**МІНІСТЕРСТВО ОСВІТИ І НАУКИ**

Дніпропетровський національний університет ім. О.Гончара

Факультет Прикладної математики

Кафедра комп’ютерних технологій

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Інструктивно-методичні матеріали до семінарських, практичних і лабораторних занять**

дисципліни

**“Інтелектуальні інформаційні системи”**

Напрям підготовки\_\_\_\_6.040302 Інформатика

(шифр і назва напряму підготовки)

Спеціальность **\_\_6.040302\_ Інформатика \_\_\_\_\_**

**Дніпропетровськ – 2014**

# 1. Теоретична частина

## 1.1. Методи розв’язання задач

Функціонування багатьох інтелектуальних систем (ІС) має цілеспрямований характер (прикладом можуть служити автономні інтелектуальні роботи). Типовим актом такого функціонування є розв’язання задачі планування шляху досягнення потрібної мети з деякої фіксованої початкової ситуації. Результатом розв’язання задачі повинний бути *план дій -* частково-упорядкована сукупність дій. Такий план нагадує сценарій, у якому в якості відносини між вершинами виступають відносини типу: "ціль-підціль" "мета-дія", "дія-результат" і т.п. Будь-який шлях у цьому сценарії, що веде від вершини, що відповідає поточній ситуації, у кожну з цільових вершин, визначає план дій.

Пошук плану дій виникає в ІС лише тоді, коли вона стикається з нестандартною ситуацією, для якої немає заздалегідь відомого набору дій, що приводять до потрібної мети. Усі задачі побудови плану дій можна розбити на два типи, яким відповідають різні моделі: *планування в просторі станів* (SS-проблема) і *планування в просторі задач* (PR-проблема).

У першому випадку вважається заданим деякий простір ситуацій. Опис ситуацій включає стан зовнішнього світу й стан ІС, яка характеризується низкою параметрів. Ситуації створюють деякі узагальнені стани, а дії ІС чи зміни в зовнішнім середовищі приводять до зміни актуалізованих у даний момент станів. Серед узагальнених станів виділені початкові стани (звичайно одне) і кінцеві (цільові) стани. SS-проблема складається в пошуку шляху, що веде з початкового стану в одне з кінцевих. Якщо, наприклад, ІС призначена для гри в шахи, то узагальненими станами будуть позиції, що складаються на шахівниці. Як початковий стан може розглядатися позиція, що зафіксована в даний момент гри, а як цільові позиції - множину нічийних позицій. Відзначимо, що у випадку шахів пряме перерахування цільових позицій неможливе. Матові і нічийні позиції, описані мовою, що відрізняється від мови опису станів, які характеризуються розташуванням фігур на полях дошки. Саме це утрудняє пошук плану дій у шаховій грі.

При плануванні в просторі задач ситуація трохи інша. Простір створюється в результаті введення на множині задач відносин типу: "частина - ціле", "задача - підзадача", "загальний випадок - окремий випадок" і т.п. Іншими словами, простір задач відбиває декомпозицію задач на підзадачі (цілі на підцілі). PR-проблема складається в пошуку декомпозиції вихідної задачі на підзадачі, що приводить до задач, розв’язання яких системі відомо. Наприклад, ІС відомо, як обчислюються значення sinx і cosx для будь-якого значення аргументу і як виробляється операція ділення. Якщо ІС необхідно обчислити **tg*****x****,* то розв’язанням PR-проблеми буде представлення цієї задачі у виді декомпозиції **tg*x=*sin*x/*cos*x*** (крім ***х=* */*2*+k***)*.*

### **Розв’язання задач методом пошуку в просторі станів**

Представлення задач у просторі станів припускає задання ряду описів: станів, множини операторів і їхніх впливів на переходи між станами, цільових станів. Описи станів можуть являти собою рядки символів, вектори, двовимірні масиви, дерева, списки і т.п. Оператори переводять один стан в інший. Іноді вони представляються у вигляді продукцій ***A*=>*B***, що означають, що стан ***А*** перетвориться в стан ***В****.*

Простір станів можна представити як граф, вершини якого позначені станами, а дуги - операторами.

Таким чином, проблема пошуку розв’язання задачі **<*А,У*>** при плануванні по станах представляється як проблема пошуку на графі шляху з ***А***в ***В****.* Звичайно графи не задаються, а генеруються в міру потреби.

