F , тя се стартира със съответните аргументи, според предвидения обмен. След приключването на работата ѝ, в главната програма се връща изчислената стойност на функцията и се преминава към изпълнение на следващото действие.

➤ **Примери за алгоритми, използващи процедура – функция**

3.2.4 Понятие за числен метод и числово решение

Предвид факта, че настоящото учебно помагало е предназначено да даде начална представа за алгоритмичните подходи при решаването на някои основни категории задачи, възникващи в практиката на програмиста, в тази тема ще се спрем на една класическа задача от областта на числените методи, а именно,

числово решаване на уравнението $f(x) = 0$.

Това ще ни позволи да проследим както начина на използване на функции на програмно равнище, така и постановката, подхода и алгоритмизирането на една много често възникваща в практиката задача. Нека направим едно важно разграничение: числовото решаване на уравнение се различава съществено от намирането на аналитично решение на уравнението. Нашият първи пример за алгоритъм, съставен за решаване на уравнението: $f(x) = 0$ беше за функция

$$f: ax^2 + bx + c$$

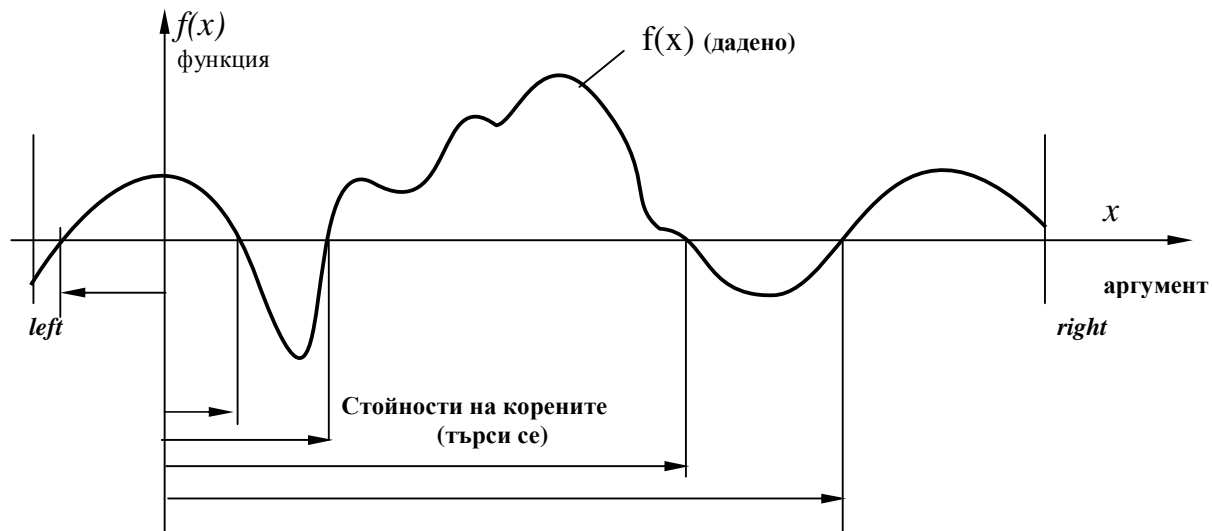
Тогава подчертахме, че търсим решението *аналитично*, т.е. търсим стойностите на неизвестните при използване на известни математически изрази и на методика, която позволява да се състави алгоритъм за директната, аналитично изведена връзка между коефициентите a , b и c , които са вход на задачата, и нейния изход. На практика, съставеният преди алгоритъм “имитира” решаване на тази задача от човек, който знае как се решава квадратно уравнение и са му известни формулите, в които трябва да замести получените на входа стойности.

➤ **Постановка на задачата**

Нека е възникнала необходимост да се намерят нулите на функцията $f(x)$ в някакъв интервал (*left, right*), т.е. онези стойности на x в този интервал, за които е в сила равенството $f(x) = 0$. (Функцията $f(x)$ е известна функция.)

Нека за корените на уравнението $f(x) = 0$ няма изведени аналитични зависимости. Следователно, за уравнението не може да се намери *точно* в математически смисъл решение. Задачата е да се намери *приближено решение*, т.е. да се намерят нулите на функцията с някаква приемлива точност. Каква да

бъде точността, това зависи от конкретната задача и се определя от *допустимата грешка* на търсеното решение.



Задача:

Дадено:

$f(x)$ – известна функция, дефинирана в интервала $(left, right)$

Δ – допустима грешка – някакво предварително зададено малко число.

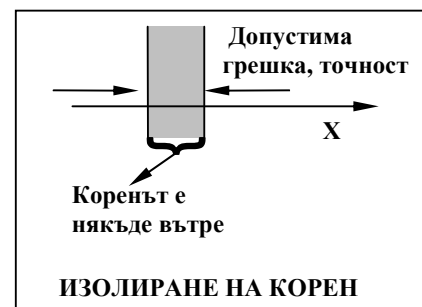
Търси се:

За кои стойности на аргумента x ,

$$f(x) = 0,$$

т.е. търсят се (приближено) корените на горното уравнение в интервала $(left, right)$ с предварително зададена допустима грешка Δ .

