**СЕМИНАРНО УПРАЖНЕНИЕ № 1**

### PROgramming in LOGic  PROLOG

**І. Общи сведения.**

Пролог се стреми да извлече логически извод от данни, с които разполага. Обикновено, програмата на Пролог не е последователност от действия, както при алгоритмичните езици, а е съвкупност от факти, заедно с правилата за извеждането на заключения от тези факти. Пролог е известен като декларативен език.

Една важна характеристика на Пролог е, че в допълнение към логичното намиране на отговори на поставени въпроси, той може да се справи с търсене на алтернативи и да се намерят всички възможни решения, а не само едно.

Програмистът определя обектите и отношенията, след което определя правилата за това кога тези отношения са верни. Една програма на Пролог се състои от два вида фрази (известен също като клаузи): факти и правила.

**Факти**: Това, което е известно.

В Пролог, връзката между обектите се нарича **предикат**. В естествения език, една връзка е символизирана от едно изречение. В предикатната логика, която Пролог използва, връзката е обобщена в една простичка фраза - факт, - която се състои от отношението име, последвано от обект или обекти (затворени в скоби). Както и при изреченията фактът, завършва с точка (.).

**Правила**: Определят зависими отношения между факти и правила.

Правилата позволяват да се изведат, факти, от други факти. Друг начин да се каже това е, че по правило, заключенията се възприемат за истина, ако едно или повече други заключения или факти се установят, че са истина. В Пролог всички правила, имат две части: глава и тяло, разделени от специален маркер :-. Главата е факт, който би бил верен, ако определен брой условия са верни. Тялото е съвкупност от условия, които трябва да бъдат проверени за истинност, за да може да се твърди, че главата е истина.

**Цел :** Получаване на отговор от системата.

След като Пролог разполага със съвкупност от факти и правила, може да се задават въпроси, свързани с тези факти, това е известно като система за отговор на Пролог. Пролог винаги търси решение, като започва от началото на фактите и / или правилата, и продължава да търси, докато достигне до дъното. Намирането на удовлетворение на една цел изисква използването на един от всичките предикати, описни в програмата. Много често обаче е възможно да се търсят няколко свързани цели, всяка от която се нарича подцел. Обвързването на няколко подцели оформя сложна цел, в която могат да се използват логическите операции конюнкция и дизюнкция( логическо „и” и „или”). Синтактичното обозначаване на двете операции е съответно за конюнкция – запетая(,), а за дизюнкция – точка и запетая(;).

При описанието на правилата също е възможно използването на тези две логически операции, обозначени по същия начин.

**Променливи**: В Пролог, променливите позволяват да се пишат общи факти и правила и да се зададат обобщаващи въпроси. Първият знак от името на променливата трябва да бъде главна буква или долна черта, след което последователност от букви (главни или малки букви), цифри, знаци или долна черта. Променливите в Пролог получават своите стойности след обвързването им с фактите или правилата. След като получат стойност те я „съхраняват” докато машината за извод намери едно решение и веднага след това променливата става необвързана за да се намери друго решение. Един много важен момент е, че променливите в Пролог не се използват за съхраняване на стойности, в смисъла на алгоритмичните езици.

В Пролог е възможно използването на анонимни променливи. Когато се изисква точно определена информация от заявка, могат да се използват анонимни променливи за да се игнорират стойностите, които не са необходими. Анонимната променлива се представя като самостоятелен знак за подчертаване (\_) и може да се използва на мястото на всяка друга променлива. Разликата е, че анонимната променлива никога няма да има установена стойност.

**ІІ. Структура на програмата**

Една програма на Visual Prolog е съставена от няколко раздела.

Обикновено програмата включва четири основни раздела: раздел за клаузи, раздел за предикати, раздел за дефиниране на променливи и раздел цел.

Разделът с клаузите е сърцето на програмата на Пролог. В нея се поставят фактите и правилата, по които Пролог ще търси удовлетворяване на целта.

Разделът за предикати е мястото, където се декларират правилата и типовете на променливите на тези предикати. Ако в програмата се използват вградени предикати не е необходимо те да се декларират.

Разделът за дефиниране на променливи е мястото, където се декларират променливи, които ще се използват в програмата. Стандартните типове не се нуждаят от деклариране.

Разделът цел е мястото, където трябва да се постави целта, която ще се удовлетворява в програмата.

**Разделът с клаузите** включва *фактите* и *правилата*, в следствие на които ще се получават изводите в програмата.

**Фактите** описват свойствата или връзките между разглежданите обекти. Синтактично фактите се описват с име, последвано от отваряща скоба обект или обекти, отделени със запетая и затваряща скоба. Задължително в края се поставя точка.

При описание на връзка(отношение) може да се разгледа следния пример:

|  |  |
| --- | --- |
| На естествен език | В Пролог |
| Иван *харесва* Мария. | *харесва*(иван, мария). |
| Мария *харесва* Иван. | *харесва*(мария, иван). |
| Иван *харесва* кучета. | *харесва*(иван, кучета). |

*Забележка*: Стойностите, които се използват във фактите трябва да се записват с малки букви. Ако все пак е необходимо стойността да бъде с главна буква, то тя се загражда с двойни кавички. Пример: *харесва*(„Иван”, „Мария”).

При описание на свойства може да се разгледа следния пример:

|  |  |
| --- | --- |
| На естествен език | В Пролог |
| Небето е синьо. | *синьо*(небе). |
| Надежда е момиче. | *момиче*(надежда). |

**Правилата** извличат информация от фактите.

Ако се приеме, че съществуват следните правила:

*Мария харесва всичко, което Иван харесва.*

*Надежда харесва всичко, което е синьо.*

могат да се направят следните изводи, а именно:

*Мария харесва Мария.*

*Надежда харесва небето.*

За да се напишат на Пролог тези правила е необходимо следното:

|  |  |
| --- | --- |
| На естествен език | В Пролог |
| *Мария харесва всичко, което Иван харесва.* | *харесва*(мария, Нещо):-харесва(иван, Нещо). |
| Надежда харесва всичко, което е синьо. | *харесва*(надежда, Нещо):-синьо(Нещо). |

Символното означение :- се чете като „ако”(if). В конкретния случай *Нещо* е променлива и поради тази причина е с главна буква. Пролог ще търси всички възможни обвързвания на променливата спрямо фактите така, че да се намерят всички възможни решения.

**Разделът за предикати** служи за деклариране на всички използвани в разделът на клаузите правила(предикати), които не са част от Пролог. При описанието на предиката задължително ако има параметри, се посочва техния тип.

Декларирането на потребителски предикати започва с името, следвано от отваряща скоба. Последователно се описват всички параметри, като се подава от какъв тип са, разделени със запетая. Типовете, които се използват трябва да бъдат или стандартни, или описани в раздела за дефиниране на променливи. Предикатът завършва със затваряща скоба. За разлика от разделът с клаузите при декларирането на предикат не се изисква в края да се поставя точка.

Общ вид:

*име\_на\_предикат(тип\_на\_аргумент1, тип\_на\_аргумент2, ..., тип\_на\_аргументN)*

За име на предикат могат да се използват латински букви малки и главни, цифри и знак за подчертаване. Първият символ от името трябва да е малка латинска буква. Дължината на името не може да надвишава 250 символа.

Примери за имена на предикати:

|  |  |
| --- | --- |
| Допустими имена | Недопустими имена |
| fact | [fact] |
| is\_a | \*is\_a\* |
| has\_a | has/a |
| patternCheckList | pattern-Check-List |
| choose\_Menu\_Item | choose Menu Item |
| predicateName | predicate<Name> |
| first\_in\_10 | >first\_in\_10 |

Типовете на аргументите трябва да са из между вградените в Пролог или потребителски дефинирани в раздела за дефиниране на променливи.

Пример:

Ако трябва да се декларира предиката *my\_predicate(symbol, integer)* в раздела за предикати, то е необходимо следното:

PREDICATES

my\_predicate(symbol, integer)

В този случай не е необходимо да се декларират типовете *symbol* и *integer*, защото те са стандартни типове в Пролог.

Ако обаче трябва да се декларира предиката по този начин:

*my\_predicate(name, number)*

ще е необходимо дефинирането на *name* и *number* в раздела за дефиниране на променливите, а именно:

DOMAINS

name = symbol

number = integer

PREDICATES

my\_predicate(name, number)

**Разделът за дефиниране на променливите** помага за по-лесната интерпретация на променливите, които се използват в програмата, дори когато типовете са стандартни за езика и служи за дефиниране на нови типове, на базата на съществуващите.

При използването на типовете и свързаните с тях променливи трябва да се има в предвид, че за разлика от алгоритмичните езици за програмиране ако две променливи са обявени от един тип, то те навсякъде могат да се използват в контекста на типа и дефинираните операции над него, независимо от последователността, спазвайки логиката на изчисление. В Пролог две обявени от един тип променливи образуват различни домейни и при тяхното последователно използване не винаги се получава очакван резултат. За илюстрация е даден следния пример:

**DOMAINS**

product, sum = integer

**PREDICATES**

add(sum, sum, sum)

multiply(product, product, product)

**CLAUSES**

add(X, Y, Sum):-

Sum=X+Y.

multiply(X, Y, Product):-

Product=X\*Y.

Тази програма извършва две неща: намира сумата и произведението на две числа.

Ако се зададе следната цел:

add(32, 54, Sum).

Резултата, който ще се получи е:

Sum=86

1 Solution

Ако се зададе следната цел:

multiply(31, 13, Product).

Резултата, който ще се получи е:

Product=403

1 Solution

Ако е необходимо да се удвои произведението на 31 и 17, целта би трябвало да бъде преобразувана по следния начин:

multiply(31, 17, Sum), add(Sum, Sum, Answer).

Очаквания резултат е:

Sum=527, Answer=1054

1 Solution

Вместо това Пролог извежда съобщение за грешка

E;Test\_Goal, pos: 221, 505 Type error: Illegal variable type for this position

Въпреки че двете променливи са от тип цяло число, те са различни области, и се приемат като такива.

Така че, ако една променлива се използва в повече от един предикат в рамките на клауза, тя трябва да бъде обявена една и съща във всеки предикат.

За да се получи очаквания резултат трябва да се промени декларацията на един от предикатите, така че да използват едни и същи променливи

PREDICATES

add(sum, sum, sum)

multiply(sum, sum, sum)

**Разделът цел** има същата структура като на правило – той е просто списък от подцели. Има две разлики между правило и цел:

1.Не се използва :-

2.Пролог автоматично изпълнява целта след стартирането на програмата.

След стартирането на програмата се търси удовлетворение на целта. Ако такова се намери програмата спира успешно с намерената цел. Ако целта е съставена от подцели, то се търси удовлетворяването на всяка една от тях. В случай, че някоя от тях пропадне, като цяло целта също не се удовлетворява.

**Подробен поглед върху декларирането на променливи и вградените типове**

|  |  |
| --- | --- |
| ***Домейн*** | ***Описание*** |

|  |  |  |
| --- | --- | --- |
| ***short***  |  |  |
|  | 16, 32bit архитектура | 16 bits  | -32768 .. 32767 |  |
| ***ushort***  |  |  |
|  | 16, 32bit архитектура | 16 bits  | 0 .. 65535 |  |
| ***long***  |  |  |
|  | 16, 32bit архитектура | 32 bits  | -2147483648 .. 2147483647 |  |
| ***ulong***  |  |  |
|  | 16, 32bit архитектура | 32 bits  | 0 .. 4294967295 |  |
| ***integer*** |  |  |
|  | 16bit архитектура | 16 bits  | -32768 .. 32767 |  |
|  | 32bit архитектура | 32 bits  | -2147483648 .. 2147483647 |  |
| ***unsigned*** |  |  |
|  | 16bit архитектура | 16 bits | 0 .. 65535 |  |
|  | 32bit архитектура | 32 bits  | 0 .. 4294967295 |  |
| ***byte*** |  |  |
|  | 16, 32bit архитектура | 8 bits  | 0 .. 255 |  |
| ***word*** |  |  |
|  | 16, 32bit архитектура | 16 bits | 0 .. 65535 |  |
| ***dword*** |  |  |
|  | 16, 32bit архитектура | 32 bits  | 0 .. 4294967295 |  |
| ***char***  | Символ вътрешно представен като байт без знак. Синтактично се изписва като символ заграден с единични кавички 'a'. |
| ***real*** | Число с плаваща запетая, заемащо 8 байта, еквивалентен тип на С - ***double***. Синтактично реалното число се записва с знак (+ или -), последвано от цифри, после от десетична точка (.), последвано от още цифри. Възможно е и използването на експоненциален вид на числото. Примери за реални числа:42705 -99.99 86.729111.929437521e238 79.83e+21Позволените стойности са от 10-307 до 10308.  |
| ***string***  | Последователност от символи, означаващи се като указател към масив от символи завършващи с 0, както е в С. Два формата са валидни за стринговете:1. последователност от букви, цифри, символ за подчертаване, започващи с малка буква; или2. последователност от символи, заградена с двойни кавички.Примери за стрингове:telephone\_number "railway ticket" "Dorid Inc"Стринговете, които се въвеждат в програмата не могат да надвишават 255 символа. Стринговете, които Пролог чете от файл, могат да бъдат с дължина до 64КB при 16 битова архитектура и теоретично до 4GB в 32 битова архитектура. |
| ***symbol*** | Последователност от символи, означаващи се като указател към запис в хеш таблица на символите. Синтаксисът е същия като при стринговете. |

Символите и стринговете до голяма степен са взаимозаменяеми в програмите на Пролог, но те се съхраняват по различен начин. Символите се съхраняват в таблица за търсене и техните адреси, а не самите символи, представят тези обекти. Това означава, че самите символи се сравняват много по-бързо и символи използвани многократно в програмата се съхраняват много по-компактно. Стринговете не се съхраняват в хеш таблици; Пролог ги проверява символ по символ, докато намери съвпадение. Трябва да се определи кои тип ще предостави по-добра производителност за конкретната програма.

**Множествено деклариране**

Възможно е в една програма на Пролог да има декларирани повече от един предикат с едно и също име. Различното между тях се явява различния брой аргументи, които те имат. Когато се описват в програмата предикатите трябва да бъдат групирани (описани последователно) и в частта на предикатите и клаузите. Ако се премахне това ограничение трябва да се разбират като напълно различими предикати.

DOMAINS

person = symbol

PREDICATES

father(person) % Този *person* е баща

father(person, person) % Един *person* е баща на друг *person*

**Операторът if в Пролог, срещу if в другите езици**

Когато се използва символа :- , разделящ главата от тялото в правило се чете if. Има разлика обаче в интерпретацията на if между стандартен език за програмиране, например Pascal, и Пролог.

В Паскал, когато е необходимо да се създаде условие се използва оператор IF, като синтактично се започва с него. Семантично може да се опише това по следния начин с малко перифразиране:

"if ГЛАВАТА е истина, then ТЯЛОТО е истина (или then изпълни ТЯЛОТО)"

Този начин на запис е известен като if/then структура. Пролог използва друга форма на логика в своите правила. Главата в правило на Пролог се оценява като истина ако тялото на правилото е оценено като истина. С други думи:

"ГЛАВАТА е истина АКО ТЯЛОТО е истина "

Спазвайки този маниер, правилото в Пролог е проявление на then/if структура.

**Автоматично преобразуване на типовете**

Когато Пролог сравнява две променливи не е задължително те да бъдат от един и същ тип. Това преобразуване е възможно поради това, че Пролог предлага автоматично преобразуване между следните типове:

* + между string и symbol
	+ между числовите типове и real

**Други раздели в програмата**

**Раздел за факти**

Програмата на Пролог е сбор от факти и правила. Понякога в процес на изпълнението на програмата е необходимо да се обновят(променят, премахнат или добавят) някои от фактите в програмата. В такъв случай фактите могат да се разглеждат като динамична база от данни, която да може да се променя в хода на изпълнение на програмата. Пролог разполага със специален раздел за деклариране на фактите в програмата, които да бъдат част от динамична база от данни.

Думата **FACTS** обозначава началото на този раздел. Пролог разполага с вградени предикати за използване на динамичната база от данни, разположена в този раздел. Думата **FACTS** може да се замести с **DATABASE**.

**Раздел за константи**

В програмата на Пролог може да се декларират и използват символни константи. Декларацията на константите започва с запазената дума **CONSTANTS** и се спазва следния синтаксис:

<Id> = <дефиниция> , където

Id е името на константата, а дефиницията е стойността, която се дава на константата. Всяка дефиниция се прекратява от символа за нов ред, което означава, че на един ред може да се декларира по една константа.

Нека е даден следния пример:

**CONSTANTS**

zero = 0

one = 1

two = 2

hundred = (10\*(10-1)+10)

pi = 3.141592653

ega = 3

slash\_fill = 4

red = 4

Преди да се компилира програмата, Пролог ще замести всяка константа със нейната стойност. Например, ако в програмата има следния код:

...,

A = hundred\*34, delay(A),

setfillstyle(slash\_fill, red),

Circumf = pi\*Diam,

...