Розрізняються *сліпі* і *спрямовані методи пошуку шляху.* Сліпий метод має два види: *пошук у глибину й пошук ушир.* При пошуку у глибину кожна альтернатива досліджується до кінця, без врахування інших альтернатив. Метод поганий для "високих" дерев, тому що легко можна прослизнути повз потрібну гілку і затратити багато зусиль на дослідження "порожніх" альтернатив. При пошуку вшир на фіксованому рівні досліджуються всі альтернативи і тільки після цього здійснюється перехід на наступний рівень. Метод може виявитися гірше методу пошуку у глибину, якщо в графі всі шляхи, що ведуть до цільової вершини, розташовані приблизно на одній і тій же глибині. Обидва сліпих методи вимагають великої витрати часу і тому необхідні спрямовані методи пошуку.

*Метод гілок і меж.* З незакінчених шляхів, що формуються в процесі пошуку, вибирається найкоротший і продовжується на один крок. Отримані нові незакінчені шляхи (їх стільки, скільки гілок у даній вершині) розглядаються поряд зі старими, і знову продовжується на один крок, найкоротший з них. Процес повторюється до першого досягнення цільової вершини, розв’язання запам'ятовується. Потім із незакінчених шляхів, що залишилися, виключаються більш довгі, ніж закінчений шлях, чи рівні йому, а що залишилися продовжуються по такому ж алгоритмі доти, поки їхня довжина менше закінченого шляху. У підсумку або всі незакінчені шляхи виключаються, або серед них формується закінчений шлях, більш короткий, чим раніше отриманий. Останній шлях починає відігравати роль еталона і т.д.

*Алгоритм найкоротших шляхів Мура.* Вихідна вершина ***X0*** позначається числом 0. Нехай у ході роботи алгоритму на поточному кроці отримана множина дочірніх вершин **X(*xi*)** вершини ***xi*.** Тоді з нього викреслюються усі раніше отримані вершини, що залишилися позначаються міткою, збільшеною на одиницю в порівнянні з міткою вершини ***xi***, і від них проводяться покажчики до ***Xi***.Далі на множині позначених вершин, що ще не фігурують як адреси покажчиків, вибирається вершина з найменшою міткою і для неї будуються дочірні вершини. Розмітка вершин повторюється доти, поки не буде отримана цільова вершина.

*Алгоритм Дейкстри* визначення шляхів з мінімальною вартістю є узагальненням алгоритму Мура за рахунок уведення дуг перемінної довжини.

*Алгоритм Дорана і Мічі пошуку з низькою вартістю.* Використовується, коли вартість пошуку велика в порівнянні з вартістю оптимального розв’язання. У цьому випадку замість вибору вершин, найменш віддалених від початку, як в алгоритмах Мура і Дейкстрі, вибирається вершина, для якої евристична оцінка відстані до мети найменша. При гарній оцінці можна швидко одержати розв’язок, але немає гарантії, що шлях буде мінімальним.

*Алгоритм Харта, Нільсона і Рафаеля.* В алгоритмі об'єднані обидва критерії: вартість шляху до вершини **g(*x*)** і вартість шляху від вершини **h(*x*)** *-* в аддитивній оціночній функції **f(*x*) *=*g(*x*)+h(*x*)***.* За умови **h(*x*)<hp(*x*)*,***де **hp(*x*)***-*дійсна відстань до мети, алгоритм гарантує перебування оптимального шляху.

Алгоритми пошуку шляху на графі розрізняються також напрямком пошуку. Існують прямі, зворотні і двонапрямлені методи пошуку. Прямий пошук йде від вихідного стану і, як правило, використовується тоді, коли цільовий стан заданий неявно. Зворотний пошук йде від цільового стану і використовується тоді, коли вихідний стан заданий неявно, а цільове явно. Двонапрямлений пошук вимагає задовільного розв’язання двох проблем: зміни напрямку пошуку й оптимізації "точки зустрічі". Одним із критеріїв для розв’язання першої проблеми є порівняння "ширини" пошуку в обох напрямках - вибирається той напрямок, що звужує пошук. Друга проблема викликана тим, що прямої і зворотний шляхи можуть розійтися і чим вужчий пошук, тим це більш імовірно.