Според постановката на задачата за числово решаване на уравнението, нулите на функцията $f(x)$ се търсят с някаква допустима грешка. Това ни дава основание да използваме термина "изолиране на корен с някаква точност". На фигурата вдясно е илюстрирано какво ще разбираме под "изолиран корен".

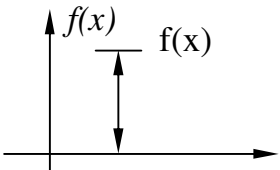
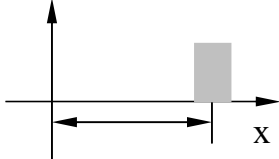


Нека е възможно да се установи със сигурност, че в интервал с големина, равна на допустимата грешка, функцията $f(x)$ има нула. Нека прилаганата методика позволява да се установи какви са лявата и дясна граница на "интервалчето", в което коренът е изолиран. Ще считаме, че с това задачата за намиране на този корен е решена.

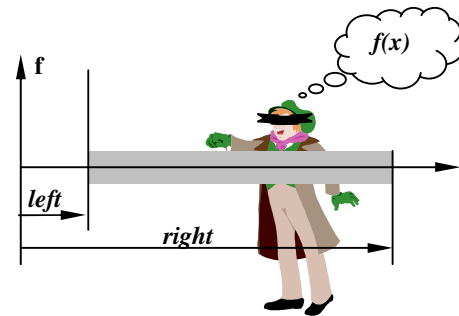
За да улесним разбирането на принципната постановка на задачата за числово решаване на уравнение, ще приведем някои допълнителни схеми, придружени с коментар.

Очевидно нулите на $f(x)$ са точките, в които графиката на $f(x)$ пресича оста x . Още по-точно, стойността на всеки корен е равна на стойността на координатата по x на пресечната точка на графиката на функцията с абсцисната

ос. Графиката на функцията не е известна нито на алгоритъма, нито на програмата. Това, което може да бъде изчислено с машината е стойността на функцията f за произволен аргумент от интервала $left, right$. Нека изразим това, което е изчисляемо и това, което се търси, по следния начин:

<p>Може да бъде изчислена : големината на $f(x)$ във всяка точка от интервала $left, right$</p> 	<p>Търси се (с някаква точност) в кои точки $f(x)$ е нула</p> 
---	--

На рисунката вдясно образно е илюстрирана тази постановка на задачата. Търсенето на числово решение е оприличено на “сляпо изпробване на аргументи от интервала”, придружено с изчисляване на стойността на функцията за всеки аргумент.

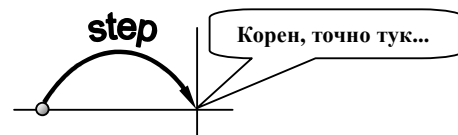


Това схематично представяне на задачата лесно навежда на интуитивно най-очевидния начин да бъдат изолирани корени, а именно – да се “претърси” на малки стъпки целия интервал.

3.2.5 Метод на “сканиране на интервала”

Образно казано, сканирането означава един обект (сканираният) да бъде “нарязан” на части и да бъде “обработен” част по част, отначало... докрай. Методът на “сканиране на интервала” най-общо може да се представи като процес на последователно намиране на нови “пробни” стойности за аргумента x , отстоящи една от друга на точно определена стъпка (*Step*).

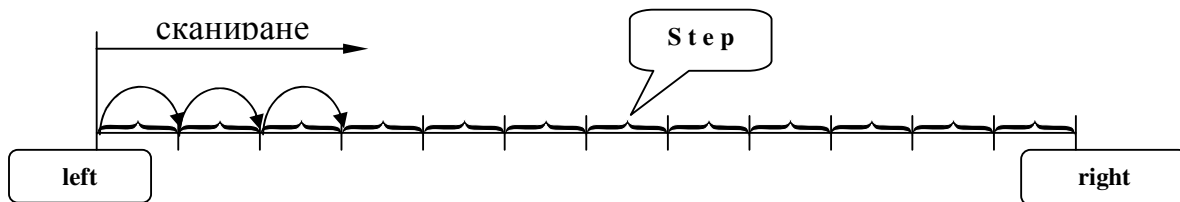
На пръв поглед може да се помисли, че методът на сканиране се базира на “точно попадение” в корен. Несъмнено е възможно сканирането да напредва на много малки стъпчици, а това поражда погрешната представа, че алгоритъмът има за цел да установява кога $f(x)$ става *точно равна на нула*.



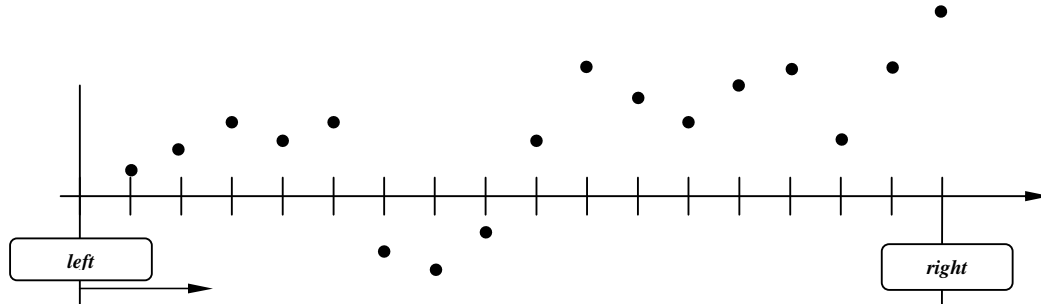
Това е много лоша идея. Първо, тя допуска, че някакъв интервал от реални числа може да бъде обхождан точка по точка, при това с краен процес. Второ, тя предполага, че множеството на реалните числа, от което е по идея аргументът x , има адекватно машинно представяне. Трето, тя допуска, че изчисленията в множеството Real са толкова прецизни, че програмистът може да си позволи да базира алгоритъма на проверка за равенство на реална стойност с число. (Подчертавахме вече, че това не трябва да се прави!).