ще бъде възприеман от компилатора по следния начин:

...,

A = (10\*(10-1)+10)\*34, delay(A),

setfillstyle(4, 4),

Circumf = 3.141592653\*Diam,

...

**ЛАБОРАТОРНО УПРАЖНЕНИЕ №1**

**ТЕОРЕТИЧНИ БЕЛЕЖКИ**

**І. Структура на програмата**

**DOMAINS**

Разделът **DOMAINS** служи за деклариране на потребителски дефинираните домейни, който се използват като аргументи в предикатите. Под домейн в Пролог се разбира тип на променлива в другите езици. Основните вградени типове в Пролог са ***char***, ***byte***, ***short***, ***ushort***, ***word***, ***integer***, ***unsigned***, ***long***, ***ulong***, ***dword***, ***real***, ***string***, и ***symbol***. Стандартния начин за деклариране е следния:

DOMAINS

argument\_type1, ..., argument\_typeN = <standard domain>

**PREDICATES**

Използва се за деклариране на твърденията(предикатите) в програмата. Името на предиката трябва да започва с малка латинска буква, последвано от буква, цифра или символ за подчертаване с обща дължина от не повече 250 символа. В името на предиката са забранени използването на интервали, знак минус, звездички, наклонени черти.

Стандартния начин за деклариране е следния:

PREDICATES

predicateName(argument\_type1, argument\_type2, ..., argument\_typeN) , където

*argument\_type1*, ..., *argument\_typeN* са или стандартни типове или декларирани в раздела Domains.

**CLAUSES**

Описва действието на предикатите или фактите – основата, върху която се вземат решения. Описанието на предикатите задължително изисква те предварително да бъдат обявени в секцията **predicates.** Възможно е да се запишат няколко действия за един и същ предикат в секцията **clauses.** Те трябва да бъдат записани последователно. Всяко описано действие на предикат завършва с точка.

Стандартния начин за деклариране е следния:

HEAD :- <Subgoal1>, <Subgoal2>, ..., <SubgoalN>.

**GOAL**

Секция за определяне на целта. Тук задължително се използва един или няколко от описаните предикати. Възможно е в предикатите да се използват променливи(не е задължително имената им да съвпадат с описаните в секцията **predicates**) или константи, посочени в необходимия вид. Включените няколко предиката, се отделят със запетая, като връзката между тях е логическо „и” или точка и запетая, като връзката между тях е логическо „или”, а удовлетворяването е в реда на посочването им. Секцията за определяне на целта завършва с точка. Ако не е указано друго, се търсят всички възможни удовлетворявания на целта.

**ІІ. Използване на коментари**

% - всичко, което е след знака за процент до края на реда, се смята за коментар

/\*.......\*/ - служи за поставяне на коментар на няколко реда

**ПРАКТИЧЕСКИ ЗАДАЧИ**

**І. Фамилни отношения**

Фамилните връзки са добър начин за да се покажат възможностите на Prolog за представяне на факти и правила за търсене на решения.

В примера са показани връзки между синове, сестри и т.н. Те са зададени като факти в секцията clauses на Visual Prolog програмата:

*% секция за обявяване на променливите*

DOMAINS

Name=symbol

PREDICATES

nondeterm son(Name Son, Name Father)

nondeterm sister(Name Sister1, Name Sister2)

nondeterm brother(Name Brother1, Name Brother2)

nondeterm father(Name Father, Name Child)

nondeterm grandfather(Name Grandfather, Name Grandchild)

nondeterm married(Name Man, Name Woman)

CLAUSES

son("Ivan","Iordan").

son("Iordan","Marin").

sister("Maria","Silvia").

brother("Petyr","Iordan").

married("Iordan","Maria").

married("Petyr","Stela").

father(A,B):-son(B,A).

grandfather(A,B):-father(A,C),father(C,B).

GOAL

grandfather("Marin",Z).

%grandfather(Z,"Ivan").

*Допълнения към задачата:*

1. Да се добави предикат и клауза за намиране на „баба”.
2. Да се добави предикат и клауза за намиране на „чичо”.
3. Да се добави предикат и клауза за намиране на „леля”.

Решения:

*за точка 1:*

1. Трябва да се добави предикат **daughter** (дъщеря) в частта на предикатите;
2. Трябва да се добави предикат **mother** (майка) в частта на предикатите;
3. Трябва да се добави предикат **grandmother** (баба) в частта на предикатите;
4. Трябва да се опише действието на предиката **grandmother** в частта на клаузите.

В частта PREDICATES се добавя

nondeterm daughter(Name Daughter, Name Mother)

nondetermmother**(**Name Mother, Name Daughter**)**

nondetermgrandmother(Name Grandmother, Name Grandchild)

В частта CLAUSES се добавя

daughter(“Maria”,”Kalina”).

daughter(“Kalina”,”Galia”).

mother(X,Y):-daughter(Y,X).

grandmother(X,Y):-mother(X,Z),mother(Z,Y).

В частта GOAL се записва

grandmother(Z,”Maria”). % отговора е Galia

*за точка 2:*

1. Трябва да се добави предикат **uncle** (чичо) в частта на предикатите
2. Трябва да се опише действието на предиката **uncle** в частта на клаузите

В частта PREDICATES се добавя

nondeterm uncle(Name Uncle, Name BrotherSon)

В частта на CLAUSES се добавя

uncle(X,Y):-brother(X,Z),father(Z,Y).

В частта GOAL се записва

uncle(X,”Ivan”). %Кой е чичото на Ivan – отговора е Petyr

*за точка 3:*

1. Трябва да се добави предикат **aunt** (леля) в частта на предикатите
2. Трябва да се опише действието на предиката **aunt** в частта на клаузите

В частта PREDICATES се добавя

nondeterm aunt(Name Aunt, Name Child)

В частта CLAUSES се добавя

aunt(X,Y):-married(Z,X),brother(Z,A),father(A,Y).

В частта GOAL се записва

aunt (X,”Ivan”). %Коя е лелята на Ivan – отговора е Stela

**ІІ. Хардуерна симулация**

Пролог позволява да се моделират логически схеми, като се използват таблици за истинността и разгледаните закони на предикатната логика. Предикатите показават отношенията между сигналите на входа и изхода на схемата.

Използват се не само външни, но и вътрешни връзки. За да разбереме как работи да разгледаме пример за конструиране на exclusive OR (*изключващо ИЛИ)* отелементи AND (*И*), OR(*ИЛИ*), NOT(*НЕ*) и след това да проверим работата му с програма.

Схемата на чипа изглежда така:



**domains**

 d = integer

**predicates**

nondeterm not\_(D,D)

 and\_(D,D,D)

 or\_(D,D,D)

nondeterm xor\_(D,D,D)

**clauses**

not\_(1,0).

 not\_(0,1).

 and\_(0,0,0).

 and\_(0,1,0).

 and\_(1,0,0).

 and\_(1,1,1).

 or\_(0,0,0).

 or\_(0,1,1).

 or\_(1,0,1).

 or\_(1,1,1).

 xor\_(Input1,Input2,Output):-

 not\_(Input1,N1),

 not\_(Input2,N2),

 and\_(Input1,N2,N3),

 and\_(Input2,N1,N4),

 or\_(N3,N4,Output).

goal

 xor\_(Input1,Input2,Output),

 format(Msg," xor\_(%,%,%)",Input1,Input2,Output),

 write(Msg).**СЕМИНАРНО УПРАЖНЕНИЕ № 2**

**І. ТЪРСЕНЕ НА СЪВПАДЕНИЕ - УНИФИКАЦИЯ**

Нека е дадена следната програма:

DOMAINS

title,author = symbol

pages = unsigned

PREDICATES

book(title, pages)

nondeterm written\_by(author, title)

nondeterm long\_novel(title)

CLAUSES

written\_by(fleming, "DR NO").

written\_by(melville, "MOBY DICK").

book("MOBY DICK", 250).

book("DR NO", 310).

long\_novel(Title):-

written\_by(\_, Title),

book(Title, Length),

Length > 300.

***Проследяване на логиката при вземане на решение от Пролог:***

**Първа цел:**

written\_by(X, Y).

Когато Пролог се опитва да удовлетвори целта written\_by(X, Y)., то трябва да се тестват всички клаузи в програмата за съвпадение. При опита за съвпадение на променливите X и Y с аргументите на всяка written\_by клауза, Пролог ще започне търсенето от началото на програмата към нейния край. Когато се намери клауза, отговаряща на целта, Пролог ще обвърже свободните променливи, така че целта и клаузата ще станат идентични. Тази търсеща съвпадение и съединяваща операция се нарича унификация.

Тъй като X и Y са свободни променливи в целта, а свободните променливи могат да се обвържат със всеки друг аргумент( дори и друга свободна променлива), унифицирането с първата клауза в програмата би изглеждала така:

written\_by( X , Y ).

 | |

written\_by(fleming, "DR NO").

Пролог намира съвпадение и Х обвързва с fleming, а Y се обвързва с “DR NO”, и извежда следния резултат:

X=fleming, Y=DR NO

Тъй като се проверява за всички възможни решения на целта, Пролог унифицира свободните променливи и с втората клауза на written\_by:

written\_by(melville,"MOBY DICK").

и Пролог извежда и второто решение:

X=melville, Y=MOBY DICK

2 Solutions

**Втора цел:**

written\_by(X, "MOBY DICK").

Пролог ще опита да намери съвпадение с първата клауза written\_by:

written\_by( X ,"MOBY DICK").

 | |

written\_by(fleming, "DR NO").

Тъй като "MOBY DICK" и "DR NO" не си съвпадат, опитът за унификация пропада. Пролог ще продължи със следващите факти в програмата:

written\_by(melville, "MOBY DICK").

Тук ще се намери унификация и Х ще се обвърже с melville.

**Трета цел:**

long\_novel(X).

Когато Пролог се опитва да удовлетвори целта се изследва дали целта не съвпада с някой факт или с глава на правило. В този случай се намира съвпадение с:

long\_novel(Title)

Пролог разглежда клаузата long\_novel, опитвайки се да намери съвпадение с унифициране на аргументите. Тъй като променливата Х не е обвързана в целта, може да бъде обвързана със всяка друга. От друга страна Title също не е обвързана променлива в главата на клаузата long\_novel. Целта съвпада с глава на правило и унификацията е направена. Пролог ще започне да извършва опит за удовлетворяване на подцелите на това правило.

long\_novel(Title):-

written\_by(\_, Title),

book(Title, Length),

Length>300.

Удовлетворяването на тялото на правилото ще премине първо през първата подцел written\_by(\_,Title). Трябва да се обърне внимание, че автора на книгата не е от значение, тъй като на мястото на author е поставена анонимна променлива. Пролог започва да търси решение от началото на програмата надолу и успява да унифицира подцелта с първия факт за written\_by и извършва следното:

written\_by( \_ , Title),

 | |

written\_by(fleming,"DR NO").

Променливата Title вече се обвързва с “DR NO” и за следващата подцел book(Title,Length) това ще има значение.

Пролог продължава със следващото търсене, опитвайки да намери съвпадение на book. Тъй като Title е обвързана с “DR NO” реално се търси book("DR NO", Length). Отново търсенето започва от началото на програмата. Трябва да се обърне внимание, че първото възможно обвързване с клаузата book("MOBY DICK", 250) пропада и Пролог ще продължи със втората клауза на book да търси съвпадение. Заглавието на книгата тук съвпада с подцелта и Пролог ще обвърже променливата Length със стойност 310.

Третата клауза в тялото на long\_novel сега се превръща в текущата подцел:

Length > 300.

Пролог извършва сравнението и успява, тъй като 310 е по-голямо от 300. В този момент всички подцели на правилото са удовлетворени и като цяло целта long\_novel(X) успява. Тъй като променливата Х от целта се унифицира с променливата Title от правилото, стойността с която правилото успее ще се върне и унифицира с променливата Х. Title има стойност “DR NO”, когато правилото е удовлетворено и поради тази причина изхода на Пролог ще е:

X=DR NO

1 Solution

### ІІ. BACKTRACKING

Често, когато се търси решение на реална задача, трябва да се проследи пътя, по който се извлича логически извод. Ако този извод не е удовлетворяващ трябва да се търси алтернативен път. Например типичен пример за търсене на алтернативни решения е играта за търсене на път в лабиринт. Един сигурен начин за намиране на изход от лабиринта е да се тръгва наляво на всяко разклонение на лабиринта, докато не се стигне до задънен край. В този момент връщането до последното разклонение и завиване на дясно и отново обхождане на ляво ще бъде успешен алгоритъм за намиране на изход.

Пролог използва подобен метод за връщане назад и нов опит известен като backtracking за намиране на решение на даден проблем. Когато Пролог започва да търси решение много възможно е да се избира между два възможни случая. В този момент той поставя маркер за разклонението(известен като точка за връщане назад) и проследява първия случай. Ако тази подцел пропадне(еквивалентно на задънен край) Пролог ще се върне в точката за връщане назад и ще проследи алтернативния случай.

Даден е следния пример:

PREDICATES

nondeterm likes(symbol,symbol)

tastes(symbol,symbol)

nondeterm food(symbol)

CLAUSES

likes(bill,X):-

food(X),

tastes(X,good).

tastes(pizza,good).

tastes(brussels\_sprouts,bad).

food(brussels\_sprouts).

food(pizza).

Тази малка програма има две двойки факти и едно правило. Правилото описва връзката likes, която има смисъл, че Bill харесва вкусната храна.

За да се поясни как работи механизмът на възврат при намиране на решение се подава целта:

likes(bill, What).

Когато Пролог започва да удовлетворява целта, търсенето на решения започва от началото на програмата.

Когато се намира съвпадение с първата клауза в програмата, променливата What се унифицира с променливата Х. Съвпадението на целта с главата на правилото определя началото на удовлетворяването на правилото. Удовлетворяването на цялото правило се разбива на удовлетворяване на всички подцели в това правило. И първото удовлетворяване е на food(X).

Когато трябва да се извърши ново удовлетворение, търсенето на съвпадение отново започва от началото на програмата.

Заради това променливата Х се свързва с първия намерен факт brussels\_sprouts. Тъй като има повече от една възможност за отговор на food(X) Пролог създава точка на възврат до факта food(brussels\_sprouts). Тази точка ще укаже мястото от където Пролог ще започне ново търсене на решение за food(X).

Когато се намери успешно съвпадение, започва да се търси удовлетворение на следващата подцел.

С променливата Х се обвързва brussels\_sprouts и следващата цел е

tastes(brussels\_sprouts, good)

и Пролог започва да търси удовлетворяване на тази подцел отново от началото на програмата. Тъй като няма клауза, която да съвпада с поставената цел Пролог задейства механизма на възврат. По този начин търсенето на съвпадения се връща в точката на възврат.

Когато една променлива се обвърже в клауза, единственият начин да се „освободи”е чрез механизма за възврат.

Връщането в точката на възврат освобождава всички променливи обвързани след тази точка и започва ново търсене с нова заявка. Променливата Х вече е свободна и при търсене на следващ факт за food(X) се привързва със стойността pizza.

Пролог преминава към следващата подцел в правилото спрямо новото обвързване на променливата. Следващата стъпка е в ход tastes(pizza, good) и отново от началото на програмата започва търсене на съвпадение. Този път съвпадение се открива и целта се удовлетворява успешно.

Тъй като променливата What в целта се унифицира с променливата Х в правилото likes, а променливата Х получи стойност pizza Пролог връща следния резултат:

What=pizza

1 Solution

### ІІІ. Използване на предиката fail

Пролог използва механизма на възврат, когато търсенето на съвпадение пропадне. В някои ситуации е необходимо да се провокира механизма на възврат за да се намерят различни решения. В пролог е предвиден специален предикат **fail,** зарадикойто проверката винаги пропада и по този начин се стартира ново търсене на решение. Ефекта на предиката може да се сравни със сравнението 2 дали е равно на 3 или всички други невъзможни цели.

Със следващият пример ще се илюстрира действието на този специален предикат:

DOMAINS

name = symbol

PREDICATES

nondeterm father(name, name)

CLAUSES

father(leonard,katherine).

father(carl,jason).

father(carl,marilyn).

GOAL

father(X,Y).

След стартирането на програмата, поради заложения механизъм на възврат, ще се получи следното решение:

X=leonard, Y=katherine

X=carl, Y=jason

X=carl, Y=marilyn

3 Solutions

Много по-удачно ще бъде ако чрез изхода на програмата се разбере реално съществуващата връзка father, т.е. да се изведе, че „leonard е баща на katherine”. За целта в програмата ще има необходимост от нов предикат, а именно:

DOMAINS

name = symbol

PREDICATES

nondeterm father(name, name)

**nondeterm everybody**

CLAUSES

father(leonard,katherine).

father(carl,jason).

father(carl,marilyn).