### **Загальна схема алгоритму Харта, Нільсона і Рафаеля**

Узагальненням для багатьох алгоритмів на графах, розглянутих у цьому розділі, є алгоритм Харта, Нільсона і Рафаеля, призначений для пошуку найкоротшого шляху між двома вершинами графу. Розглянемо основні ідеї цього алгоритму . Введемо такі позначення:

***s*** — початкова вершина;

***q*** — цільова вершина;

**с** **(*i,j*)** — довжина ребра, яке з'єднує i-ту вершину з j-ю;

**d (*і, j*)** — довжина найкоротшого шляху між ***і***-ю та ***j***-ю вершинами; зокрема; **g (*n*)** — довжина найкоротшого шляху від початкової вершини до ***n***-ї; **h(*n*)** — довжина найкоротшого шляху від ***n***-ї вершини до цільової; **f(*n*)** — довжина найкоротшого шляху від початкової вершини до цільової серед усіх шляхів, які проходять через ***n***-ну вершину; при цьому **f(*n*) =g (*n*) + h(n);**

**g*’*(*n*)** — оцінка довжини найкоротшого шляху від початкової вершини до ***n***-ї; ця оцінка змінюється, якщо знайдено деякий новий (не обов'язково оптимальний) шлях до n-ї вершини; для оптимального шляху **g’(*n*) = g (*n*)**

**h’(*n*)** — евристична функція, яка задає оцінку довжини найкоротшого шляху від ***n***-ї вершини до цільової; у переважній більшості випадків ми не можемо знати точне значення **h(*n*),** але можемо оцінити його, виходячи зі специфіки конкретної задачі; приклади таких оцінок будуть наведені далі;

**f’(*n*) = g’(*n*)+h’(*n*)—** оцінка для **f(*n*);**

**L (*n*)** — множина всіх наступників вершини ***n***. Говоритимемо, що алго­ритм розкриває вершину ***n***, якщо він знаходить множину її наступників.

Далі нам будуть потрібні дві робочі множини: **OPEN** і **CLOSE**.

Загальну схему пошуку найкоротшого шляху на графі від початкової вершини до цільової можна описати так:

1. Сформувати граф пошуку ***G***, який на початку роботи алгоритму складається з однієї початкової вершини ***s***. Занести ***s*** до **OPEN**. Покласти **g*’*(*s*) = g (*s*) = 0**.
2. Сформувати множину **CLOSE**, яка на початку роботи алгоритму є порожньою.
3. Якщо множина **OPEN** порожня, вихід: невдача, потрібного шляху не існує.
4. Взяти з множини **OPEN** перший елемент **т** (відповідно до порядку, встановленому кроком 9). Вилучити **т** з **OPEN** та занести її до **CLOSE**.
5. Якщо **т** — цільова вершина, успіх. Відновити шлях від ***s*** до **т** на основі відновлюючих вказівників, що були встановлені на кроці 6—8, і завершити алгоритм.
6. Розкрити вершину **т**: отримати множину наступників **L (т).** Додати до графу ***G*** всі вершини, які належать **L (т),** але не належать ***G,*** разом з відповідними дугами. Додати ці вершини до **OPEN**. Для кожної вершини ***k* ∈ L (*m*) \ *G*** обчислити оцінки **f’(*k*) = g*’*(*k*) + h’(*k*),** поклавши **g’(*k*) = g’(*m*) + c(*m,k*);** встановити з ***k*** до **т** відновлюючий вказівник.
7. Для кожної вершини ***n*** з тих, які потрапили до **L (т),** але вже належали **OPEN** або **CLOSE**, переобчислити оцінки **g’(*n*) = min((g’(*m*) + c(*m,n*)),g’(*n*))**. Якщо для деякої вершини попередня оцінка, що була обчислена раніше, зменшилася, перевстановити з неї відновлюючий вказівник до **т** (в напрямку найкоротшого шляху).
8. Для всіх вершин з **L (т),** які до цього знаходилися в **CLOSE**: перевстановити відновлюючі вказівники для кожного з наступників цих вершин у напрямку найкоротшого шляху, а також переобчислити оцінки наступників.
9. Переупорядкувати множину **OPEN** за зростанням значень **f’**.
10. 10. Повернутися на крок 3.

Взагалі евристичну функцію **h’** в алгоритмі Харта, Нільсона і Рафаеля можна обирати довільно і, залежно від вибору цієї функції, надійність і якість алгоритму можуть суттєво змінюватися. Сформульовані три важливі теореми, які встановлюють, за яких умов алгоритм гарантовано матиме ті чи інші бажані властивості.

Алгоритм Харта, Нільсона і Рафаеля називається допустимим, якщо він гарантовано знаходить оптимальний шлях до цільової вершини.