Всички тези разсъждения довеждат до извода, че не е възможно машинен алгоритъм да се базира на предположението за точно попадение в корен, въпреки че по принцип не е изключено процесът на сканиране по случайност да попадне точно в корен.

Да се сканира интервала ($left$, $right$), в който търсим нулите на $f(x)$ означава интервалът да бъде “нарязан” на малки интервалчета, равни на “стъпката на сканиране” ($Step$). Всяко интервалче-стъпка се проверява (по някакъв начин) за наличие на корен. Ако за дадена поредна стъпка се установи наличие на корен, счита се, че в това интервалче-стъпка е *изолиран един корен*. Процесът на това постъпково претърсване се развива по ос x , систематично и без прескачане, от единия другия край на интервала.



Функцията $f(x)$ е известна функция и нейната стойност може да бъде изчислена чрез просто заместване на всяка “стъпкова” стойност на аргумента, получена при “напредването” на по ос x . Можем да си представяме, че сканирането работи на базата на информация от следната обща картина:



Визуално, тази обща картина, без да позволява да се установят точните стойности на нулите на $f(x)$, с цената просто на многократно изчисляване на стойността на функцията в n точки, дава едно добро приблизително “познание” за това къде “горе-долу” са корените. За щастие, многократните изчисления за стойността на $f(x)$ се извършват от машината. Бързо. Например така:

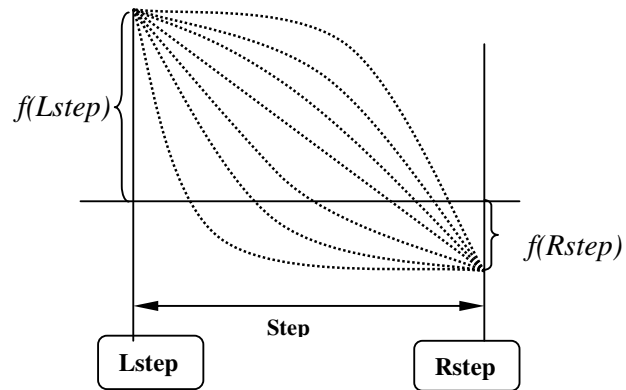


Остана неизяснен въпросът за това как при сканирането да се установява, че вътре в рамките на дадено интервалче-стъпка има корен. Да припомним, че стойностите на функцията в двата края на стъпката са известни.

Нека “пробните” стойности на аргумент “напредват” със стъпка $Step$ и получаваното за всяка стъпка интервалче има лява и дясна граници съответно $Lstep$ и $Rstep$, както е илюстрирано на следващата схема.

Съществува една теорема от математическия анализ (теорема на Болцано), смисълът на която ще преразкажем в контекста на нашата задача:

Ако една функция е непрекъсната в даден интервал (например този с големина $Step$) и стойността ѝ в единия му край е положителна, а в другия – отрицателна, то тя задължително пресича оста x вътре в рамките на интервала.

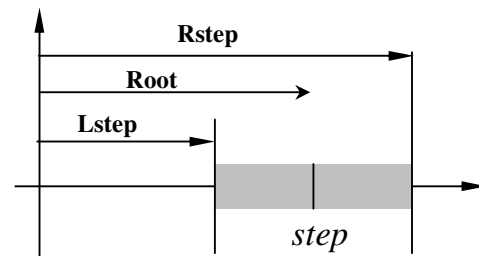


На тази теорема се основава логиката на алгоритъма – принципът на проверката за наличие на корен в дадено интервалче-стъпка. За всяка такава стъпка се проверява за различие на знаците на двете крайни стойности на функцията. Твърдението от теоремата би могло да се окачестви като “очевидно” и дори да се мисли, че прилагането на тази проста проверка в алгоритъма е логично, сигурно и безобидно. Методиката при съставяне на алгоритми от този тип изисква дори “очевидните неща” да бъдат внимателно теоретично проверявани и обосновавани. По-точно, според цитираната теорема, функцията пресича оста в рамките на интервала веднъж или нечетен брой пъти. Следователно, проверката за различие на знаците на функцията в двата края на една стъпка не може да се използва като индикация за наличие на корен(и), ако функцията има четен брой нули в границите на тази стъпка. Откъдето следва, че така описаният процес на сканиране на интервала може и “да пропусне” корени. Това, което е сигурно, е следното: ако процесът на сканиране попадне на интервалче, в рамките на което функцията *сменя знака си, то в това интервалче има поне един корен*.