**everybody:-**

**father(X,Y),**

**write(X," is ",Y,"'s father\n"). % \n - нов ред**

GOAL

**everybody.**

След стартирането на програмата вместо извеждането на всички възможни решения Пролог извежда само едно.

leonard is katherine's father

yes

Причината за това да не се изведат всички, а само едно решение е, че Пролог е намерил едно решение и нищо не е в състояние да го провокира за търсене на ново. За да може все пак да се изведат всички възможни решения е необходимо да се провокира изкуствено механизма на възврат. Решението е следното:

DOMAINS

name = symbol

PREDICATES

nondeterm father(name, name)

everybody

CLAUSES

father(leonard,katherine).

father(carl,jason).

father(carl,marilyn).

everybody:-

father(X,Y),

write(X," is ",Y,"'s father\n"),fail.

GOAL

everybody.

Резултатът вече е :

leonard is katherine's father

carl is jason's father

carl is marilyn's father

no

Въпреки, че Пролог намира всички решения накрая извежда **no**. За да завърши целия процес утвърдително е добре да се добави и още едно записване на предиката everybody без тяло. Той винаги успява и резултата е yes. Последната редакция на програмата е следната:

DOMAINS

name = symbol

PREDICATES

nondeterm father(name, name)

 everybody

CLAUSES

father(leonard,katherine).

father(carl,jason).

father(carl,marilyn).

everybody:-

father(X,Y),

write(X," is ",Y,"'s father\n"),fail.

everybody.

GOAL

everybody.

### ІV. Предотвратяване на механизма на възврат (Прекъсване - Cut)

Пролог притежава средство, чрез което може да се предотврати механизма на възврат, който синтактично се записва със знак удивителен(!). Ефекта от него е, че не е възможно да се търси повторно удовлетворяване на подцелите, записани преди това.

Мястото на прекъсване може да бъде използвано като подцел в тялото на правило. Когато обработката премине през прекъсване се получава удовлетворение и започва да се търси успех на следващата подцел. Когато се премине през прекъсване е невъзможно да се търсят други решения на подцелите поставени преди него.

Съществуват две основни причини за използването на прекъсване:

* + Когато предварително се знае, че някои от възможностите няма да доведат до значими резултати, загуба на време и на място за съхранение на възможните решения ще бъде да се разгледат всички. Ако в такъв момент се използва прекъсване програмите ще станат по-бързи и ще се нуждаят от по-малко памет. Това е известно като зелено прекъсване.
	+ Когато логиката на програмата изисква прекъсване за да не се разглеждат някои възможни решения. Това е известно като червено прекъсване.

**Предотвратяване на механизма на възврат за предишни подцели в правило:**

r1 :- a, b, !, c.

С това правило се указва на Пролог, че ако решенията за подцели *a* и *b* са удовлетворени успешно, въпреки че Пролог би могъл да намери множество решения за *c* с използване на възврат и търсене на алтернативи за *a* и *b* това няма да се извърши. Не е възможно и използването за механизма на възврат и за друга клауза, която определя предиката *r1*.

Нека е даден следния пример:

**PREDICATES**

nondeterm buy\_car(symbol,symbol)

nondeterm car(symbol,symbol,integer)

colors(symbol,symbol)

**CLAUSES**

buy\_car(Model,Color):-

car(Model,Color,Price),

colors(Color,hot),

Price < 25000.

car(maserati,green,25000).

car(corvette,black,24000).

car(porsche,red,23000).

car(corvette,red,22000).

colors(red,hot).

colors(black,mean).

colors(blue,preppy).

**GOAL**

buy\_car(X, Y).

С този пример отговаря на въпроса какъв модел автомобил да бъде купен, ако желаният цвят е „hot” , при заложено ценово ограничение.

След стартиране на програмата и разглеждане на всички възможни решения с помощта на механизма на възврат се получава и следното решение:

X=porsche, Y=red

X=corvette, Y=red

2 Solutions

Ако добавим прекъсване в търсенето на решения, а именно:

**PREDICATES**

nondeterm buy\_car(symbol,symbol)

nondeterm car(symbol,symbol,integer)

colors(symbol,symbol)

CLAUSES

buy\_car(Model,Color):-

car(Model,Color,Price),

colors(Color,hot),!, 

Price < 25000.

car(maserati,green,25000).

car(corvette,black,24000).

car(porsche,red,23000).

car(corvette,red,22000).

colors(red,hot).

colors(black,mean).

colors(blue,preppy).

GOAL

buy\_car(X, Y).

Резултата би бил следния:

X=porsche, Y=red

1 Solution

В този случай първият автомобил, който удовлетворява и трите подцели на правилото е крайното решение, въпреки че съществува и друго. Все пак остава въпроса какво би трябвало да се промени за да се получи резултата:

X=corvette, Y=red

1 Solution

**Предотвратяване на механизма на възврат за други клаузи на правило:**

Прекъсване може да се използва и да се укаже на Пролог, че е подбрана коректната клауза от множество зададени клаузи. Например:

r(1):- ! , a , b , c.

r(2):- ! , d.

r(3):- ! , c.

r(\_):- write("Това е универсална клауза.").

Използването на прекъсване тук води до това, че предиката *r* става еднозначно определен. В конкретния случай Пролог използва предиката *r* с единствен целочислен аргумент. Ако се предположи, че това е 1, т.е. *r(1)* Пролог започва да търси съвпадение и го намира още с първата клауза. Въпреки, че съществуват повече от една възможности Пролог поставя точка на възврат до тази клауза. По този начин няма да се търси друго удовлетворяване на клаузата *r*. Ако се разгледа и от друг ъгъл този пример може да се каже, че по този начин се реализира множествен избор като в другите езици за програмиране(switch в C case в Pascal).

### V. Предикатът *not*

Предикатът not успява, когато зададената подцел не може да се укаже, че е истина.

Даден е следния пример, в който се пита кой е добър студент(с успех над 3.50) и не е на стаж:

**DOMAINS**

name = symbol

gpa = real

**PREDICATES**

nondeterm honor\_student(name)

nondeterm student(name, gpa)

probation(name)

**CLAUSES**

honor\_student(Name):-

student(Name, GPA),

GPA>=3.5,

not(probation(Name)).

student("Betty Blue", 3.5).

student("David Smith", 2.0).

student("John Johnson", 3.7).

probation("Betty Blue").

probation("David Smith").

**GOAL**

honor\_student(X).

**ЛАБОРАТОРНО УПРАЖНЕНИЕ №2**

**ТЕОРЕТИЧНИ БЕЛЕЖКИ**

**Унификацията** е процес на търсене на съвпадение между два предиката и обвързването на техните променливи. Този механизъм е необходим на Пролог да идентифицира коя клауза е обработена и да обвърже със стойност променливите. Съществуват няколко важни момента при осъществяването на унификация:

Когато Пролог се опита да удовлетвори целта търсенето на съвпадение започва от началото на програмата за.

Когато се прави ново обръщение за удовлетворяване, търсенето на съвпадение отново започва от началото на програмата.

Когато при търсене на съвпадение се стигне до успех започва да се търси решение за следващата подцел.

Когато една променлива бъде обвързана в клауза единствения начин за нейното освобождаване е чрез механизма на възврат.

**Механизма за връщане назад (backtracking)** с цел търсене на нови решения е начинът, по които Пролог търси решение в програмата. Този процес дава на Пролог възможност да търси решение през всички известни факти и правила за решение. Съществуват няколко основни принципа на този механизъм:

Подцелите се удовлетворяват в реда на посочването си.

Клаузите на предикатите се проверяват отново в реда на срещането си в програмата.

Когато подцелта съвпадне с глава на правило, тогава започва удовлетворяването на тялото. Тялото на правилото определя нова последователност от подцели за удовлетворяване.

Целта се удовлетворява като цяло, когато се намерят всички факти за всички проверени подцели в дървото на целите.

Обръщение към предикат към който са свързани няколко решения е неопределено (**non-deterministic**), докато обръщение което се свързва само с едно решение е определено (**deterministic**)*.*

Пролог предлага три средства за контролиране на последователността за намиране на логическия извод при търсене на решение:

Предикатът **fail** винаги не успява. По този начин стартира незабавно търсене на ново решение с механизма на възврат.

Предикатът **not** успява, когато свързаната с него подцел е неистина.

Прекъсването посредством **cut** предотвратява търсенето на нови решения.

**ПРАКТИЧЕСКИ ЗАДАЧИ**

**Задача 1.**

1)Да се въведе следната програма:

**DOMAINS**

 title,author = symbol

pages = unsigned

**PREDICATES**

 book(title, pages)

 nondeterm written\_by(author, title)

nondeterm long\_novel(title)

**CLAUSES**

written\_by(fleming, "DR NO").

written\_by(melville, "MOBY DICK").

 book("MOBY DICK", 250).

book("DR NO", 310).

 long\_novel(Title):-

 written\_by(\_, Title),

book(Title, Length),

Length > 300.

2)Да се изпробва със следните цели:

* + written\_by(X, Y).
	+ written\_by(X, "MOBY DICK").
	+ long\_novel(X).

**Задача 2.**

1) Да се въведе следната програма:

**PREDICATES**

 nondeterm likes(symbol,symbol)

 tastes(symbol,symbol)

nondeterm food(symbol)

**CLAUSES**

 likes(bill,X):-

food(X),

tastes(X,good).

 tastes(pizza,good).

tastes(brussels\_sprouts,bad).

 food(brussels\_sprouts).

food(pizza).

2)Да се изпробва със следната цел и да се проследи последователността при вземане на решение:

likes(bill, What).

**Задача 3.**

Да се напише коректно целта, чрез която в турнир но тенис на маса да се определят двойките от 9 годишни ученици, между всеки от които ще има по две срещи.

**DOMAINS**

 child = symbol

age = integer

**PREDICATES**

nondeterm player(child, age)

**CLAUSES**

 player(peter,9).

 player(paul,10).

 player(chris,9).

player(susan,9).

**GOAL**

 **player(Person1, 9),**

**player(Person2, 9),**

**Person1 <> Person2.**

**Задача 4.**

1)Да се въведе следната програма:

**DOMAINS**

name = symbol

**PREDICATES**

nondeterm father(name, name)

**CLAUSES**

father(leonard,katherine).

father(carl,jason).

father(carl,marilyn).

**GOAL**

father(X,Y).

2)В програмата да се направят необходимите корекции, за да може да се изведат всички възможни решения и изхода на програмата да е следния:

leonard is katherine's father

carl is jason's father

carl is marilyn's father

yes

**Задача 5.**

1)Да се въведе следната програма:

**PREDICATES**

nondeterm buy\_car(symbol,symbol)

nondeterm car(symbol,symbol,integer)

colors(symbol,symbol)

**CLAUSES**

buy\_car(Model,Color):-

car(Model,Color,Price),

colors(Color,sexy),

Price < 25000.

car(maserati,green,25000).

car(corvette,black,24000).

car(porsche,red,23000).

car(corvette,red,22000).

colors(red,sexy).

colors(black,mean).

colors(green,preppy).

**GOAL**

buy\_car(X, Y).

2)Да се ограничи решението само до едно

X=porsche, Y=red

1 Solution

3)Да се направят необходимите корекции за да се изведе следното решение:

X=corvette, Y=red

1 Solution

**Задача 6.**

1)Да се въведе програмата:

**DOMAINS**

name = symbol

gpa = real

**PREDICATES**

nondeterm honor\_student(name)

nondeterm student(name, gpa)

probation(name)

**CLAUSES**

 **honor\_student(Name):-**

**student(Name, GPA),**

**GPA>=3.5,**

**not(probation(Name)).**

student("Betty Blue", 3.5).

student("David Smith", 2.0).

student("John Johnson", 3.7).

probation("Betty Blue").

probation("David Smith").

**GOAL**

honor\_student(X).

2)Да се напише правилото honor\_student, чрез което ще се изведат имената на всички студенти с успех над 3.50 и които не са на стаж(probation).

3) Да се напише правилото best, чрез което ще се изведе името и успеха на най-добрия студент (с най-добър успех и не е на стаж).

PREDICATES

nondeterm best(gpa)

CLAUSES

best(M):-student(X,Y),Y>M,not(probation(X)),best(Y).

best(M):-student(X,M),write("Naj-visokijat uspeh e ",M, " na studenta ", X),nl.

GOAL

best(0).

**СЕМИНАРНО УПРАЖНЕНИЕ №3**

**І.СЛОЖНИ ТИПОВЕ ОБЕКТИ И ФУНКТОРИ**

Сложните типове обекти позволяват да се обединят отделни части от информация, като едно цяло. За пример може да се вземе датата 15.10.2010. По същество тя съдържа три отделни данни, но е много удобно да се разглежда като едно цяло. От друга страна това представяне може да се разглежда като дървообразно:

 DATE

 / | \

 15 10 2010

В Пролог е възможно да се декларират в раздела за дефиниране на променливи сложни типове обекти, в случая да се дефинира обекта **date**:

DOMAINS

date\_obj = date(unsigned, unsigned, unsigned)

В последствие в програмата може да се използва например

..., D = date(15,10,2010), ...

На пръв поглед може да се каже, че това е факт, но трябва да се разбира като обект с данни, с който може да се оперира по подобен начин както със символи и числа. Този обект започва с име т.нар. функтор, последван от три аргумента.

Трябва да се отбележи, че функтора в Пролог няма нищо общо с функциите в другите езици за програмиране. Функтора от друга страна не е пречка да бъдат извършвани изчисления. Това е просто име, чрез което се идентифицира сложен тип обект и обединява определен брой аргументи.

Аргументите на сложен тип данни, от своя стана също могат да бъдат от сложен тип. Например ако трябва да се опише информация за рожден ден на конкретен човек би трябвало да се създаде следната структура:

 BIRTHDAY

 / \

 / \

 person date

 / \ / | \

 "Иван" Петров" 4 12 1960

В Пролог описанието на тази структура би изглеждала така:

birthday(person("Иван","Петров"),date(4,12,1960))

В този пример сложният тип обект с фунтор birthday е съставен от два други сложни типа съответно с функтори person и date.

**Унификация на сложни обекти**

Сложните обекти могат да се унифицират както с отделна променлива, така и със сложен обект. Това означава, че е възможно да се използва сложния обект като едно цяло, а също да се използва унификация на отделните елементи. Например:

date(14,4,1960)

Ако се съпостави отделна променлива Х на този обект, то като цяло променливата Х ще се обвърже са date(14,4,1960).

От друга страна е възможно date(14,4,1960) да се съпостави на date(DD,MM,YY) и по този начин обвързването на променливите да е следното: DD с 14 MM с 4 и YY с 1960

**Използване на знак за равенство за унификация на сложни обекти**

Пролог прилага унификация по два начина. Първият е при удовлетворяване на цел с глава на клауза. Вторият начин преминава през знака за равенство, който на практика е инфиксен предикат, т.е. предикат, който се намира между аргументите си, а не преди тях.

Пролог извършва необходимите съпоставяния на обектите от двете страни на равенството. Това се използва за намиране на стойностите на аргументи в сложен тип обект.

**ІІ.ПОВТОРЕНИЯ И РЕКУРСИЯ**

Голяма част от ефективността на компютрите е вследствие на това, че те се справят с идентични операции, които се повтарят многократно. В Пролог повторението може да се изрази както в повторение на процедурите, така и в структурите си от данни. Идеята за повтарящи се структури от данни звучи особено, но Пролог позволява създаването на структури от данни, за които целият им размер не известен в момента на създаването им.

**Повтарящи се процеси**

Програмистите, които използват процедурни езици като С и Pascal много често задават въпроса защо в Пролог няма използване на оператори като For, While или Repeat. В езика няма директен начин за представяне на итерации. Пролог позволява два вида повторения: механизма на възврат за търсене на всички възможни решения в една заявка и рекурсия, в която процедурата извиква себе си.

**Вграждане на механизма на възврат в цикъл**

Механизмът на възврат е добър начин за намиране на всички решения в цел. Дори когато самата цел не съдържа множество решения може да се търси ново удовлетворяване, използвайки механизма на възврат, което на практика означава използване на повторения. Достатъчно е да се дефинира предикат с две клаузи:

repeat.

repeat :- repeat.

По този начин може да се осигури безкрайно много различни решения, позовавайки се на безкраен цикъл от повторения.

Със следващия пример се демонстрира тази възможност. Целта на програмата е до натискането на клавиш Enter, да осигури прочитането на символ от клавиатурата, в последствие неговото изобразяване.

**PREDICATES**

 repeat

typewriter

**CLAUSES**

 repeat.

repeat:-repeat.

 typewriter:-

repeat,

readchar(C), /\* Прочитане на символ и обвързването му с С \*/

write(C), /\* Извеждане на символ, обвързан с променливата С \*/

C = '\r',!. /\* Проверка за натиснат клавиш Enter \*/

GOAL

typewriter.

**Рекурсивни процедури**

Другият начин за осигуряване на повторения е използването на рекурсия. Рекурсивна процедура е тази, която се обръща към себе си. Рекурсивните процедури не изискват съхраняването на междинна информация по време на изпълнението на процедурата, защото броячи, суматори и междинни резултати се предават като аргументи при следващата итерация.