***Теорема 1****. Алгоритм Харта, Нільсона і Рафаеля є допустимим,якщо виконуються такі умови:*

* *будь-яка вершина або взагалі не має наступників, або має скінченну кількість наступників;*
* *довжина будь-якої дуги перевищує деяку величину* **ε *>* 0***;*
* *для будь-якої вершини п виконується нерівність* **h’(*n*) ≤ h(*n****), тобто евристична функція є оцінкою знизу і не перевищує справжньої відстані до цільової вершини.*

Нехай два допустимих алгоритми Харта, Нільсона і Рафаеля А і В відріз­няються лише своїми евристичними функціями **h’A** та **h’B**. Кажуть, що алгоритм А є інформованішим, ніж алгоритм В, якщо для будь-якої нецільової вершини **h’A(*n*)> h’B (*n*).**

***Теорема 2*** *. Якщо алгоритм* ***А*** *є інформованішим, ніж алгоритм* ***В****, то при завершенні роботи алгоритму для будь-якого графу, для якого існує шлях до цільової вершини, будь-яка вершина, що розкривається алгоритмом* ***А****, розкривається і алгоритмом* ***В****.*

Інакше кажучи, теорема стверджує, що інформованіший алгоритм розкриває не більше вершин, ніж менш інформований, і тому є оптимальнішим за швидкодією.

Наступна важлива вимога, яка накладається на евристичну функцію, полягає в її консистентності (від англ. consistency). Евристична функція **h’(*n*)** називається консистентною, якщо для будь-яких двох вершин т i п таких, що т є безпосереднім наступником ***n***, виконується нерівність **h’(*n*)-h’(*m*)≤c(*n,m*).**

Це означає, що при переході від будь-якої вершини до її наступника консистентна евристична функція не може зменшитися на величину, що перевищує довжину з'єднуючого ребра.

***Теорема 3****. Якщо евристична функція, що використовується в алгоритмі Харта, Нільсона і Рафаеля, є консистентною, то алгоритм розкриває будь-яку вершину не раніше, ніж знайде оптимальний шлях до неї.*

Інакше кажучи, якщо виконується умова консистентності, то в момент, коли алгоритм розкриває вершину ***n***, для неї виконується рівність **g’(*n*)=g(*n*).**

Консистентність евристичної функції має дуже велике значення. Вона дозволяє не повертатися до тих вершин, які вже були проаналізовані і потапили до множини **CLOSE**.

Якщо евристична функція не використовується, тобто якщо **h’(*n*)≡ 0** і, відповідно, **f’(*n*)=g’(*n*),** маємо алгоритм Дейкстри. Теорема 2. стверджує, що введення належної евристичної функції дозволяє у більшості випадків прискорити роботу алгоритму.

Якщо взяти **g’(*n*)≡ 0**, утворюється алгоритм Дорана і Мічі. Цей алгоритм у ряді випадків дозволяє швидко отримати шлях до цільової вершини. Але гарантії того, що цей шлях буде оптимальним, немає. Більше того, не гарантується навіть знаходження хоча б якогось шляху до цільової вершини.

Проілюструємо методику застосування алгоритму Харта, Нільсона і Рафаеля до вирішення головоломки "гра у 8". Цю головоломку можна розглядати як спрощений варіант відомої головоломки С. Лойда "гра у 15". В коробці розміром 3x3 лежать 8 фішок, що пронумеровані цифрами від 1 до 8; одне місце є пустим (рис. 1).

Фішки можна довільно переміщувати, але не можна виймати з коробки, задача полягає в тому, щоб будь-яку початкову позицію (початкову розстановку фішок) перевести до цільової позиції, зображеної на рис 2. (якщо це можливо), при цьому потрібно зробити це за мінімальну кількість ходів.

|  |  |  |
| --- | --- | --- |
| **1** | **2** | **3** |
| **8** |  | **4** |
| **7** | **6** | **5** |

|  |  |  |
| --- | --- | --- |
| **1** | **5** | **7** |
| **6** | **8** |  |
| **2** | **4** | **3** |

**Рис. 2. Цільова позиція**

##### Рис. 1. Головоломка "гра у 8"

Не будь-яку початкову позицію можна з додержанням правил гри перевести до цільової. Можна показати, що це можливо тільки для тих позицій, які переходять до цільової за парну кількість перестановок. Перестановкою називається така операція: дві фішки виймаються з коробки та міняються місцями. Втім, якщо для даної початкової позиції задача не має розв'язку, це буде з'ясовано в ході роботи алгоритму.