Ще приемаме следното: ако стойностите на функцията $f(x)$ в двата края на една стъпка са с различен знак, то в рамките на тази стъпка функцията има една нула. Ще приемаме също, че приближената стойност на нулата (коренът $Root$) е в средата на интервала, т.е. :

$$Root = (Rstep + Lstep) / 2.$$

При тази постановка, точната стойност на една нула на функцията би могла да е отдалечена (вляво или вдясно) от изчисления приблизителен резултат най-много на $Step/2$. Така се определя и каква да е големината на стъпката на сканиране – в зависимост от зададената по условие допустима грешка $Delta$. За жалост, при зададена стъпка на сканиране няма гаранция за това дали процесът



на сканиране е “уловил” всички интервали, в които функцията има нули и дали вътре в рамките на стъпка с “регистрирана нула” има повече от една нули. Принципно, колкото е по-малка стъпката *Step*, толкова по-точен общ резултат дава методът и с толкова по-голяма точност се изолират нулите.

Изложените съображения дават интуитивна представа за смисъла на термина “грешка на самия числен метод”.

Въпреки направените уговорки, този метод е практически ефективен и дава напълно удовлетворителни резултати при решаването на голям брой задачи. Към него могат да се добавят редица подобрения и неговото прилагане се препоръчва винаги, когато се очаква функцията да има много нули в дадения интервал и дори когато поведението на функцията в дадения интервал не е предварително анализирано.

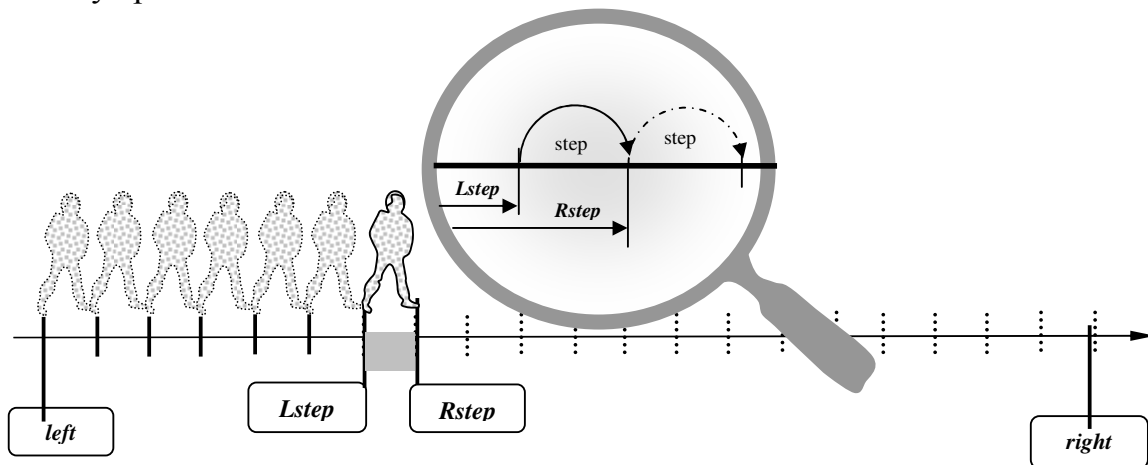
➤ Съставяне на алгоритъм за метода на сканиране

А. Схема на метода и определяне на елементите на средата на алгоритъма

При съставяне на алгоритми, базирани на някаква теоретична постановка и метод, първо се съставя подробна принципна *схема на метода*, като на нея се означават със съответни имена елементите, които са необходими за работата на алгоритъма. Това е необходимо условие за дефинирането на средата на алгоритъма, т.е. на променливите, с които той работи. Точно тези означени на схемите имена се използват по-нататък в алгоритъма и програмния текст. Това прави възможно и леко проследяването и на общата логика на алгоритъма, и на работата на програмата. То позволява и осъществяването на адекватни промени, подобрения или корекции впоследствие.

Нека опишем словесно предлагания алгоритъм така:

Интервалът, в който се търсят нулите, се обхожда на стъпки от левия до десния му край.



За всяка стъпка се прави проверка за еднаквост на знаците на функцията в краищата на стъпката и се “докладва” дали в рамките на стъпката функцията има нула или не. За стойността на нулата се приема средата на стъпката, при което се допуска грешка, не по-голяма от половината стъпка.

Да дефинираме напредването по стъпки на процеса на сканиране още така:

Текуща стъпка: проверка за нула в интервалчето-стъпка, стъпка напред.

Както се вижда от фигурата, преминаването към следваща стъпка изисква просто да се изместят границите на текущата стъпка, като лявата граница заеме стойността на дясната граница, а стойността на новата дясна граница се произчисли.

На схемите са означени графично елементи, съответстващи на елементи от средата на алгоритъма както следва:

Граници на интервала – *left* и *right* – две реални променливи, входни величини,

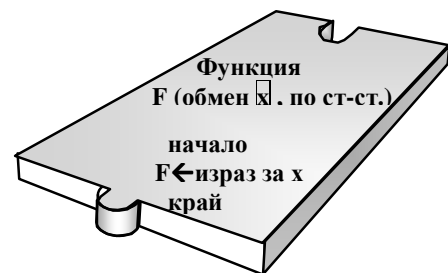
Граници на текущата стъпка – *Lstep* и *Rstep* – две реални променливи,

Големина на стъпката на сканиране – *Step* – една реална променлива.