Задачата за намиране на N! използва рекурсивен алгоритъм за своето решение:

* Ако N е равно на 0 и 1, то факториела е 1 (дъно на рекурсията)
* Иначе се търси факториел от N-1, умножен по N

В пролог този алгоритъм би изглеждал по този начин:

factorial(0,1) :- !.

factorial(1,1) :- !.

factorial(X, FactX) :-

Y = X-1,

factorial(Y, FactY),

FactX = X\*FactY.

Цялата програма е следната:

PREDICATES

factorial(unsigned,real)

CLAUSES

factorial(0, ) :- !.

factorial(1,1):-!.

 factorial(X,FactX):-

Y=X-1,

factorial(Y,FactY),

FactX = X\*FactY.

GOAL

factorial(5,Fakt).

**ІІІ. Рекурсивни структури от данни**

**Дървета**

Най-важните рекурсивни типове данни са списъците, чрез които се конструират по-сложните типове данни използвани включително и в ИИ.

Дърветата са конструкции, които позволяват да се организира ефективно търсене в пространството на състоянията. По долу е показан пример за дефиниране на тази структура:

В Pascal може да се дефинира дърво чрез указател така:

treeptr = ^tree;

tree = record

name: string[80];

left, right: treeptr

 **end**;

Искат се усилия и знания за правилното използване на такива структури, което значително затруднява програмистите.

*В C как би трябвало да се дефинира същото?*

Пролог позволява да се определи действително рекурсивен тип, в който указателите се създават и обработват автоматично. Например дървото може да се определи по следния начин:

DOMAINS

treetype = tree(string, treetype, treetype)

Тази декларация показва, че дървото се описва чрез функтора ***tree***, аргументите на който са низ и две други дървета. Това обаче не е съвсем добра декларация, защото не се показва кога свършва самото дърво. В Пролог това става като се дефинира празно дърво empty.

DOMAINS

treetype = tree(string, treetype, treetype) ; empty

Названията ***tree*** и ***empty*** не са преопределени в езика и могат да бъдат всякякви други ***ххх*** или ***ууу*** например.



***фиг. 1*** *Част от фамилно (родословно) дърво*

Дървото от фиг. 1 може да се опише в Prolog програма така:

tree("Cathy",

tree("Michael"

tree("Charles", empty, empty)

tree("Hazel", empty, empty))

tree("Melody"

tree("Jim", empty, empty)

tree("Eleanor", empty, empty)))

Не е задължителен този запис, а може и така:

tree("Cathy"

tree("Michael", tree("Charles", empty, empty), tree("Hazel", empty, empty))

tree("Melody", tree("Jim", empty, empty), tree("Eleanor", empty, empty)))

**Обхождане на дървета**

Най-често възлите на дървото се изследват по някакъв начин или се търсят някакви или всичките им стойности. Тези процедури се наричат обхождане на дървото. Алгоритъм е следния:

1. Ако дървото е празно, то не прави нищо.

2. Иначе, обработи текущия възел, премини към лявото поддърво, след това премини към дясното поддърво.

Както и самото дърво алгоритъмът е рекурсивен: той обработва лявото и дясното поддърво точно както началното дърво. В Пролог това се описва с две клаузи, една за празното и една за непразното дърво:

traverse(empty). % не прави нищо

traverse(tree(X, Y, Z)) :-

do something with X,

traverse(Y),

traverse(Z).



***Фиг.2.****: Търсене „в дълбочина” в дървото от фиг.1.*

Този алгоритъм за обхождане познат като търсене „в началото в дълбочина” се спуска по клоните докато е възможно след което се връща горе, за да обходи следващия клон (Фиг.2.).

/\* Обхождане „в дълбочина” и отпечатване на срещнатите стойности \*/

DOMAINS

treetype = tree(string, treetype, treetype) ; empty()

PREDICATES

traverse(treetype)

CLAUSES

traverse(empty).

traverse(tree(Name,Left,Right)):-

write(Name,'\n'),

traverse(Left),

traverse(Right).

GOAL

traverse(tree("Cathy",

tree("Michael",

tree("Charles", empty, empty),

tree("Hazel", empty, empty)),

tree("Melody",

tree("Jim", empty, empty),

tree("Eleanor", empty, empty)))).

 Резултатът е:

Cathy

Michael

Charles

Hazel

Melody

Jim

Eleanor

**Създаване на дърво.**

Създаването на дърво с един възел чрез присвояване е тривиална задача:

**create\_tree(N, tree(N, empty, empty)).**

В този случай се казва: "Ако *N* са данни, то tree(N, empty, empty) е дърво от един възел, съдържащ тези данни."

Създаването на структурата на дървото също е лесно. На процедурата са необходими три дървета като аргументи. Тя вмъква първото дърво като ляво поддърво на второто, а резултата присвоява на третото дърво:

**insert\_left(X, tree(A, \_, B), tree(A, X, B)).**

Необходимо е да се забележи, че това правило няма тяло, т.е. няма точни стъпки за неговото изпълнение. Всичко, което трябва компютърът да извърши е да съедини аргументите един с друг по правилен начин и работата е завършена.

Да се предположи, че трябва да се вмъкне tree("Michael", empty, empty) като ляво поддърво на tree("Cathy", empty, empty). За това е необходимо да изпълните целта:

insert\_left(tree("Michael", empty, empty),

tree("Cathy", empty, empty),

T)

и тогава *T* незабавно получава стойност:

tree ("Cathy", tree("Michael", empty, empty), empty).

Това дава метода за построяване на дървото стъпка по стъпка и е показан в програмата по-долу. В действителност елементите, които трябва да се добавят в дървото могат да бъдат от външен вход.

/\* Елементарна процедура за построяване на дърво

 create\_tree(A, B) поставя A в полето за данни на едноелементното дърво B

 insert\_left(A, B, C) вмъква A като ляво поддърво за B и присвоява резултата на C

 insert\_right(A, B, C) вмъква A като дясно поддърво за B и присвоява резултата на C

\*/

**DOMAINS**

 treetype = tree(string,treetype,treetype) ; empty()

**PREDICATES**

 create\_tree(string,treetype)

 insert\_left(treetype,treetype,treetype)

 insert\_right(treetype,treetype,treetype)

 run

**CLAUSES**

 create\_tree(A,tree(A,empty,empty)).

 insert\_left(X,tree(A,\_,B),tree(A,X,B)).

 insert\_right(X,tree(A,B,\_),tree(A,B,X)).

 run:-

 /\* Първо се създават дървета с един възел \*/

 create\_tree("Charles",Ch),

 create\_tree("Hazel",H),

 create\_tree("Michael",Mi),

 create\_tree("Jim",J),

 create\_tree("Eleanor",E),

 create\_tree("Melody",Me),

 create\_tree("Cathy",Ca),

 /\* ...след това се съединяват... \*/

 insert\_left(Ch, Mi, Mi2),

 insert\_right(H, Mi2, Mi3),

 insert\_left(J, Me, Me2),

 insert\_right(E, Me2, Me3),

 insert\_left(Mi3, Ca, Ca2),

 insert\_right(Me3, Ca2, Ca3),

 /\* ...и накрая се разпечатва резултата. \*/

 write(Ca3,'\n').

**GOAL**

 run.

Необходимо е да се отбележи, че в Пролог няма възможност да се изменя стойността на променливата след като присвояването е станало. Затова в програмата се използват много имена на променливи; всеки път когато е необходимо да се получи нова стойност се изисква нова променлива. Големия брой имена на променливи тук е нещо необичайно, т.к. в общия случай повтарящите се процедури получават нови променливи викайки се рекурсивно, и всяко извикване има определена съвкупност от променливи.

**Двоични дървета за търсене.**

Дърветата могат да се използват и за други полезни цели освен за представяне на отношения между елементите им. В тях може да се поместват данни, така че те могат много бързо да се търсят и намират. Дърво, което е построено за такава цел се нарича *дърво за търсене.* От гледната точка на потребителя, структурата дърво не носи информация, дървото – това е *по- бърза алтернатива на списъка или масива*. За да се намери определена стойност в разгледаните дървета е необходимо да се разгледа всеки възел от цялото дърво.

***Времето което се губи за търсене в обикновено дърво с N възела е пропорционално на N.***

*Двоичното дърво за търсене* се строи по такъв начин, че гледайки от който и да е възел, може да се предскаже, в кой от неговите възли се намира зададената стойност. Това се прави като се подреждат стойностите в ред, като азбучния или числовия. Стойностите в лявото поддърво предшестват стойността в текушия възел, а в дясното – следват стойността му.

Забележете, че същите имена, подредени в различен ред, дават друго дърво, независимо че в дървото има десет имена, всяко едно от тях може да се намери максимум за пет стъпки.



**Фиг****. 3.** Двоични дърво за търсене 11

Всеки път по време на търсенето, когато се погледне възел от дървото, може да се елиминират от разглеждането на половината от останалите възли и търсенето да става много бързо. Ако дървото се увеличи двойно, то за търсене ще е необходима само една допълнителна стъпка.

***Времето което се губи за търсене в бинарното дърво с N възела е средно пропорционално на log2N (в действителност е пропорционално на log N, при всякаква основа).***

За да се построи бинарно дърво, трябва да се започне с празно дърво и да се добавят една след друга стойностите. Процедурата за добавяне е също такава както при търсенето: необходимо е да се намери мястото, където стойността трябва да се запише и се прави това. Алгоритъма е следния:

1. Ако текущия възел е празно дърво, то се вмъква в него стойността.
2. Иначе се сравнява стойността, която трябва да се вмъкне със стойността в текущия възел. Вмъкването се извършва в лявото или дясното поддърво, в зависимост от резултата от сравнението.

В Пролог за този алгоритъма са необходими три правила- по едно за всеки от случаите. Първото правило е:

insert(NewItem, empty, tree(NewItem, empty, empty) :- !.

Това на естествен език се превежда така ”Резултата от вмъкването *NewItem* (новата стойност) в *empty* (празно дърво) ще бъде дървото tree(NewItem, empty, empty). Знакът ! (отсичане) означава че ако това правило е успешно, то другите правила не се изпълняват. ”

Второто и третото правило извършват вмъкване в непразно дърво:

insert(NewItem, tree(Element, Left, Right), tree(Element, NewLeft, Right) :-

NewItem < Element,

!,

insert(NewItem, Left, NewLeft).

insert(NewItem, tree(Element, Left, Right), tree(Element, Left, NewRight) :-

insert(NewItem, Right, NewRight).

Ако *NewItem* < *Element*, то вмъкването се извършва в лявото поддърво иначе в дясното поддърво. Трябва да се отбележие, че повечето работа се извършва чрез проверка в съответствие с аргументите в главата на правилото.**ЛАБОРАТОРНО УПРАЖНЕНИЕ №3**

**ТЕОРЕТИЧНИ БЕЛЕЖКИ**

Пролог включва много типове от данни: прости и сложни, вградени и дефинирани от потребителя. Простите типове данни са:

променливите, като X,MyVariable, *\_*another*\_*variable, \_

константите, като char, integer или real число, или symbol или string

Сложните типове данни позволяват обединяването на няколко отделни стойности, като едно цяло. Сложните типове се обявяват чрез име известно като функтор и един или няколко аргументи.

Функторът в Пролог не трябва да се разглежда като функция в другите езици. Функтора от друга страна не е пречка да бъдат извършвани изчисления. Това е просто име, чрез което се идентифицира сложен тип обект и обединява определен брой аргументи.

В Пролог съществуват два начина да се повтаря някоя клауза чрез механизма на възврат и рекурсия. След пропадане на цел Пролог ще търси ново решение с помощта на механизма на възврат. Рекурсията е процес, при който клаузата извиква сама себе си.

**ПРАКТИЧЕСКИ ЗАДАЧИ**

**Задача 1.**

1)Да се разгледа програмата:

**PREDICATES**

count(ulong)

**CLAUSES**

count(N):-

write('\r',N),

NewN = N+1,

count(NewN).

**GOAL**

nl,

count(0).

2)Да се стартира в този вариант

3)Да се направят необходимите корекции да се изведат числата до 20

**Задача 2.**

Да се напише рекурсивен предикат, изчисляващ xn.

**Задача 3.**

1)Да се въведе програмата:

DOMAINS

treetype = tree(string, treetype, treetype) ; empty()

PREDICATES

traverse(treetype)

CLAUSES

traverse(empty).

traverse(tree(Name,Left,Right)):-

write(Name,'\n'),

traverse(Left),

traverse(Right).

GOAL

traverse().

2)Да се запише коректно целта, в която трябва да се опише дървото, показано на схемата.



3) Да се направят необходимите корекции, обхождането на дървото да е „корен-дясно-ляво”.

**Задача 4.**

1)Като се има в предвид програмата от задача 2 да се напише програма за обхождане на показаното дърво с разпечатване на срещаните стойности:

2)Да се направят необходимите корекции и/или допълнения за да се намира и извежда сумата на възлите от дървото.

3)Да се направят необходимите корекции и/или допълнения за да се намира и извежда максималната стойност на възел в дървото.

**Задача 5.**

1)Да се въведе следната програма:

**DOMAINS**

 treetype = tree(string,treetype,treetype) ; empty()

**PREDICATES**

 create\_tree(string,treetype)

 insert\_left(treetype,treetype,treetype)

 insert\_right(treetype,treetype,treetype)

 run

**CLAUSES**

 run:-

 create\_tree("Charles",Ch),

 create\_tree("Hazel",H),

 create\_tree("Michael",Mi),

 create\_tree("Jim",J),

 create\_tree("Eleanor",E),

 create\_tree("Melody",Me),

 create\_tree("Cathy",Ca),

 insert\_left(Ch, Mi, Mi5),

 insert\_right(H, Mi2, Mi3),

 insert\_left(J, Me2, Me),

 insert\_right(E, Me2, Me6),

 insert\_left(Mi2, Ca, Ca2),

 insert\_right(Me1, Ca3, Ca4),

 write(Ca3,'\n').

**GOAL**

 run.

2)Да сенапишат коректно клаузите на предикатите create\_tree, insert\_left и insert\_right.

3)Да се направят необходимите корекции, програмата да изведе коректно дървото:



**СЕМИНАРНО УПРАЖНЕНИЕ №4**

**РАБОТА СЪС СПИСЪЦИ**

Използването и обработката на списъци е много мощна техника в Пролог, защото се манипулира с обекти, които съдържат неизвестен брой елементи.

В Пролог списъкът е обект, които съдържа неуточнен като брой елементи в себе си. Списъците съответстват приблизително на масивите в другите езици, но разликата е, че при неговото обявяване не е необходимо да се окаже колко голям е той, преди неговото използване.

Съществуват и други начини да се обединят няколко обекта в един. Ако броят на обектите, които ще се обединяват е известен, те могат да се обединят в сложен обект от данни. И дори когато броят на елементите е непредсказуем, може да се използва рекурсивен тип данни, като дърво. Използването на списъците обикновено е по-лесно, защото езика предоставя лесно означение за тях.

Списък, който съдържа числата 1, 2 и 3 се записва като:

[1, 2, 3]

Всеки член на списък се нарича елемент. Синтактично структурата списък се загражда с квадратни скоби, а всеки елемент се отделя със запетая. Няколко примера:

 [dog, cat, canary]

["valerie ann", "jennifer caitlin", "benjamin thomas"]

**Деклариране на списък**

Елементите на списък могат да бъдат от всякакъв тип, включително и други списъци. Тъй като елементите на списъка трябва за принадлежат на един и същ тип в допълнение към декларацията на списъка трябва да се обяви и типа на елементите на списъка, ако той не е измежду стандартните за езика. Това се извършва в раздела DOMAINS. Например:

DOMAINS

 elementlist = elements\*

 elements = …

Тук elements трябва да се приравни или към прост тип данни, например integer, real, и symbol, или към набор от алтернативи, отбелязани с различни функтори.

Ако е необходимо да се декларира списък от цели числа декларацията изглежда по този начин:

DOMAINS

 integerlist = integer\*

Звездичката в декларацията означава „списък от” или като цяло може да се разчете като: „списък от цели числа”.

Трябва да се отбележи, че думата list (списък) няма специално значение. Всяко име, подходящо подбрано последвано от звездичка се интерпретира в списъчен контекст.

**Глава и опашка на списък**

Списъкът е сложен рекурсивен тип данни. Той се състои от две части: от глава, който е първият елемент и от опашка, която включва всички останали елементи.

**Опашката на списък е винаги списък, а главата е един елемент.**

Например:

главата на списъка [a,b,c] е a

опашката на списъка [a,b,c] е [b,c]

Интересен е случая, когато се разглежда списък от един елемент.

Например:

главата на списъка [c] е с

опашката на списъка [c] е [] ,т.е. празен списък

Ако достатъчно дълго се извежда първия елемент от опашка на списък, почти сигурно е, че ще се достигне до празен списък ([]).

**Празният списък не може да бъде разделян на глава и опашка.**

**Обработка на списъци**

В Пролог има начин за отделяне на главата на списъка от опашката. Вместо разделянето на елементите със запетая, разделянето се описва с вертикална черта ( | ).