Розв'язок задачі можна автоматизувати на основі планування в просторі станів; побудова графу станів є очевидною. Кожна вершина графу відповідає певній позиції; якщо існує хід, який дозволяє перейти від позиції А до позиції В, то з А до В йде орієнтована дуга.

Так, у позиції на рис. 3 можна зробити два ходи:

пересунути фішку "8" вправо;

пересунути фішку "З" вгору.

Тому відповідний фрагмент графу станів має вигляд, зображений на рис. 3:

|  |  |  |
| --- | --- | --- |
| **1** | **5** | **7** |
| **6** | **8** |  |
| **2** | **4** | **3** |

|  |  |  |
| --- | --- | --- |
| **1** | **5** | **7** |
| **6** |  | **8** |
| **2** | **4** | **3** |

|  |  |  |
| --- | --- | --- |
| **1** | **5** | **7** |
| **6** | **8** | **3** |
| **2** | **4** |  |

|  |  |  |
| --- | --- | --- |
| **1** | **5** | **7** |
| **6** | **8** | **3** |
| **2** | **4** |  |

**Рис. 3. Фрагмент графу станів**

Тепер, коли граф станів побудовано, для розв'язання задачі можна застосовувати будь-який алгоритм пошуку найкоротшого шляху. Для алгоритму Харта, Нільсона та Рафаеля необхідно задати конкретні евристичні функції, наприклад:

■ **h’1**, — кількість фішок, які стоять не на своїх місцях;

■ **h’2** — сума відстаней усіх фішок до свого місця, тобто сума ходів, які потрібно зробити, щоб перевести кожну фішку на своє місце, за умови, що інших фішок на дошці немає.

Легко побачити, що обидві евристичні функції є допустимими, тобто є оцінками знизу. Обидві функції також є консистентними. Легко також бачити, що алгоритм, який використовує функцію **h’2**, є інформованіший, ніж той, який використовує функцію **h’1**.

### **Розв’язання задач методом редукції**

Цей метод приводить до гарних результатів тому, що часто розв’язання задач має ієрархічну структуру. Однак не обов'язково вимагати, щоб основна задача і всі її підзадачі вирішувались однаковими методами. Редукція корисна для представлення глобальних аспектів задачі, а при рішенні більш специфічних задач кращий метод планування по станах. Метод планування по станах можна розглядати як окремий випадок методу планування за допомогою редукцій, тому що кожне застосування оператора в просторі станів означає зведення вихідної задачі до двох більш простих, з яких одна є елементарною. У загальному випадку редукція вихідної задачі не зводиться до формування таких двох підзадач, з яких хоча б одна була елементарною.

Пошук планування в просторі задач полягає в послідовному зведенні вихідної задачі до усе більш простої доти, поки не будуть отримані тільки елементарні задачі. Частково упорядкована сукупність таких задач складе розв’язання вихідної задачі. Розчленовування задачі на альтернативні множини підзадач зручно представляти у вигляді І/АБО-графа. У такому графі усяка вершина, крім кінцевої, має або кон’юнктивно зв'язані дочірні вершини (І-вершина), або диз’юнктивно зв'язані (АБО-вершина). В окремому випадку, при відсутності І-вершин, має місце граф простору станів. Кінцеві вершини є або заключними (їм відповідають елементарні задачі), або тупиковими. Початкова вершина (корінь І/АБО-графа) представляє вихідну задачу. Ціль пошуку на І/АБО-граф-показати, що початкова вершина розв'язувана. Розв'язуваними є заключні вершини (І-вершини), у яких розв'язувані всі дочірні вершини, і АБО-вершини, у яких розв'язувана хоча б одна дочірня вершина. Розв'язувальний граф складається з розв'язуваних вершин і вказує спосіб можливості розв'язання початкової вершини. Наявність тупикових вершин приводить до нерозв'язуваних вершин. Нерозв'язуваними є тупикові вершини, І-вершини, у яких нерозв'язувана хоча б одна дочірня вершина, і АБО-вершини, у яких нерозв'язувана кожна дочірня вершина.

*Алгоритм Ченга і Слейгла.* Заснований на перетворенні довільного І/АБО-графа в спеціальний АБО-граф, кожна АБО-галузь якого має І-вершини тільки наприкінці. Перетворення використовує представлення довільного І/АБО-графа як довільної формули логіки висловлювань з подальшим перетворенням цієї довільної формули в диз'юнктивну нормальну форму. Подібне перетворення дозволяє далі використовувати алгоритм Харта, Нільсона і Рафаеля.