Големината на стъпката *Step* се изчислява от допустимата грешка на решението. Най-лесно е да се приеме направо, че стъпката е равна на удвоената допустима грешка, зададена на входа на алгоритъма. Може и да се приеме, че стъпката трябва да се нанася цяло число пъти в интервала, подлежащ на сканиране. В този случай се налага в алгоритъма да бъде осигурено и произчисляване (намаляване) на големината на допустимата грешка (големината на интервала се разделя на зададената допустима грешка, закръглява се към най-близкото по-голямо цяло число и после интервалът се дели на така полученото число). Принципно, произчисляването на допустимата грешка не може да доведе до завършване на сканирането със “стъпване точно в десния край на интервала”. Това се дължи на факта, че изчисляването на границите на всяка следваща стъпка става със събиране на числа с плаваща запетая, което се прави от машината винаги с грешка. При това, в нашия случай грешката се “натрупва”, т.е. всяко следващо “стъпване” става на базата на вече грешно изчислен резултат от всички предишни стъпки. Изводът, който произтича е, че процесът на сканиране не може да има условие за край, което да разчита на точно съвпадение на края на последната стъпка с края на интервала (равенство на две реални стойности – нали по принцип така не се прави!). Нещо повече – ясно е, че с *отброяване на стъпките* (от едно до пресметнатото общо количество стъпки в интервала) интервалът не може да бъде обходен съвсем точно – докрай. Оттук следва, че не е разумно обхождането на интервала да е реализирано в алгоритъма чрез цикъл по брояч.

В допълнение на разгледаните дотук променливи, алгоритъмът ползва и изчислените стойности на една функция. Ще подходим по най-логичния начин и ще организираме функцията $f(x)$ като процедура-функция, която има за вход един аргумент и връща на повикващата я програма резултат – изчислената стойност на функцията.

Програмно, тази функция е съвсем елементарна – тя съдържа само един оператор: на *F* се присвоява съответен израз (аналитичният вид на избраната функция, разписан по синтактическите правила на езика за програмиране).



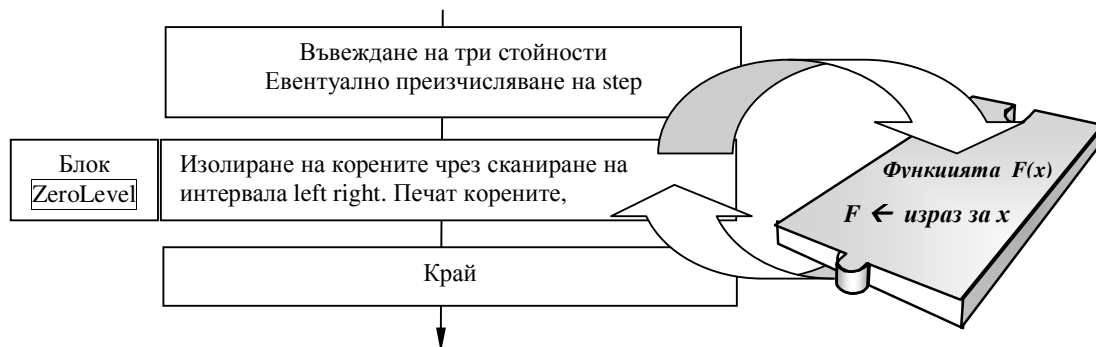
Изразът за F зависи от аргумента X , който е единствен формален параметър на обмен по стойност на така обособеното алгоритмично блокче.

В допълнение на изложеното за средата на алгоритъма, нека обърнем внимание на една програмистична “хватка”, която се прилага често: *За да се установи дали две стойности имат еднакъв знак, проверява се какъв е знакът на произведението им.* Очевидно, ако произведението е равно на нула, поне едната стойност е нула, а ако е отрицателно, то те са с различни знаци.

Б. Съставяне на алгоритъма по методика “отгоре надолу”

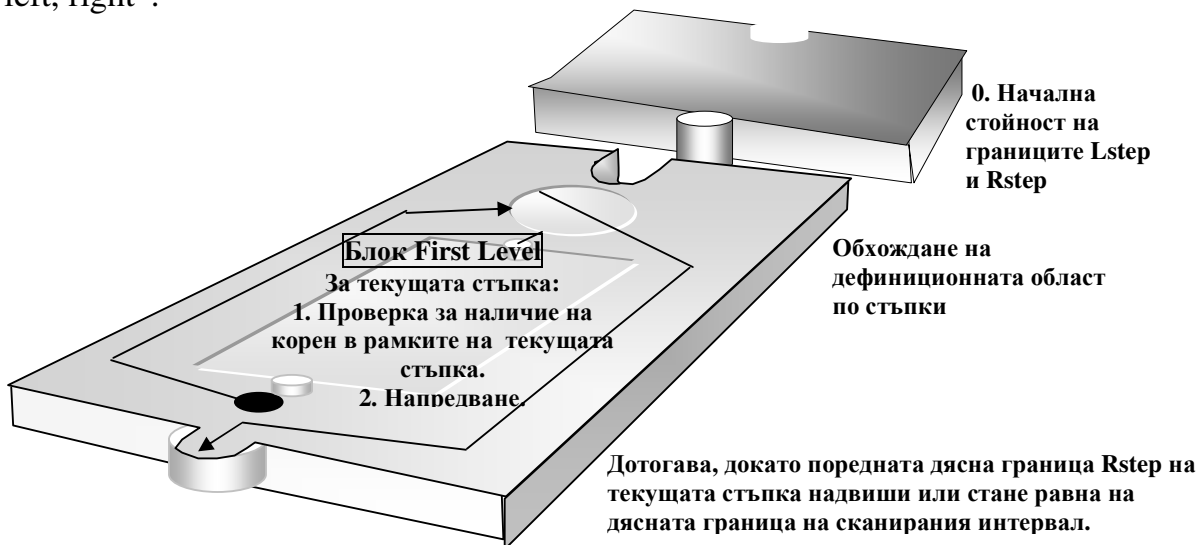
На следващите схеми е показана последователността при прилагането на методиката “отгоре надолу” за съставянето на този конкретен алгоритъм.