Например:

[a,b,c] е еквивалентно на [a | [b,c]]

и ако се продължи по този начин

[a | [b,c]] е еквивалнтно на [a | [ b| [c] ] ]

което е еквивалнтно на [a | [b | [c| [] ] ] ]

Глави и опашки на списък

|  |  |  |
| --- | --- | --- |
| Списък | Глава | Опашка |
| ['a', 'b', 'c'] | 'a' | ['b', 'c'] |
| [ 'a' ] | 'a' | [] /\* празен списък \*/ |
| [ ] | неопределена | неопределена |
| [[1, 2, 3], [2, 3, 4], []] | [1, 2, 3] | [[2, 3, 4], []] |

.

Унификация на списъци

|  |  |  |
| --- | --- | --- |
| Списък 1 | Списък 2 | Обвързване на променливите |
| [X, Y, Z] | [иван, яде, сладолед] | X=иван, Y=яде, Z=сладолед |
| [7] | [X | Y] | X=7, Y=[] |
| [1, 2, 3, 4] | [X, Y | Z] | X=1, Y=2, Z=[3,4] |
| [1, 2] | [3 | X] | грешка |

**Използване на списък**

Тъй като списъкът е сложен рекурсивен тип данни, към него се прилага и рекурсивен алгоритъм за неговата обработка. Основният начин за работа със списъци е извършването на някакво действие с всеки елемент докато се стигне края на списъка.

Алгоритъма, който стандартно се използва включва обикновено две клаузи. С едната се описва действието, което се извършва със самия списък(този който има глава и опашка). С другата се описва действието, когато се достигне до празен списък.

**Извеждане на списък**

Със следващата програма се показва извеждането на елементите на списък:

**DOMAINS**

list = integer\* /\* списък от цели числа \*/

**PREDICATES**

write\_a\_list(list)

**CLAUSES**

write\_a\_list([]). /\* Ако списъкът е празен, не прави нищо. \*/

write\_a\_list([H|T]):-

 /\* Обвържи главата с H опашката с T, и тогава... \*/

write(H),nl,

write\_a\_list(T).

**GOAL**

write\_a\_list([1, 2, 3]).

Резултата е:

1

2

3

yes

В програмата се използват две клаузи write\_a\_list, които на естествен език трябва да се разбират по следния начин:

Ако списъка е празен, не прави нищо.

Иначе за да изведе списъка, се разпечатва главата (която е самостоятелен елемент) и се продължава с разпечатването на опашката.

Първоначално целта е

 write\_a\_list ([1, 2, 3]).

Целта съвпада с втората клауза и заради това H=1 и T=[2, 3]. След това се разпечатва 1 и write\_a\_list се извиква рекурсивно с опашката на списъка:

write\_a\_list ([2, 3]). /\* Така трябва да се разбира write\_a\_list(Т) \*/

Това рекурсивно обръщение отново съвпада с втората клауза и този път H=2 и T=[3]. След това се разпечатва 2 и write\_a\_list се извиква рекурсивно с опашката на списъка:

write\_a\_list ([3]).

Сега следва въпроса с коя от клаузите ще се обвърже тази заявка? Дори и с един елемент в списъка [3] може да се определи коя е главата и коя е опашката, т.е. главата е 3, а опашката []. Това означава, че това обръщение отново ще съвпадне с втората клауза и поради това H=3 , а T=[]. Извежда се 3 и write\_a\_list се извиква рекурсивно по този начин:

write\_a\_list([]).

Сега вече се вижда необходимостта от първата клауза на това правило. Втората клауза не съвпада с това обръщение, защото празния списък не може да се дели на глава и опашка. Ако не беше описана първата клауза целта няма да може се удовлетвори. Първата клауза съвпада със заявката и целта се удовлетворява без да е необходимо да се прави нещо друго.

**Преброяване на елементите на списък**

В някои случаи е необходимо определянето на броя на елементите на даден списък. Това се извършва със следния алгоритъм:

Дължината на празния списък ( [] ) е 0.

Дължината на всеки друг списък е дължината на опашката плюс 1.

Следната програма илюстрира този алгоритъм

**DOMAINS**

list = integer\* /\* в този случай типа не е от значение \*/

**PREDICATES**

length\_of(list,integer)

**CLAUSES**

 length\_of([], 0).

 length\_of([\_|T],L):-

length\_of(T,TailLength),

L = TailLength + 1.

Трябва да се обърне специално внимание на втората клауза, в която [\_|T] ще съвпадне с всеки непразен списък, като по този начин Т ще се обвърже с неговата опашка. Стойността на главата на списъка не е от значение, стига да съществува ще бъде отброена като един елемент.

Ако програмата се стартира със следната цел, да се проследи стъпките, през които преминава удовлетворяването й:

length\_of( [ 1, 2, 3 ], L ).

Подадената цел ще съвпадне с втората клауза и T=[2,3]. Следващата стъпка е да се изчисли дължината на Т. Когато това бъде направено TailLength ще има стойност 2 и компютъра ще добави 1 към тази стойност и L ще се обвърже с 3.

През какви стъпки преминават тези изчисления: Стъпката, през която се търси дължината на [2,3] се отъждествява с целта:

length\_of( [2,3], TailLength).

Рекурсивния алгоритъм извиква отново length\_of и този път целта съвпада с втората клауза и T се обвързва с [3], а TailLength с L в главната цел.

Задължително трябва да се направи едно уточнение, а именно TailLength в отделните цели не трябва да се възприема по един и същ начин. Причината за това е, че всяко рекурсивно обръщение към клауза създава свой набор от променливи.

Оттук нататък задачата се свежда до намирането на дължината на [3], което е 1, след това се прибавя 1 за да се определи дължината на [2,3], което е 2.

Когато length\_of се извика рекурсивно за да търси дължината на [3], опашката на този списък е празния и T ще се обвърже с []. По този начин целта сега е:

length\_of( [], TailLength)

Сега вече целта съвпада с първата клауза и TailLength се обвързва с 0. Според рекурсивния алгоритъм дъното е достигнато и последователно в обратен ред стойността на променливата, връщаща дължината на списъка ще получава необходимата стойност. Нагледно алгоритъма е представен по следния начин:

length\_of([1, 2, 3], L1).

length\_of([2, 3], L2).

length\_of([3], L3).

length\_of([], 0).

L3 = 0+1 = 1.

L2 = L3+1 = 2.

L1 = L2+1 = 3.

**Членство на елемент на списък**

Много често се налага да се използва предикат, чрез които може да се установи връзката членство на елемент в списък. Това може да се опише чрез предиката member:

member(name, namelist) /\* ”name” е член на „namelist” \*/

В следващата програма се описва действието на предиката:

**DOMAINS**

 namelist = name\*

name = symbol

**PREDICATES**

nondeterm member(name, namelist)

**CLAUSES**

 member(Name, [Name|\_]).

 member(Name, [\_|Tail]):-

member(Name,Tail).

С първата клауза се изследва главата на списъка. Ако главата на списъка е равна на Name (търсената стойност), то проверката за членство е успяла. Подаването на опашката като анонимна променлива оказва, че тя не е определяща при търсенето на съвпадение. С втората клауза вече се стартира търсенето на съвпадение в опашката на списъка.

**Добавяне на един списък към друг**

За да може да се реализира със средствата на Пролог добавянето на един списък към друг е необходимо да се дефинира предикат с три аргумента:

append(List1, List2, List3)

Схемата на обединяването е първите два списъка ще се обединят в третия. Алгоритъмът за това действие е отново рекурсивен.

Ако първият списък List1 е празен, резултата от добавянето на List1 и List2 ще бъде самия List2.

append( [], List2, List2).

Ако List1 не е празен, обединението на двата списъка ще се извърши с прехвърляне на главата на List1 като глава на List3.

append([H|L1], List2, [H|L3]):-

 append(L1, List2, L3).

Като цяло действието на предиката може да се опише по следния начин: Докато списъка List1 не е празен, рекурсивно правило премества по един негов елемент в списъка List3. Когато списъка List1 е празен, поради действието на първата клауза, List2 се поставя в края на List3.

Следва програма, която използва този предикат:

**DOMAINS**

 integerlist = integer\*

**PREDICATES**

 append(integerlist,integerlist,integerlist)

**CLAUSES**

 append([],List,List).

 append([H|L1],List2,[H|L3]):-

 append(L1,List2,L3).

**GOAL**

append([1, 2, 3], [5, 6], L).

**Търсене на път.**

Да се допусне, че трябва да се разработи програма която помага за намиране на най-добрия маршрут между два града в САЩ.

В началото се строи прототип с помощта на Пролог, който в последствие може да бъде разширен. Трябва да може да се отговаря на следните въпроси:

Има ли път от дадения град до другия?

Колко е разстоянието от един град до друг?

Кой е най-краткия път между два града?

Кои са всички възможни маршрути от град до град?

Тази фигура показва опростена карта на пътищата за прототипа:



**Фиг. 2.1.** Карта на прототипа

Пътят между два града ще се описва чрез предиката ***road,*** като аргументите му са градовете и разстоянието помежду им в километри.

Определянето на маршрут между два града се реализира чрез предиката ***route.*** Търсенето на маршрут е процес, при които се извършва преминаване през няколко междинни пункта. Обхождайки маршрута, чрез третия параметър ***distance,*** може да се определи общото разстояние на маршрута. Предикатът ***route*** е дефиниран рекурсивно, така че маршрута може да се състои от един участък от пътя. Тогава дължината на маршрута е равна на дължината на този участък

От друга страна може да се продължи маршрута от град Town1 до град Town2, достигайки в началото до град Х, а след това преминавайки по друг път от град Х до град Town2. Общата дължина на маршрута е равна на сумата от разстоянията от Town1 до Х и от Х до Town2, както е показано във втората реализация на ***route***.

Следва определяне на цел –

*route(gordon,tampa, X).*

**DOMAINS**

 town = symbol

 distance = integer

**PREDICATES**

 nondeterm road(town,town,distance)

 nondeterm route(town,town,distance)

**CLAUSES**

*% път между два града и разстоянието между тях*

 road(gordon,houston,100).

 road(houston,kansas\_city,120).

 road(kansas\_city,tampa,80).

*% има маршрут от Town1 до Town2, ако има път между тях*

route(Town1,Town2,Distance):-

 road(Town1,Town2,Distance).

*% има маршрут от Town1 до Town2, ако има път от Town1 до Х и се търси маршрут % от X до Town2*

route(Town1,Town2,Distance):-

 road(Town1,X,Dist1),

 route(X,Town2,Dist2),

 Distance=Dist1+Dist2,!.

**GOAL**

route(tampa, gordon, X).

Трябва да се отбележи един недостатък, че ако за цел се посочи

***route(houston,X, D).***

Пролог няма да посочи, че има път между gordon и houston, защото не е определено ако има път между Town1 и Town2 е все едно, че има път между Town2 и Town1.

В секцията на предикатите следва да се добави нов предикат ***connected,*** които отстранява този недостатък

predicates

 nondeterm connected(town,town,distance)

В секцията на клаузите е неговото описание:

clauses

 connected(X,Y,Dist):-

 road(X,Y,Dist).

 connected(X,Y,Dist):-

 road(Y,X,Dist).

Търсенето на път е свързано и с проверка за това дали не е разглеждано състоянието, в което сме в момента. Ако това не се съобрази е възможно рекурсията да не намери гранично условие.

За целта в Пролог се използват предикати, предназначени за работа със списъци. При работа със списък, елементите му се обхождат един по един, като този които разглеждаме се нарича „глава”, а остатъка „опашка”.

**domains**

townlist=town\*

**predicates**

% предиката проверява дали града *town* е в списъка *townlist*

 nondeterm member(town,townlist)

**clauses**

 member(X,[X|\_]):-!. % ако елемента съвпадне с главата, то той е елемент на списъка

 member(X,[\_|L]):- % ако елемента не съвпадне с главата, то той се търси в опашката

 member(X,L).

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 Има маршрут от Town1 до Town2 ако са директно свързани с път. Иначе се търси град X , който е свързан с Town1. След това се търси маршрут от X до Town2, като се провери дали X не е част от градовете, които са посетени вече ! (Иначе рекурсията ще работи безкрайно !)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**predicates**

 nondeterm route(town,town,townList,townList,distance)

**clauses**

 route(Town,Town,VisitedTowns, VisitedTowns, 0) :- !.

 route(Town1,Town2,VisitedTowns,ResultVisitedTowns,Distance) :-

*% проверка за свързаност между Town1 и произволен град Х*

 connected(Town1,X,Dist1),

*% проверка града да не е измежду тези, за които е направена проверка*

 not(member(X,VisitedTowns)),

*% търсене на маршрут от намерения град Х до целевия Town2, като Х се включва*

*% в посетените градове*

 route(X,Town2,[X|VisitedTowns],ResultVisitedTowns,Dist2),

*% сумиране на разстоянията с определяне на цялата дължина*

 Distance=Dist1+Dist2.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 Връщане назад към вече получените решения и търсене на възможни маршрути.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**predicates**

 procedure showAllRoutes(town,town)-(i,i)

 procedure write\_route(town FirstTown,townList,distance)

 procedure reverse\_list(townList InList, townList Tmp, townList Reversed)

**clauses**

 showAllRoutes(Town1,Town2):-

% извежда подходящо съобщение

 write("All routes from ",Town1," to ",Town2," are:\n"),

% започва търсене на маршрут, с отбелязване на преминатите градове и сумиране на

% разстоянията

 route(Town1,Town2, [Town1] ,VisitedTowns, Dist),

% разпечатва намерените пътища

 write\_route(Town1,VisitedTowns,Dist),nl, % nl - нов ред

% със служебния предикат fail се търси не само едно, а всички възможни решения

% неговото действие би могло да се опише по следния начин – когато започне да се

% търси удовлетворяване, при наличие на fail целта винаги “пропада” и започва ново

% удовлетворение

 fail.

% с неуточнените променливи \_ се указва, всички възможни случаи

showAllRoutes(\_,\_).

% предикат за разпечатване на маршрут от град до град

 write\_route(Town1,[Town1|VisitedTowns],Dist):-

 !,

 Towns = [Town1|VisitedTowns],

 write(" ",Towns," --> ",Dist),nl.

 write\_route(\_,VisitedTowns,Dist):-

% при намиране на отделните градове, участващи в маршрута, те се записват в

% обратен на стандартния за разпечатване ред. Поради тази причина се налага

% обръщане на списъка с цел правилното му извеждане

 reverse\_list(VisitedTowns, [], VisitedTowns\_Reversed),

 write(" ",VisitedTowns\_Reversed," --> ",Dist),nl.

%предикат за обръщане на списък

% ако сме обходили целия списък спираме - !

 reverse\_List([],LIST,LIST):-!.

% в противен случай се местят един по един елементите от първия списък в празен

% и така се получава правилната подредба

 reverse\_List([H|SeenListRest],Interm,SeenList):-

 reverse\_List(SeenListRest,[H|Interm],SeenList).

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 Показване на най-късия маршрут

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**predicates**

 procedure showShortestRoutes(town,town)-(i,i)

 nondeterm shorterRouteExist(town,town,distance)-(i,i,i)

**clauses**

 showShortestRoutes(Town1,Town2):-

% извежда подходящо съобщение

 write("Shortest routes between ",Town1," to ",Town2," is:\n"),

% търси маршрут

 route(Town1,Town2, [Town1] ,VisitedTowns, Dist),

% ако няма по-къс маршрут от намерения

 not(shorterRouteExist(Town1,Town2,Dist)),

% се извежда съдържанието на списъка

 write\_route(Town1,VisitedTowns,Dist),nl,

 fail.

 showShortestRoutes(\_,\_).

% предикат за намиране на по-къс маршрут

 shorterRouteExist(Town1,Town2,Dist):-

 route(Town1,Town2, [Town1] ,\_, Dist1),

 Dist1<Dist,!.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 Старт на програмата

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**goal**

 showAllRoutes("gordon", "tampa"),nl,

 showShortestRoutes("gordon", "tampa").

**ЛАБОРАТОРНО УПРАЖНЕНИЕ №4**

**ТЕОРЕТИЧНИ БЕЛЕЖКИ**

Списъците са обекти, който могат да съдържат неопределен брой аргументи. Декларирането им изисква поставяне на звездичка след дефиниран предварително тип.

Списъците са сложен рекурсивен тип данни, който са съставени от глава и опашка. Главата е първият елемент, а главата е всичко останало от списъка. Опашката винаги е списък, а главата е един елемент. Списъкът може да не съдържа нито един елемент. В този случай става дума за празен списък, който се отбелязва с [].

Елементите на списък могат да бъдат от всякакъв тип, включително и други списъци. Всички елементи на списъка трябва да принадлежат на един тип. Най общия вид за деклариране на списък е:

DOMAINS

 elementlist = elements\*

 elements = …

където elements е един от стандартните типове.

Допустимите символи, използвани в списъците с цел разделяне на главата и тялото са: запетая, [, ], |. Например списъка:

[a, b, c, d]

може да се напише по следния начин:

[a | [b, c, d]] или

[a, b | [c, d]] или

[a, b, c | [d]] или

[a | [b | [c, d]]] или

[a | [b | [c | [d]]]] или

[a | [b | [c | [d | []]]]]

При обработването на списък се използва рекурсивен алгоритъм, който премахва главата на списъка(като обикновено извършва някакво действие с нея) и продължава докато списъка остане празен.