0 : Ниво алгоритмична задача. Вход и изход на задачата:



I. Първо ниво на детайлизация

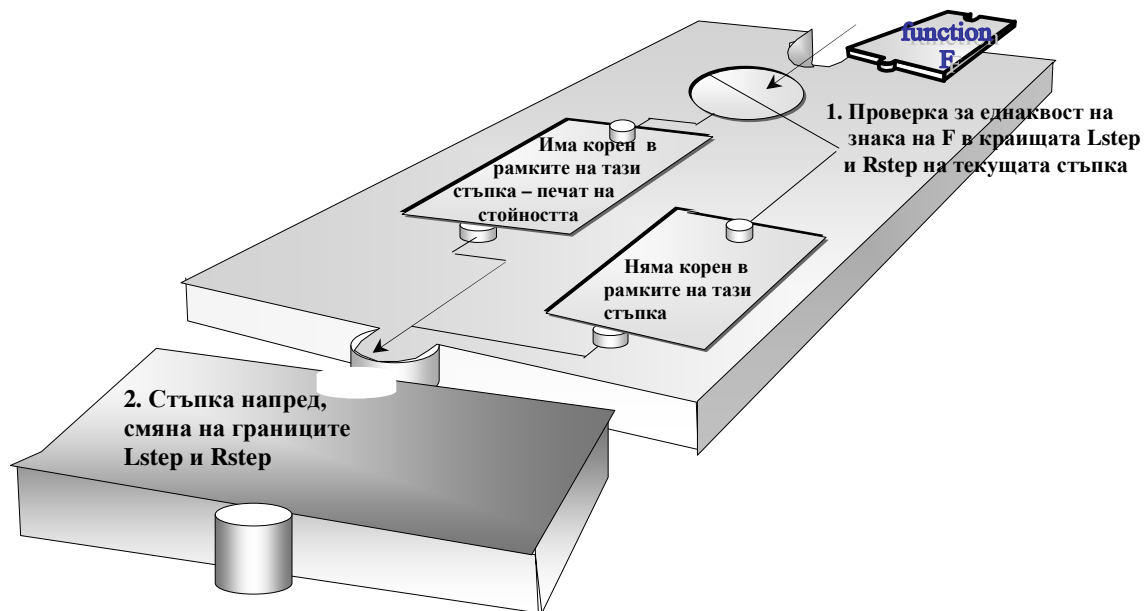
Разграждане на блок ZeroLevel “Сканиране от началото до края на интервала left, right”:



II. Второ ниво на детайлизация. Разграждане на блока “FirstLevel”

В рамките на текущата стъпка :

1. Проверка за наличие на нула на функцията,
2. Преместване на границите на интервала-стъпка с една стъпка напред.



С това всички действия и оператори в получените блокове са изразими на програмен език от нивото, което ползваме в настоящото учебно помагало, и процесът на детайлизация е завършен. Получената схема на управление на алгоритъма може да кодира.

Несъмнено над тази основна схема на алгоритъма могат да бъдат извършвани промени и допълнения. Например, може да се направи така, че никога да не се налага функцията F да се изчислява с аргумент, надвишаващ дясната граница *right* на интервала “дефиниционна област” (явно сега това може да се случи на последната стъпка). Също така, може да се направи подходящо допълнение към средата на алгоритъма, за да могат стойностите на корените да се “съхраняват” записани, а не да бъдат само извеждани на печат.

Нека сега се спрем на още един числен метод за решаване на уравнение, който се базира на много сходна постановка и за който алгоритъмът се съставя по подобен начин – по методиката “отгоре надолу”.

3.2.6 Дихотомично решаване на уравнение

Това е вторият числен метод за решаване на уравнение, на който ще се спрем в тази тема, защото неговата алгоритмична реализация ползва по естествен начин функция.