**ПРАКТИЧЕСКИ ЗАДАЧИ**

**Задача 1.**

Да се напише програма, която извежда всеки от елементите на нов ред на задададен списък в обратен ред, като не се използва допълнителен предикат за обръщане на списък.

**Задача 2.**

Да се напише програма, която намира и извежда n-тия елемент на списък. **GOAL**

**length\_of( [ 1, 2, 3, 4, 5 ], L ).**

**Задача 3.**

Да се напише програма, която намира и извежда сумата на елементите на списък.

**DOMAINS**

**list = integer\***

**PREDICATES**

**sum\_of(list,integer)**

**CLAUSES**

 **sum\_of([], 0).**

 **sum\_of([X|T],L):-**

**sum\_of(T,Sum),**

**L = Sum + X.**

**GOAL**

**sum\_of([1,2,3,4,5],Sum).**

**Задача 4.**

Да се напише програма, която добавя единица, към всеки от елементите на списък от цели числа.

Пример: Ако зададеният списък е [1,3,6,4,2], то резултатният е [2,4,7,5,3].

**DOMAINS**

**list = integer\***

**PREDICATES**

**add1(list,list)**

**CLAUSES**

**add1([], []). /\* гранично условие \*/**

**add1([Head|Tail],[Head1|Tail1]):- /\* отделя главата \*/**

 **/\* за останалата част на списъка \*/**

**Head1= Head+1, /\* добавя 1 към главата \*/**

**add1(Tail,Tail1). /\* обработва опашката \*/**

**GOAL**

 **add1([1,2,3,4], NewList).**

**Задача 5.**

Да се напише програма, която добавя един списък към друг. Да се зададе подходяща цел, така че ако има два списъка [1,2] и [3] да се получат два [1,2,3] и [1,2,3,1,2,3]

**append([1, 2], [3], L), append(L, L, LL).**

**Задача 6.**

Да се напише програма на Пролог, която:

1)Описва възможните връзки между градовете в североизточна България и разстоянията в километри като се използва следната карта:



2)Намира маршрут между два непряко свързани града и разстоянието между тях.

3)Отпечатва всички възможни маршрути между два града и разстоянието между тях.

4)Намира и отпечатва най-краткият маршрут между два града и разстоянието между тях.

**СЕМИНАРНО УПРАЖНЕНИЕ №5**

**Търсене при ограничения**

# Задача 1. Приключения в опасната пещера

Нека да се предположи, че във вътрешността на една пещера има голямо златно съкровище. Много хора са правили опити да го намерят, но без успех. Пещерата представлява лабиринт от галерии, свързващи различни зали, в които се намират опасни същества, подобни на чудовища или разбойници. Необходимо е да се знае фактът, че цялото съкровище се намира в една от залите на пещерата. Кой маршрут трябва да се следва, за да се стигне до съкровището и да се избегнат опасностите?



**Фиг.1.** Карта на пещерата

Ще бъде конструирано представяне на картата на пещерата в Пролог, за да се осигури откриването на сигурен маршрут в пещерата. Всяка галерия се описва с един факт. Правилата са дадени посредством предикатите *go* и *route*.

**DOMAINS**

% зали

 room = symbol

% списък от зали

 roomlist = room\*

**PREDICATES**

% съществува галерия (път) между две галерии - всички възможности

nondeterm gallery(room,room)

% съседство на две зали – две зали са свързани по между си независимо от посоката,

% от която се проследяват

nondeterm neighborroom(room,room)

% списък от зали, през които не трябва да се преминава

avoid(roomlist)

% предикат за стартиране на търсенето, при заложеното ограничение

 nondeterm go(room,room)

% предикат за проследяване на галериите, като се записват преминатите в *roomlist*

 nondeterm route(room,room,roomlist)

% стандартен предикат за определяне на членство на елемент към списък

 nondeterm member(room,roomlist)

**CLAUSES**

% описание на галериите

 gallery(entry,monsters). gallery(entry,fountain).

 gallery(fountain,hell). gallery(fountain,food).

 gallery(exit,gold\_treasure). gallery(fountain,mermaid).

 gallery(robbers,gold\_treasure). gallery(fountain,robbers).

 gallery(food,gold\_treasure). gallery(mermaid,exit).

 gallery(monsters,gold\_treasure). gallery(gold\_treasure,exit).

%определяне на съседство

 neighborroom(X,Y):-gallery(X,Y).

 neighborroom(X,Y):-gallery(Y,X).

%списък от залите, през които не трябва да се преминава

 avoid([monsters,robbers]).

% търсене на път от начална до крайна зала

 go(Here,There):-route(Here,There,[Here]).

 go(\_,\_).

% описание на действията, когато се е достигнало целевото състояние – елемент

% на списъка на обходените зали да е залата *“gold\_treasure”*

 route(Room,Room,VisitedRooms):-

 member(gold\_treasure,VisitedRooms),

 write(VisitedRooms),nl.

% търсене на маршрут от зала до зала

 route(Room,Way\_out,VisitedRooms):-

% намира се съсед на Room

 neighborroom(Room,Nextroom),

% зарежда се списъка със зали, през които не се преминава

 avoid(DangerousRooms),

% проверка за това дали съседаната зала *Nextroom* не е из между забранените

 not(member(NextRoom,DangerousRooms)),

% проверка за това дали залата *Nextroom* не е из между преминатите

 not(member(NextRoom,VisitedRooms)),

% търсене на маршрут от залата *NextRoom* до целевата с добавяне на *NextRoom* към

% залите, през които се е преминало

 route(NextRoom,Way\_out,[NextRoom|VisitedRooms]).

% стандартен предикат за определяне на членство

 member(X,[X|\_]).

 member(X,[\_|H]):-member(X,H).

**GOAL**

 go(entry,exit).

# Задача. 2 Преминаването на фермера, вълка, коза и зелка през река с лодка

Описание на задачата: Фермерът и неговата коза, вълк и зелка трябва да преминат през река. За преминаването има лодка, но има място най-много за двама, като фермерът може да премине и сам. Ако козата и зелката са на едно и също място без фермера, то зелката ще бъде изядена. Аналогично, ако вълка и козата са на едно и също място без фермера, то козата ще бъде изядена. След поредица от преминавания на реката всички трябва безопасно да преминат от едната на другата страна на реката.

При описанието на местоположението на всеки един от участниците се използва предиката *state.*

state ( Farmer, Wolf, Goat, Cabbage)

Усложнението на проблема е, че всяко едно от отделните състояния може да бъде използвано последователно, като някои от тях са неподходящи. Тези състояния се проверяват с предикатите *unsafe* и “*member*”. Предикатът "*go*" стартира търсенето на всички състояния от първоначалното до крайното.

go( state(east,east,east,east), state(west,west,west,west) ).

**DOMAINS**

% отделните състояния, описани условно от едната и от другата страна на реката

 LOC = east;

 west

% описва текущото състояние на всички участници

 STATE = state(LOC farmer,LOC wolf,LOC goat,LOC cabbage)

% обходените ситуации са списък от състояния

 PATH = STATE\*

**PREDICATES**

% използва се за стартиране на алгоритъма

 go(STATE,STATE)

% намира път от едно състояние към друго

 path(STATE,STATE,PATH,PATH)

% сменя състоянията от едно към друго

 nondeterm move(STATE,STATE)

% извършва преминаването на двама от едната страна към другата на реката

 opposite(LOC,LOC)

% описва нестабилните (ограничителните) състояния

 nondeterm unsafe(STATE)

% проверка за преминато състояние

 nondeterm member(STATE,PATH)

% извежда целия път

 write\_path(PATH)

% извежда промяната на състоянията

 write\_move(STATE,STATE)

**CLAUSES**

 go(StartState,GoalState):-

% започва търсене на последователни състояния от началното към крайното

 path(StartState,GoalState,[StartState],Path),

 write("A solution is:\n"),

% извежда целия път с отделните състояния

 write\_path(Path).

% търси отделните правилни състояния до целевото

 path(StartState,GoalState,VisitedPath,Path):-

% намира следващо състояние

 move(StartState,NextState),

% проверка за неподходящо състояние

 not(unsafe(NextState)),

% проверка за преминато състояние

 not(member(NextState,VisitedPath)),

% търси път от това, преминало успешно през проверките състояние, към ново

 path(NextState,GoalState,[NextState|VisitedPath],Path),

 !.

% ако текущото намерено е целевото, то решение е намерено

 path(GoalState,GoalState,Path,Path).

% описва отделните смени на състоянията и преходи на участниците

 move(state(X,X,G,C),state(Y,Y,G,C)):-

 opposite(X,Y). % Прехвърля фермера и вълка

 move(state(X,W,X,C),state(Y,W,Y,C)):-

 opposite(X,Y). % Прехвърля фермера и козата

 move(state(X,W,G,X),state(Y,W,G,Y)):-

 opposite(X,Y). % Прехвърля феремера и зелката

 move(state(X,W,G,C),state(Y,W,G,C)):-

 opposite(X,Y). % Прехвърля се само фермера

% описва промените на местото на участник от двете страни на реката

 opposite(east,west).

 opposite(west,east).

% описва първото неподходящо състояние – вълка и козата са на едно и също място

 unsafe(state(F,X,X,\_)):-

 opposite(F,X),

 !.

% описва второто неподходящо състояние – козата изяжда зелката

 unsafe(state(F,\_,X,X)):-

 opposite(F,X),

 !.

% предикат за определяне на членство на елемент към списък

 member(X,[X|\_]):-

 !.

 member(X,[\_|L]):-

 member(X,L).

% извежда пътя от всички състояния

 write\_path([H1,H2|T]):-

% от списъка със състояния взема две и ги извежда

 write\_move(H1,H2),

% продължава с остатъка

 write\_path([H2|T]).

% докато списъка от състояния остане празен

 write\_path([]).

% извежда подходящо преходите на участниците

 write\_move(state(X,W,G,C),state(Y,W,G,C)):-

 !,

 write("The farmer crosses the river from ",X," to ",Y),

 nl.

 write\_move(state(X,X,G,C),state(Y,Y,G,C)):-

 !,

 write("The farmer takes the Wolf from ",X," of the river to ",Y),

 nl.

 write\_move(state(X,W,X,C),state(Y,W,Y,C)):-

 !,

 write("The farmer takes the Goat from ",X," of the river to ",Y),

 nl.

 write\_move(state(X,W,G,X),state(Y,W,G,Y)):-

 !,

 write("The farmer takes the cabbage from ",X," of the river to ",Y),

 nl.

**GOAL**

 go(state(east,east,east,east),state(west,west,west,west)),

 write("solved").

**ЛАБОРАТОРНО УПРАЖНЕНИЕ №5**

**Tърсене при ограничения**

# ПРАКТИЧЕСКИ ЗАДАЧИ

#  Задача 1. Приключения в опасната пещера



**Фиг.1.** Карта на пещерата

1) Да се въведе следната програма:

**DOMAINS**

 room = symbol

 roomlist = room\*

**PREDICATES**

nondeterm gallery(room,room)

nondeterm go(room,room)

nondeterm route(room,room,roomlist)

**CLAUSES**

 go(Here,There):-route(Here,There,[Here]).

 go(\_,\_).

 route(Room,Room,VisitedRooms):-

 member(gold\_treacure,VisitedRooms),

 write(VisitedRooms),nl.

 route(Room,Way\_out,VisitedRooms):-

 gallery(Room,Nextroom),

 not(member(NextRoom,VisitedRooms)),

 route(NextRoom,Way\_out,[NextRoom|VisitedRooms]).

**GOAL**

 go(entry,exit).

2)Като се има в предвид картата на пещерата, да се опишат всички възможни връзки между галериите.

3) Да се направят необходимите корекции(синтактични и логически), за да може програмата да работи нормално.

4)Да се добави следната клауза

 gallery(mermaid, gold\_treasure).

Да се направи необходимата корекция в програмата за де се изведат всички възможни решения.

5)Да се добави предикат, чрез който обходените галерии да се изведат в правилния ред.

# Задача 2. Преминаването на фермера, вълка, коза и зелка през река с лодка

1)Да се въведе следната програма:

**DOMAINS**

 LOC = east;

 west

 STATE = state(LOC farmer,LOC wolf,LOC goat,LOC cabbage)

 PATH = STATE\*

**PREDICATES**

 go(STATE,STATE)

 path(STATE,STATE,PATH,PATH)

 nondeterm move(STATE,STATE)

 opposite(LOC,LOC)

 nondeterm unsafe(STATE)

 nondeterm member(STATE,PATH)

 write\_path(PATH)

 write\_move(STATE,STATE)

**CLAUSES**

 go(StartState,GoalState):-

 path(StartState,GoalState,[StartState],Path),

 write("A solution is:\n"),

 write\_path(Path).

 path(StartState,GoalState,VisitedPath,Path):-

 move(StartState,NextState),

 not(unsafe(NextState)),

 not(member(NextState,VisitedPath)),

 path(NextState,GoalState,[NextState|VisitedPath],Path),

 !.

 path(GoalState,GoalState,Path,Path).

 move(state(X,X,G,C),state(Y,Y,G,C)):-

 opposite(X,Y).

 unsafe(state(F,X,X,\_)):-

 opposite(F,X),

 !.

 write\_path([H1,H2|T]):-

 write\_move(H1,H2),

 write\_path([H2|T]).

 write\_path([]).

 write\_move(state(X,W,G,C),state(Y,W,G,C)):-

 !,

 write("The farmer crosses the river from ",X," to ",Y),

 nl.

 write\_move(state(X,X,G,C),state(Y,Y,G,C)):-

 !,

 write("The farmer takes the Wolf from ",X," of the river to ",Y),

 nl.

 write\_move(state(X,W,X,C),state(Y,W,Y,C)):-

 !,

 write("The farmer takes the Goat from ",X," of the river to ",Y),

 nl.

 write\_move(state(X,W,G,X),state(Y,W,G,Y)):-

 !,

 write("The farmer takes the cabbage from ",X," of the river to ",Y),

 nl.

**GOAL**

 go(state(east,east,east,east),state(west,west,west,west)),

 write("solved").

2)Да се извършат необходимите корекции в действието на предикатите move и unsafe.

3)Да се добавят необходимите клаузи за липсващите предикати.**СЕМИНАРНО УПРАЖНЕНИЕ № 6**

**Евристични методи за търсене в пространството на състоянията**

І. Задача за осемте царици

1. Обща постановка

Задачите за търсене в пространството на състоянията, традиционно се смятат за интелектуални. Един от типовете задачи за търсене е търсене на цел при ограничителни условия. Същността на такива задачи е да се построи описание на търсеното целево състояние, което удовлетворява дадено множество от ограничителни условия.

От този тип са задачата за осемте царици (8 queens), решаване на криптограми, съставяне на разписания, проектантски задачи (с различни ограничения откъм време, пространство, цена u пр.), задача за зебрата, задача за триъгълниците u много други.

2. Същност на задачата

Задачата за осемте царици е да се поставят 8 царици върху шахматна дъска по такъв начин, че нито една царица да не попада под обсега на другите. Това означава, че никои две царици не могат да бъдат поставени на един и същ ред, колона или диагонал.

Разсъжденията ще бъдат ориентирани към търсене на решение за дъска с размер 8Х8. Обаче, както често става в програмирането, ключът към решението е по-лесно да се намери, разглеждайки по-обща постановка на задачата. Решението на по-общата задача често е по-лесно да се формулира спрямо решението на частната, изходната задача, а след това изходната задача да се реши като частен случай на общата задача. Така, задачата ще бъде преформулирана за N царици върху дъска с N x N полета, където N може да бъде всяко естествено число, включително и нула. Основната част на такъв подход е да се намери правилното обобщение на задачата. За тази задача могат да се използват няколко различни идеи.

3. Решение

Вариант 1

Един начин за решаване на задачата е да се номерират редовете и колоните на шахматната дъска от 1 до N. За да се номерират диагоналите, можем да ги разделим на два типа, като по този начин диагоналът да може уникално да се определи чрез типа и стойността, изчислена от номерата на неговите ред и колона.

Diagonal = N + Column - Row (Тип 1 – наклонени наляво)

Diagonal = Row + Column - 1 (Тип 2 – наклонени надясно)

На фигурата е показано номерирането на диагонали от тип 2 (наклонени надясно) върху дъска 4Х4. Номерацията на колоните започва отляво надясно, а на редовете – отгоре надолу.



**фиг.1** Номерирането на диагонали от тип 2

Описание на задачата

За да се реши задачата за N царици с програма на Пролог, трябва да се запише кои редове, колони и диагонали са незаети, а също да се отбележи къде са разположени цариците.

Мястото на една царица се описва чрез номера на реда и номера на колоната в секцията domain

**queen = q(integer, integer)**

Тази декларация представя позицията на една царица. Трябва да се използва списък, за да се опишат повече позиции. Това се дефинира така:

queens = queen\*

По същия начин, се нуждаем от няколко списъка от числов тип, указващи редовете, колоните и диагоналите, незаети от цариците. Те се описват чрез:

freelist = integer\*

Ще третираме шахматната дъска като единичен обект със следната декларация в секцията domain:

board = board(queens, freelist, freelist, freelist, freelist)

Четирите параметъра freelist представят свободните редове, колони и диагонали от тип 1 и тип 2.

За да разберем как работи това представяне, ще разгледаме 4х4 дъска в две положения:

1. Без царици

board([], [1,2,3,4], [1,2,3,4], [1,2,3,4,5,6,7],[1,2,3,4,5,6,7])

2. С една царица в горния ляв ъгъл

board([q(1,1)], [2,3,4], [2,3,4], [1,2,3,5,6,7], [2,3,4,5,6,7])

Задачата може да се реши чрез описване на връзката (релацията) между празната дъска и дъска с N царици. Дефинираме предиката:

placeN(integer, board, board)

Цариците се разполагат една по една, докато заемат всеки ред и колона. Това се описва в клауза, в която двата списъка на свободните редове и свободните колони са празни:

placeN(\_, board(D, [], [], X, Y), board(D, [], [], X, Y)) :- !.

placeN(N, Board1, Result) :-

 place\_a\_queen(N, Board1, Board2),

 placeN(N, Board2, Result).

Във втората клауза предикатът place\_a\_queen дава връзка между състояние 1 (Board1) и състояние 2 (Board2). Board2 има една царица повече от Board1. Използва се следната декларация на предиката:

place\_a\_queen(integer, board, board)

Същността на задачата за N царици е в това, как да се добавят допълнителни царици докато се разположат успешно, започвайки с празна дъска. За да се реши този проблем, трябва да се добавя новата царица към списъка на вече разположените царици:

 [q(R, C)|Queens]

Измежду оставащите свободни редове, необходимо е да се намери ред R, където може да се постави следващата царица. В същото време трябва да се премахне R от списъка на свободните редове, което се отразява в един нов списък на свободните редове NewR. Това се формулира така:

findandremove(R, Rows, NewR)

Съответно, трябва да се намери и премахне свободна колона C. От R и C се изчисляват номерата на заетите диагонали. След това може да се определи дали D1 и D2 са сред свободните диагонали.

Това е клаузата place\_a\_queen:

place\_a\_queen(N,

board(Queens, Rows, Columns, Diag1, Diag2),

 board([q(R,C)|Queens], NewR, NewC, NewD1, NewD2)) :-

 findandremove(R, Rows, NewR),

 findandremove(C, Columns, NewC),

 D1=N+S-R,

 findandremove(D1, Diag1, NewD1),

 D2=R+S-1,

 findandremove(D2, Diag2, NewD2).

По-долу е целият текст на програмата, написана на Пролог:

domains

 queen = q(integer, integer)

 queens = queen\*

 freelist = integer\*

 board = board(queens, freelist, freelist, freelist, freelist)

predicates

 nondeterm placeN(integer, board, board)

 nondeterm place\_a\_queen(integer, board, board)

 nondeterm nqueens(integer)

 nondeterm makelist(integer, freelist)

 nondeterm findandremove(integer, freelist, freelist)

 nextrow(integer, freelist, freelist)

clauses

 nqueens(N):-

 makelist(N,L),

 Diagonal=N\*2-1,

 makelist(Diagonal,LL),

 placeN(N,board([],L,L,LL,LL),Final),

 write(Final).

 placeN(\_,board(D,[],[],D1,D2),board(D,[],[],D1,D2)):-

 !.

 placeN(N,Board1,Result):-

 place\_a\_queen(N,Board1,Board2),

 placeN(N,Board2,Result).

 place\_a\_queen(N,

 board(Queens,Rows,Columns,Diag1,Diag2),

 board([q(R,C)|Queens],NewR,NewC,NewD1,NewD2)):-

 nextrow(R,Rows,NewR),

 findandremove(C,Columns,NewC),

 D1=N+C-R,

 findandremove(D1,Diag1,NewD1),

 D2=R+C-1,

 findandremove(D2,Diag2,NewD2).

 findandremove(X,[X|Rest],Rest).

 findandremove(X,[Y|Rest],[Y|Tail]):-

 findandremove(X,Rest,Tail).

 makelist(1,[1]).

 makelist(N,[N|Rest]):-

 N1=N-1,

 makelist(N1,Rest).

 nextrow(Row,[Row|Rest],Rest).

goal

 nqueens(5),

 nl.

4. Тестване на системата

В тривиалния случай N=1 (една царица на дъска с 1 квадратче) отговорът на системата е:

board([q(1,1)],[],[],[],[])

yes

Отговорът yes показва, че целта е достигната.

При N=4 отговорът на системата е:

board([q(1,2),q(2,4),q(3,1),q(4,3)],[],[],[7,4,1],[7,4,1])

yes

При N=5 отговорът на системата е:

board([q(1,2),q(2,4),q(3,1),q(4,3),q(5,5)],[],[],[9,8,2,1],[8,7,4,1])

yes

При N=8 (класическата задача) отговорът на системата е:

board([q(1,5),q(2,7),q(3,2),q(4,6),q(5,3),q(6,1),q(7,4),q(8,8)],[],[],[15,14,11,9,4,2,1],[14,13,12,11,3,2,1])

yes

Особени случаи са N=2 и N=3. Лесно може да се провери, че в тези случаи задачата няма решение. Съобщението на системата е следното: „Препълване на стека Gstack. Няма достатъчно памет или се изпада в безкраен цикъл.”

PROGRAM ERROR. Module:OBJ\GOAL$000.PRO Pos:1427

 Message:1001 Gstack overflow. Not enough memory or an endless loop

Подобно съобщение се появява и при N=0, въпреки че в този случай има решение - няма нападение, поради липса на царици.



**фиг.2.**  Едно решение на задачата за осемте царици

5. Други варианти на решение.

При един от тях се използва представяне на позицията във вид на списък от осем елемента, всеки от които съответства на една царица. Всяко поле на дъската се описва с двойка координати ( Х и Y), където всяка координата е цяло число от 1 до 8. Такава двойка в програмата се описва с X/Y (символът / обединява координатите в един елемент на списъка). Задачата се свежда до намиране на списък от вида:

**[Xl/Yl, X2/Y2, X3/Y3, X4/Y4, X5/Y5, X6/Y6, X7/Y7, X8/Y8]**

Друг вариант разглежда положението на една царица като система с 4 координати:

x вертикала

у хоризонтала

u диагонал наляво

v диагонал надясно

Тези координати не са независими: при зададени  х  и  у,  u  и   v  се определят еднозначно (фиг. 3). Например,

    u = х - у    v = х + у



**Фиг. 3.**  Връзка между хоризонтала, вертикала и диагонали. Отбелязаното поле има следните
координати: x = 2,  у = 4,  u = 2 - 4 = -2,  v = 2 + 4 = 6.

Областите на изменение на четирите координати са:

        **Dx = [1, 2, 3, 4, 5, 6, 7, 8]
        Dy = [1, 2, 3, 4, 5, 6, 7, 8]**

**Du = [-7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7]
        Dv = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]**

Задачата за осемте царици сега може да се формулира така: да се изберат осем четворки (X, Y, U, V), влизащи в областта на изменение, така че нито един елемент да не се повтаря. Разбира се, изборът на Х и Y определя изборът на U и V

Решението при такава постановка на задачата може да бъде накратко такова: при зададени 4 области на изменение се избира позицията за първата царица, зачеркват се съответните елементи от 4-те области на изменение, а след това се използват останалите елементи от тези области за разполагане на другите царици.

**Планиращи системи.**

**Ханойски кули**

Решението на задачата за ханойските кули е класическа рекурсивна задача. В основата на старинната задача за Ханойските кули се състои от определен брой дървени дискове подредени на три стълба. Отделните стълбове са поставени на обща основа. Всички дискове са с различен диаметър и отвор по средата на диска, който е достатъчно широк за да поставен на които и да е от стълбовете. В началото всички дискове са наредени в най-левия стълб, както е показано:



Целта на задачата е да бъдат преместени всички дискове на третия диск, като се запази подредбата от първия. Преместването на дисковете се извършва един по един. Средният стълб се използва за междинен, като трябва да се има в предвид, че не може да се постави диск с по-голям диаметър, върху диск с по-малък. Сравнително лесно е да се реши задачата с два или три диска, но съобразяванията стават доста повече ако дисковете станат четири или повече.

Една примерна стратегия за решаване на задачата е следната:

1. Може да се измества само по един диск директно
2. Може да се изместят N диска в следните три стъпки:
	1. Изместват се N-1 диска в междинния стълб
	2. Измества се N диск в десния стълб
	3. Изместват се N-1 диска от междинния диск до десния

Програмата на Пролог, която решава задачата за Ханойските кули използва три предиката:

**hanoi** – с един параметър, които определя броя на дисковете

**move –** описва преместването на N диска от един стълб до друг, като използва един междинен стълб

**inform** – описва всяка стъпка на преместването на дисковете

**DOMAINS**

 loc =right;middle;left

**PREDICATES**

 hanoi(integer)

 move(integer,loc,loc,loc)

 inform(loc,loc)

**CLAUSES**

 hanoi(N):-

 move(N,left,middle,right).

move(1,A,\_,C):-

 inform(A,C),

 !.

 move(N,A,B,C):-

 N1=N-1,

 move(N1,A,C,B),

 inform(A,C),

 move(N1,B,A,C).

 inform(Loc1, Loc2):-

 write("\nMove a disk from ", Loc1, " to ", Loc2).

**GOAL**

 hanoi(3).

**ЛАБОРАТОРНО УПРАЖНЕНИЕ № 6**

**ПРАКТИЧЕСКИ ЗАДАЧИ**

**Задача 1**.

1)Да се въведе програмата:

**DOMAINS**

 queen = q(integer, integer)

 queens = queen\*

 freelist = integer\*

 board = board(queens, freelist, freelist, freelist, freelist)

**PREDICATES**

 nondeterm placeN(integer, board, board)

 nondeterm place\_a\_queen(integer, board, board)

 nondeterm nqueens(integer)

 nondeterm makelist(integer, freelist)

 nondeterm findandremove(integer, freelist, freelist)

 nextrow(integer, freelist, freelist)

**CLAUSES**

 nqueens(N):-

 makelist(N,L),

 Diagonal=N\*2-1,

 makelist(Diagonal,LL),

 placeN(N,board([],L,L,LL,LL),Final),

 write(Final).

 placeN(\_,board(D,[],[],D1,D2),board(D,[],[],D1,D2)):-

 !.

 placeN(N,Board1,Result):-

 place\_a\_queen(N,Board1,Board2),

 placeN(N,Board2,Result).

 place\_a\_queen(N,

 board(Queens,Rows,Columns,Diag1,Diag2),

 board([q(R,C)|Queens],NewR,NewC,NewD1,NewD2)):-

 nextrow(R,Rows,NewR),

 findandremove(C,Columns,NewC),

 D1=N+C-R,

 findandremove(D1,Diag1,NewD1),

 D2=R+C-1,

 findandremove(D2,Diag2,NewD2).

 findandremove(X,[X|Rest],Rest).

 findandremove(X,[Y|Rest],[Y|Tail]):-

 findandremove(X,Rest,Tail).

 makelist(1,[1]).

 makelist(N,[N|Rest]):-

 N1=N-1,

 makelist(N1,Rest).

 nextrow(Row,[Row|Rest],Rest).

**GOAL**

 nqueens(5),

 nl.

2)Да се провери резултата от действието на програмата при дъски съответно с размерност 0, 1, 2 и 3.

3)Да се добави необходимото да се изобразят не само едно решение на разполагането на цариците.

**Задача 2.**

1)Да се въведе следната програма:

**DOMAINS**

 loc =right;middle;left

**PREDICATES**

 hanoi(integer)

 move(integer,loc,loc,loc)

 inform(loc,loc)

**CLAUSES**

 hanoi(N):-

 move(N,left,middle,right).

 move(1,A,\_,C):-

 inform(A,C),

 !.

 move(N,A,B,C):-

 N1=N-1,

 move(N1,A,C,B),

 inform(A,C),

 move(N1,B,A,C).

 inform(Loc1, Loc2):-

 write("\nMove a disk from ", Loc1, " to ", Loc2).

**GOAL**

 hanoi(3).

2)Да се проверят стъпките на изпълнение при различен брой дискове

3)Да се направят необходимите промени за да може да се извежда и текущия номер на преместването.

**ЛАБОРАТОРНО УПРАЖНЕНИЕ № 7**

## Методи за обработка на естествен език

**І. Разделяне думите на срички**

Тук ще се приложи алгоритъм, който ще търси поредица от гласни и съгласни букви в дума и ще предложи къде ще може да се раздели думата на срички. При определяне на това къде ще се радели думата на срички се има в предвид двете основни последователности:

*1) гласна – съгласна – гласна*

В този случай думата ще се раздели след първата гласна. Един пример:

ruler > ru-ler

prolog > pro-log

2) гласна – съгласна – съгласна - гласна

 В този случай думата ще се раздели между двете съгласни. Например:

number > num-ber

panter > pan-ter

console > con-sole

Тези две правила работят добре за повечето думи, но за думи като **handbook** and **hungry** двата шаблона не дават резултат. За да се справи програмата и с тези думи, трябва да име речник с всички думи.

Програмата първо ще попита за дума, която трябва да се въведе и ще се опита да раздели думата на срички, спазвайки двете правила. Заради това е възможно с някои думи да не се получи коректен резултат.

DOMAINS

letter = symbol % дефинираме буква

word= letter\* % дефинираме дума, като списък от букви

Трябва да има предикат, с който да се определи дали буквата е гласна или съгласна.

В правилата, които програмата ще спазва трябва ясно да се определи, коя буква е звучна, т.е. освен гласните букви a, e, i, o, и u, трябва да се включи и звучната буква y, тъй като в много думи тя е гласната буква в думата. Например hyphen, pity, myrrh, syzygy, and martyr (Забележка: Направените заключения са свързани с особености в английския език). За да опишем гласните ще използваме предиката vocal. Неговите клаузи са:

vocal(a). vocal(e). vocal(i).

vocal(o). vocal(u). vocal(y).

Съгласна буква се определя като такава, която не е гласна.

consonant(L) :- not(vocal(L)).

Програмата има нужда от предикат **append,** с който ще се добавя елемент към списък:

append(word, word, word)

Следващия предикат, който е необходим е този, с който ще се преобразува думата, като списък от букви:

string\_word(string, word)

В този предикат ще се използват стандартните предикати **frontstr** (взема първия символ от низ), **free** (той успява, ако проверяваната променлива няма стойност) и **bound** (той успява, ако проверяваната стойност има стойност), за да може да определи коя клауза да активира.

След всичко това, вече е необходимо да се разреши главния проблем: да се дефинира предиката **divide**, който ще раздели думата на срички. Този предикат ще има четири параметъра и ще бъде рекурсивен. Първият и втория параметър съдържат съответно началото и оставащото от думата по време на рекурсията. Последните два аргумента връщат съответно първата и последната част на думата след като думата е разделена на срички.

Например първото правило за този предикат е:

divide(Start, [T1, T2, T3|Rest], D, [T2, T3|Rest]) :-

vocal(T1), consonant(T2), vocal(T3),

append(Start, [T1], D).

където Start е първата група от букви в думата за разделяне. Следващите три букви в думата са представени съответно с T1, T2, и T3, като Rest съдържа останалите букви в думата. В списъка D, буквите Т2 и Т3 и списъка Rest представят завършената част от думата. Думата се дели на срички в края на буквите, съдържащи се в D.

Това правило може да се опише със следния пример:

divide([p, r], [o, l, o, g], P1, P2)

divide([p, r], [o, l, o|[g]], [p, r, o], [l, o | [g]]) :-

vocal(o), consonant(l), vocal(o),

append([p, r], [o], [p, r, o]).

Предикатът append обединява първата гласна с началото на думата. P1 е [p, r, o], а P2 е [l, o, g].**DOMAINS**

 letter = char

 word\_ = letter\*

**PREDICATES**

 nondeterm divide(word\_,word\_,word\_,word\_)

 vocal(letter)

 consonant(letter)

 nondeterm string\_word(string,word\_)

 append(word\_,word\_,word\_)

 nondeterm repeat

 quit(string)

**CLAUSES**

 divide(Start,[T1,T2,T3|Rest],D1,[T2,T3|Rest]):-

 vocal(T1),

 consonant(T2),

 vocal(T3),

 append(Start,[T1],D1).

 divide(Start,[T1,T2,T3,T4|Rest],D1,[T3,T4|Rest]):-

 vocal(T1),

 consonant(T2),

 consonant(T3),

 vocal(T4),

 append(Start,[T1,T2],D1).

 divide(Start,[T1|Rest],D1,D2):-

 append(Start,[T1],S),

 divide(S,Rest,D1,D2).

 vocal('a').

 vocal('e').

 vocal('i').

 vocal('o').

 vocal('u').

 vocal('y').

 consonant(B):-

 not(vocal(B)),

 B <= 'z',

 'a' <= B.

 string\_word("",[]):-

 !.

 string\_word(Str,[H|T]):-

 bound(Str),

 frontchar(Str,H,S),

 string\_word(S,T).

 string\_word(Str,[H|T]):-

 free(Str),

 bound(H),

 string\_word(S,T),

 frontchar(Str,H,S).

 append([],L,L):-

 !.

 append([X|L1],L2,[X|L3]):-

 append(L1,L2,L3).

 repeat.

 repeat:-repeat.

 quit(""):-

 exit,

 !.

 quit(\_).