Разглеждането на този числен метод е от съществено значение в курс по алгоритми главно заради алгоритмичния принцип, прилаган в него, а именно – принципа на дихотомичното отхвърляне. Думата “дихотомия” е с гръцки произход и означава “разрязване” (томия) на две (дихо) части. Подходът “дихотомично отхвърляне” довежда до общ процес, състоящ се в многократно разделяне на нещо на две части и отхвърляне на едната, като всяко следващо

разделяне с отхвърляне се извършва над частта, останала от предишното разделяне. Интуитивно е ясно, че такъв процес води до много бързо намаляване на “нещото”, подложено на разделяне с отхвърляне. Това е и основната сила на този принцип – неговото уместно прилагане води до получаване на много ефективни (бързи) алгоритми. Описаният подход се прилага най-удачно за намиране на тази част от подложеното на обработка “нещо”, която е дефинирана като решение на дадена алгоритмична задача. Това го прави особено ценен за прилагане в един много широк клас алгоритмични задачи – задачите за *претърсване*.

Ако се направи справка в някоя математическа енциклопедия, ще се установи, че под “дихотомия” се разбира именно прилагането на принципа на дихотомичното отхвърляне при решаване на уравнението $f(x)=0$. Вероятно този принцип е бил за първи път активно прилаган именно в числените методи. Впечатляващият резултат от неговото прилагане е довел до “кръщаването” с името “дихотомия” на числен метод за решаване на следната:

Задача:

Дадена е:

Функцията $f(x)$, дефинирана и непрекъсната в интервал $[left, right]$. Функцията има *единствена нула* в този интервал.

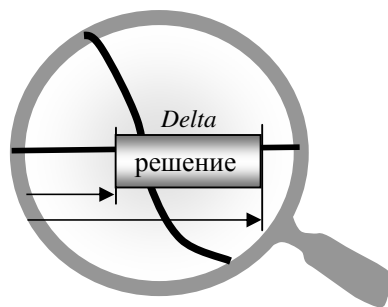
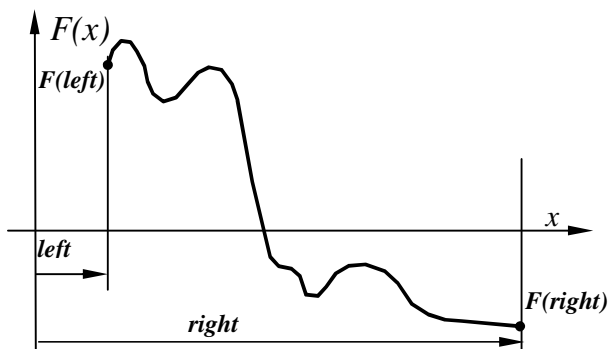
Търси се:

Коренът на $f(x)=0$.

(С някаква предварително зададена допустима грешка $Delta$)

Както се вижда, при така дефинираната постановка, задачата има някои ограничения по отношение на функцията и интервала, в който се търси корен. Искане се графиката на функцията да пресича оста X точно един път, някъде в границите на този интервал.

Да дефинираме изхода на задачата така: решението на задачата е намерено, ако са известни границите на интервал, в който функцията $F(x)$ има единствената си нула и дължината на интервала е по-малка или равен на допустима грешка $Delta$.

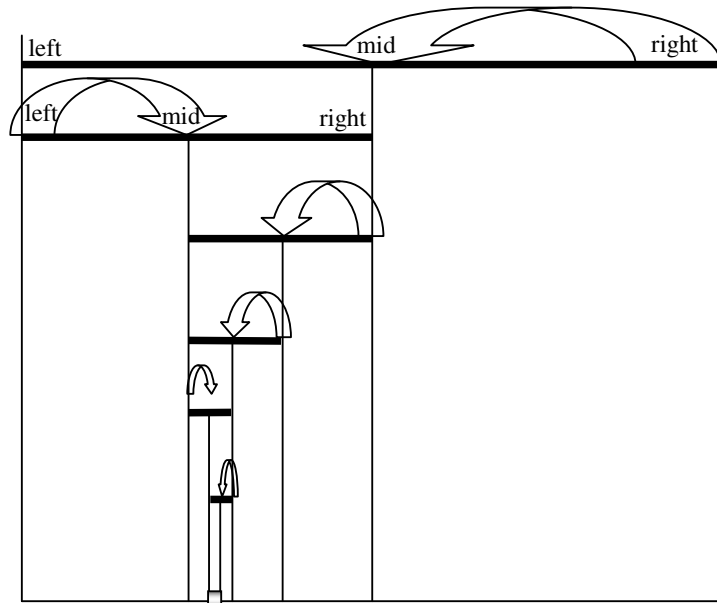


Според изложената принципна постановка на числово решаване на уравнение, тази задача се свежда до задача за изолиране на единствената нула на функцията в интервалче с големина не по-голяма от $Delta$, както е илюстрирано на предходната схема.