**GOAL**

 repeat,

 write("Write a multi-syllable word: "),

 readln(S),nl,

 quit(S),

 string\_word(S,Word),

 divide([],Word,Part1,Part2),

 string\_word(Syllable1,Part1),

 string\_word(Syllable2,Part2),

 write("Division: ",Syllable1,"-",Syllable2),nl,

 fail.

Задачи:

1)Направете необходимите промени за да може да се използва програмата за думи на български.

2)Направете необходимите промени за да може да се разделят на срички и думи, в които има поредица от три съгласни букви.

**ІІ.Семантичен анализатор**

Следващата програма е добър пример за анализатор на изреченията в английския език. С леки поправки програмата може да се пригоди и към други освен използваните типове изречения. Входа на семантичния анализатор разбира се е изречение, а изхода е списък от клаузи, които показват всяка част от изречението какъв граматически компонент е – глагол, съществително и т.н.

Програмата SEN\_AN.PRO демонстрира основата, при която програмистите могат да вложат английски синтактичен анализатор в Пролог. Когато стартира анализатора той чака потребителя да въведе изречение на английски език. Програмата се опитва да направи синтактичен анализ по начин, който потребителя може да разбере.

Резултантния обект данни е известен като синтактично дърво. След като синтактичния анализ премине успешно, програмата създава синтактично дърво през което може да се премине в предвид специфичната задача. Ако програмата не успее правилно да извърши анализа извежда съобщение, съответстващо на грешката. Ако при въвеждане на изречението се въведе дума, която не е част от речника, програмата извежда съобщение, че не е разпозната думата.

Например, ако се въведе: „a mother loves her children”, парсерът ще изведе следното:

sent(nounp(determ("a"), "mother", none),

verbp("loves", nounp(determ("her"), "children", none)))3

facts - senan\_db

 det(string)

 noun(string)

 rel(string)

 verb(string)

domains

 DETERM = none;

 determ(string)

 NOUNP = nounp(DETERM,string,RELCL)

 RELCL = none;

 relcl(string,VERBP)

 SENTENCE = sent(NOUNP,VERBP)

 VERBP =verb(string);

 verbp(string,NOUNP)

 TOKL = string\*

predicates

% Recognition of words in different forms

 is\_det(string)

 is\_noun(string)

 is\_rel(string)

 is\_verb(string)

% Parser

 nondeterm s\_determ(TOKL,TOKL,DETERM)

 nondeterm s\_nounp(TOKL,TOKL,NOUNP)

 nondeterm s\_relcl(TOKL,TOKL,RELCL)

 nondeterm s\_sentence(TOKL,TOKL,SENTENCE)

 nondeterm s\_verbp(TOKL,TOKL,VERBP)

% scanner

 check(string)

 tokl(string,TOKL)

clauses

 s\_sentence(TOKL,TOKL2,sent(NOUNP,VERBP)):-

 s\_nounp(TOKL,TOKL1,NOUNP),

 s\_verbp(TOKL1,TOKL2,VERBP),

 TOKL2=[],

 !.

 s\_sentence(\_,\_,\_):-

 write(">> Sentence not recognized\n"),

 fail.

 s\_nounp(TOKL,TOKL2,nounp(DETERM,NOUN,RELCL)):-

 s\_determ(TOKL,[NOUN|TOKL1],DETERM),

 is\_noun(NOUN),

 s\_relcl(TOKL1,TOKL2,RELCL).

 s\_determ([DETERM|TOKL],TOKL,determ(DETERM)):-

 is\_det(DETERM).

 s\_determ(TOKL,TOKL,none).

 s\_relcl([REL|TOKL],TOKL1,relcl(REL,VERBP)):-

 is\_rel(REL),

 s\_verbp(TOKL,TOKL1,VERBP).

 s\_relcl(TOKL,TOKL,none).

 s\_verbp([VERB|TOKL],TOKL1,verbp(VERB,NOUNP)):-

 is\_verb(VERB),

 s\_nounp(TOKL,TOKL1,NOUNP).

 s\_verbp([VERB|TOKL],TOKL,verb(VERB)):-

 is\_verb(VERB).

 is\_noun(X):-

 noun(X),

 !.

 is\_noun(X):-

 noun(Y),

 concat(Y,"s",X),

 !.

 is\_det(X):-

 det(X),

 !.

 is\_rel(X):-

 rel(X),

 !.

 is\_verb(X):-

 verb(X),

 !.

 is\_verb(X):-

 verb(Y),

 concat(Y,"s",X),

 !.

 is\_verb(X):-

 verb(Y),

 concat(Y,"ed",X),

 !.

 is\_verb(X):-

 verb(Y),

 concat(Y,"es",X),

 !.

 is\_verb(X):-

 verb(Y),

 concat(Y,"ing",X),

 !.

 tokl(STR,[TOK|TOKL]):-

 fronttoken(STR,TOK,STR1),

 check(TOK),

 !,

 tokl(STR1,TOKL).

 tokl(\_,[]).

 check(WORD):-

 is\_noun(WORD),

 !.

 check(WORD):-

 is\_det(WORD),

 !.

 check(WORD):-

 is\_rel(WORD),

 !.

 check(WORD):-

 is\_verb(WORD),

 !.

 check(WORD):-

 write(">> Unknown word: ",WORD),

 nl.

goal

 consult("sen\_an.dba",senan\_db),

 write("Try: every man that lives loves a woman\n"),

 write("Write a sentence: "),

 readln(STR),

 tokl(STR,TOKL),

 s\_sentence(TOKL,RESTTOKL,SENT),

 RESTTOKL=[],

 write(SENT).

**ЛАБОРАТОРНО УПРАЖНЕНИЕ № 8**

**Експертни системи.**

Секцията **FACTS** е съвкупност от факти, които могат да бъдат добавени или премахнати в програмата на Пролог по време на изпълнението на програмата. Декларираните предикати в тази секция се използват в програмата, както и останалите предикати декларирани в секцията **PREDICATES**.

В Пролог могат да се използват служебните предикатите ***assert, asserta, assertz*** за да се добавят нови факти и предикатите ***retract*** and ***retractall,*** за да се премахнат съществуващи факти. Може да се променя съдържанието на секцията **FACTS,**  като първо се премахне съществуващ и тогава да се добави нова версия на този факт.

**FACTS**

 xpositive(symbol,symbol)

 xnegative(symbol,symbol)

**PREDICATES**

 animal\_is(symbol)

 it\_is(symbol)

 ask(symbol,symbol,symbol)

 remember(symbol,symbol,symbol)

 positive(symbol,symbol)

 negative(symbol,symbol)

 clear\_facts

 run

**CLAUSES**

 animal\_is(cheetah):- % леопард

 it\_is(mammal), % бозайник

 it\_is(carnivore), % месоядно

 positive(has,tawny\_color), % жълтеникавокафяв цвят

 positive(has,dark\_spots). % тъмни петна

 animal\_is(tiger):- % тигър

 it\_is(mammal), % бозайник

 it\_is(carnivore), % месоядно

 positive(has, tawny\_color), % има жълтеникавокафяв цвят

 positive(has, black\_stripes). % има черни раета

 animal\_is(giraffe):- % жираф

 it\_is(ungulate), % копитно

 positive(has,long\_neck), % има дълъг врат

 positive(has,long\_legs), % има дълги крака

 positive(has, dark\_spots). % има тъмни петна

 animal\_is(zebra):- % зебра

 it\_is(ungulate), % копитно

 positive(has,black\_stripes). % има черни раета

 animal\_is(ostrich):- % щраус

 it\_is(bird), % птица

 negative(does,fly), % не може да лети

 positive(has,long\_neck), % има дълъг врат

 positive(has,long\_legs), % има дълги крака

 positive(has, black\_and\_white\_color). % има черно бял цвят

 animal\_is(penguin):- % пенгвин

 it\_is(bird), % птица

 negative(does,fly), % не може да лети

 positive(does,swim), % може да плува

 positive(has,black\_and\_white\_color). % има черно бял цвят

 animal\_is(albatross):- % албатрос

 it\_is(bird), % птица

positive(does,fly\_well). % може добре да лети

 it\_is(mammal):- % бозайник

 positive(has,hair). % има козина

 it\_is(mammal):-

 positive(does,give\_milk). % може да дава мляко

 it\_is(bird):- % птица

 positive(has,feathers). % има пера

 it\_is(bird):-

 positive(does,fly), % може да лети

 positive(does,lay\_eggs). % мъти яйца

 it\_is(carnivore):- % месоядно

 positive(does,eat\_meat). % храни се с месо

 it\_is(carnivore):-

 positive(has,pointed\_teeth), % има остри зъби

 positive(has, claws), % има нокти

 positive(has,forward\_eyes). % има остро зрение

 it\_is(ungulate):- % копитно животно

 it\_is(mammal), % бозайник

 positive(has,hooves). % има копита

 it\_is(ungulate):-

 it\_is(mammal), % бозайник

 positive(does,chew\_cud). % преживно животно

 positive(X,Y):-

 xpositive(X,Y),!.

 positive(X,Y):-

 not(xnegative(X,Y)),

 ask(X,Y,yes).

 negative(X,Y):-

 xnegative(X,Y),!.

 negative(X,Y):-

 not(xpositive(X,Y)),

 ask(X,Y,no).

 ask(X,Y,yes):-

 !,

 write(X," it ",Y,'\n'),

 readln(Reply),nl,

 frontchar(Reply,'y',\_),

 remember(X,Y,yes).

 ask(X,Y,no):-

 !,

 write(X," it ",Y,'\n'),

 readln(Reply),nl,

 frontchar(Reply,'n',\_),

 remember(X,Y,no).

 remember(X,Y,yes):-

 assertz(xpositive(X,Y)).

 remember(X,Y,no):-

 assertz(xnegative(X,Y)).

 clear\_facts:-

 write("\n\nPlease press the space bar to exit\n"),

 retractall(\_,dbasedom),

 readchar(\_).

 run:-

 animal\_is(X),!,

 write("\nYour animal may be a (an) ",X),

 nl,

 clear\_facts.

 run:-

 write("\nUnable to determine what"),

 write("your animal is.\n\n"),

 clear\_facts.

**GOAL**

 run.

Задачи:

1)Да се допълни програмата, така че да разпознава клас „насекоми”.

2)Да се допълни програмата, така че да разпознава един представител от клас „Насекоми”.

**ЛАБОРАТОРНО УПРАЖНЕНИЕ № 9**

## Програма - Зебра

Програмата решава класическа задача, при която се търси решение при ограничения. В 5 къщи живеят 5 човека от различни националности, с различно оцветяване на фасадите на къщите, различни предпочитания за марки цигари, питиета, различни домашни любимци и т.н. Разполага се с определен брой първоначални факти и възможност да се задават въпроси от вида: Кой живее в къщата в ляво от мъжа, който има за домашен любимец зебра?

В същността си решението на задачата е с използване на списъци. За да може да се реши този пъзел се използва кратък списък от предикати. Важно е да се отбележи, че ако се създаде много мощна програма в общия случай, то може след определена адаптация тя да реши по-конкретна задача.

Програмата на пролог решава задачата „с петте къщи” като използва метода на „проверка и избор”. Възможните решения за разпределяне на 5 цвята, 5 питиета, 5 националности, 5 вида цигари, 5 домашни любимци в 5 къщи е (5!)\*\*5.

Информацията, с която се разполага е:

Англичанинът живее в къща с червен цвят.

Испанецът има куче.

Норвежецът живее в първата къща от ляво.

Семейство Kool`s са пушачи и живеят в жълтата къща.

Мъжът, който пуши Chesterfields живее в съседната къща до мъжа с лисицата.

Норвежецът живее до къща в синьо.

Пушачът на Winston притежава охлюв.

Пушачът на lucky strike пие портокалов сок.

Украинецът пие чай.

Японеца пуши parliament.

Семейство Kool`s пушат в къща, съседна на тази, в която има кон.

В къщата оцветена в зелено се пие кафе.

Зелената къща е непосредствено в дясно от къщата с цвят слонова кост.

В средната къща пият мляко.

Въпросът към програмата е:

Къде живее зебрата и в коя от къщите се пие вода?

The question is then: Where does the zebra live, and in which house do they drink water ?

DOMAINS

 ID= symbol

 HOUSE = h(ID,NO) % Къщата се идентифицира с номер

 HLIST = reference HOUSE\* % Списък от къщи

 NO = integer % Номер на къщата

 NOLIST = NO\* % Списък от номера на къщи

PREDICATES

 nondeterm solve

 nondeterm candidate(HLIST,HLIST,HLIST,HLIST,HLIST)

 nondeterm perm(HLIST)

 nondeterm constraints(HLIST,HLIST,HLIST,HLIST,HLIST)

 nondeterm permutation(NOLIST,NOLIST)

 nondeterm delete(NO,NOLIST,NOLIST)

 member(HOUSE,HLIST)

 nondeterm next(NO,NO)

 nondeterm lleft(NO,NO)

CLAUSES

 solve():-

 constraints(Colours,Drinks,Nationalities,Cigarettes,Pets),

 candidate(Colours,Drinks,Nationalities,Cigarettes,Pets),

 member(h(water,WaterHouse),Drinks),

 member(h(WaterColour,WaterHouse),Colours),

 member(h(zebra,ZebraHouse),Pets),

 member(h(ZebraColour,ZebraHouse),Colours),

 write("They drink water in the ",WaterColour," house\n"),

 write("The zebra live in the ",ZebraColour," house\n").

 candidate(L1,L2,L3,L4,L5):-

 perm(L1),

 perm(L2),

 perm(L3),

 perm(L4),

 perm(L5).

 perm([h(\_,A),h(\_,B),h(\_,C),h(\_,D),h(\_,E)]):-

 permutation([A,B,C,D,E],[1,2,3,4,5]).

 constraints(Colours,Drinks,Nationalities,Cigarettes,Pets):-

 % Англичанинът живее в къща с червен цвят

 member(h(englishman,H1),Nationalities),

 member(h(red,H1),Colours),

 % Испанецът има куче

 member(h(spaniard,H2),Nationalities),

 member(h(dog,H2),Pets),

 % Норвежецът живее в първата къща от ляво

 member(h(norwegian,1),Nationalities),

 % Семейство Kool`s са пушачи и живеят в жълтата къща.

 member(h(kools,H3),Cigarettes),

 member(h(yellow,H3),Colours),

 % Мъжът, който пуши Chesterfields живее в съседната къща до мъжа с лисицата

 member(h(chesterfields,H4),Cigarettes),

 next(H4,H5),

 member(h(fox,H5),Pets),

 % Норвежецът живее до къща в синьо

 member(h(norwegian,H6),Nationalities),

 next(H6,H7),

 member(h(blue,H7),Colours),

 % Пушачът на Winston притежава охлюв.

 member(h(winston,H8),Cigarettes),

 member(h(snails,H8),Pets),

 % Пушачът на lucky strike пие портокалов сок

 member(h(lucky\_strike,H9),Cigarettes),

 member(h(orange\_juice,H9),Drinks),

 % Украинецът пие чай

 member(h(ukrainian,H10),Nationalities),

 member(h(tea,H10),Drinks),

 % Японеца пуши parliament

 member(h(japanese,H11),Nationalities),

 member(h(parliaments,H11),Cigarettes),

 % Семейство Kool`s пушат в къща, съседна на тази, в която има кон.

 member(h(kools,H12),Cigarettes),

 next(H12,H13),

 member(h(horse,H13),Pets),

 % В къщата оцветена в зелено се пие кафе

 member(h(coffee,H14),Drinks),

 member(h(green,H14),Colours),

 % Зелената къща е непосредствено в дясно от къщата с цвят слонова кост

 member(h(green,H15),Colours),

 lleft(H16,H15),

 member(h(ivory,H16),Colours),

 % В средната къща пият мляко.

 member(h(milk,3),Drinks).

 permutation([],[]).

 permutation([A|X],Y):-

 delete(A,Y,Y1),

 permutation(X,Y1).

 delete(A,[A|X],X).

 delete(A,[B|X],[B|Y]):-

 delete(A,X,Y).

 member(A,[A|\_]):-!.

 member(A,[\_|X]):-

 member(A,X).

 next(X,Y):-

 lleft(X,Y).

 next(X,Y):-

 lleft(Y,X).

 lleft(1,2).

 lleft(2,3).

 lleft(3,4).

 lleft(4,5).

GOAL

 solve.

Решението е:

They drink water in the yellow house

The zebra live in the green house

**ЛАБОРАТОРНО УПРАЖНЕНИЕ № 10**