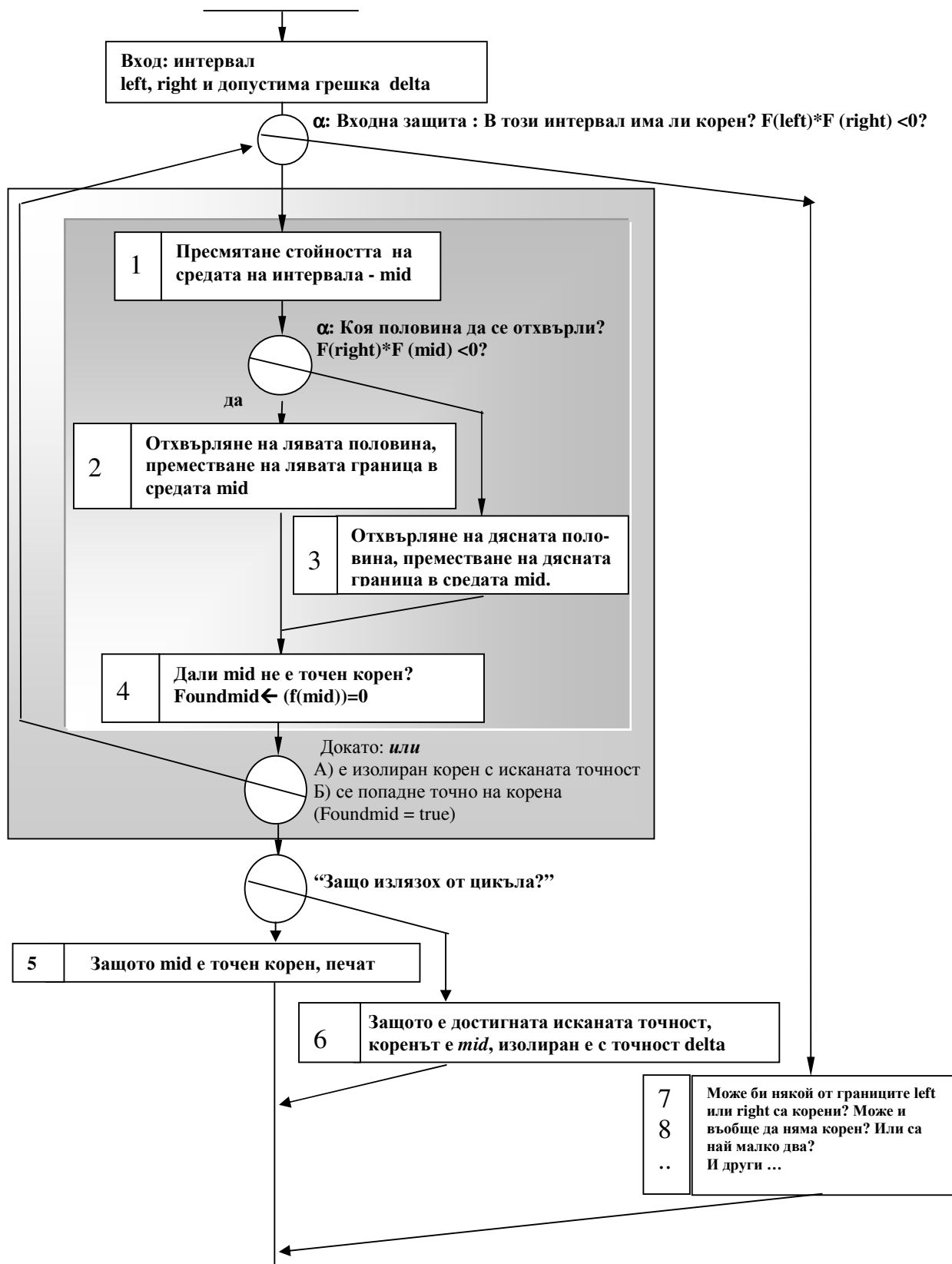
При така дефинирания изход – решение на задачата, интервалът $[left, right]$ би могъл да се окаже търсеното решение, ако неговата големина удовлетворява налаганите изисквания за точност. В подадения на входа да задачата интервал има единствена нула. Ако този интервал е по-голям от допустимата за изолирането на нулата грешка, интервалът се дели на две равни части. При това е ясно, че търсената нула е останала в едната от тях. Другата част се отхвърля и задачата се свежда до изходното си положение – зададен е интервал, в който функцията има единствена нула. Необходимата на алгоритъма информация е следната: за краищата на интервала, подлежащ на “обработка”, трябва стойностите на функцията да са с различен знак.

Така описаният алгоритъм очевидно може да бъде реализиран итеративно. Итерационният процес работи над големината на интервала, в който се намира нулата на функцията. Той получава на входа си интервал, проверява коя половина от интервала да отхвърли, пресмята границите на нов интервал, два пъти по-малък, и го подава обратно на входа си. Това се повтаря дотогава, докато интервалът не стане достатъчно малък.

Да подкрепим изложението с някои подробности: Половината, която не съдържа нулата се “отхвърля”, като нейният край, този, който е в ролята на граница на целия интервал, се премести в средата и стане новата граница на интервала. На фигурата вдясно е илюстриран процесът на последователно местене на границите на интервала $left$ или $right$ в средата му mid .



Както споменахме, процесът на преместване на границите завършва тогава, когато интервалът стане по-малък или равен на удвоената допустима грешка.



Горе е приведена една примерна схема на управление за този алгоритъм. Използвани са имената на променливи от общата схема на метода. Основният блок на управлението при този алгоритъм – итеративният цикъл, е показан в

реализация със следусловие. Тъй като условието за изход от цикъла проверява допълнително и за “точно попадение” в корена, след края на цикъла се прави проверка за причината на прекратяването му – алгоритмична особеност, на която се спряхме преди.

◆ *Примерна задача за упражнение*

Да се съставят двете програми за числово решаване на уравнение – по метода на сканиране на интервала и по метода дихотомията.