*Метод ключових операторів.* Нехай задана задача **<*A*, *B>*** і відомо, що оператор **f** обов'язково повинний входити в розв’язання цієї задачі. Такий оператор називається ключовим. Нехай для застосування f необхідний стан ***C****,* а результат його застосування є **I(*c*).** Тоді І-вершина **<A,У>** породжує три дочірні вершини: **<*A, C*>, <*C,* f(*c*)>** і **<f(*c*), *B*>,** з яких середня є елементарною задачею. До задач **<*A, C*>** і **<f(*c*), *B*>** також підбираються ключові оператори, і зазначена процедура редукування повторюється доти, поки це можливо. У підсумку вихідна задача **<*A*, *B****>* розбивається на упорядковану сукупність підзадач, кожна з який розв’язується методом планування в просторі станів.

Можливі альтернативи на вибір ключових операторів, так що в загальному випадку буде мати місце І/АБО-граф. У більшості задач вдається не виділити ключовий оператор, а тільки вказати множину, яка його містить. У цьому випадку для задачі **<*A*, *B>*** обчислюється розходження між ***A*** і ***B***, якому ставиться у відповідність оператор, що усуває це розходження. Останній і є ключовим.

*Метод планування загального вирішувача задач (ЗВЗ).* ЗВЗ з'явився першою найбільш відомою моделлю планувальника. Він використовувався для розв’язання задач інтегрального числення, логічного висновку, граматичного розбору й ін. ЗВЗ поєднує два основних принципи пошуку:

аналіз цілей і засобів та рекурсивне розв’язання задач. У кожнім циклі пошуку ЗВЗ вирішує у твердій послідовності три типи стандартних задач: перетворити об'єкт ***А*** в об'єкт ***У****,* зменшити розходження ***D*** між ***А*** и ***В****,* застосувати оператор ***f*** до об'єкта ***А****.* Розв’язання першої задачі визначає розходження ***D***другої - придатний оператор ***f***, третьої - необхідну умову застосування ***С****.* Якщо ***С*** не відрізняється від ***A***, то оператор ***f*** застосовується, інакше ***С*** представляється як чергова мета і цикл повторюється, починаючи з задачі "перетворити ***A*** у ***С***". У цілому стратегія ЗВЗ здійснює зворотний пошук від заданої мети ***В***донеобхідного засобу її досягнення ***С***, використовуючи редукцію вихідної задачі **<*A*, *У>*** *до* задач <***A, C***> і *<****С, У****>.*

Помітимо, що в ЗВЗ мовчазно передбачається незалежність розходжень одне від одного, відкіля випливає гарантія, що зменшення одних розходжень не приведе до збільшення інших.

*Планування за допомогою логічного висновку*. Таке планування припускає: опис станів у виді правильно побудованих формул (ППФ) деякого логічного числення, опис операторів у виді або ППФ, або правил перекладу одних ППФ в інші. Представлення операторів у виді ППФ дозволяє створювати дедуктивні методи планування, представлення операторів у виді правил перекладу - методи планування з елементами дедуктивного висновку.

*Дедуктивний метод планування системи QA3*, ЗВЗ не виправдав надій, що покладалися на його, в основному через незадовільне представлення задач. Спроба виправити положення привела до створення запитально-відповідальної системи QA3. Система розрахована на довільну предметну область і здатна шляхом логічного висновку відповісти на запитання: чи можливо досягнення стану В з A? Як метод автоматичного висновку використовується принцип резолюцій. Для напрямку логічного висновку QA3 застосовує різні стратегії, в основному синтаксичного характеру, що враховують особливості формалізму принципу резолюцій. Експлуатація QA3 показала, що висновок у такій системі виходить повільним, детальним, що невластиво міркуванням людини.

*Метод продукцій системи* **STRIPS**. У цьому методі оператор представляє продукцію ***Р, А=>У****,* де ***Р, А*** і ***В*** - множини ППФ числення предикатів першого порядку, *Р* виражає умови застосування ядра продукції ***А=>У****,* де ***В*** містить список ППФ,що додаються, і список ППФ,що виключаються , тобто постумови. Метод повторює метод ЗВЗ із тією відмінністю, що стандартні задачі визначення розходжень і застосування придатних операторів вирішуються на основі принципу резолюцій. Придатний оператор вибирається так само, як в ЗВЗ, на основі принципу "аналіз засобів і цілей". Наявність комбінованого методу планування дозволило обмежити процес логічного висновку описом стану світу, а процес породження нових таких описів залишити за евристикою "від мети до засобу її досягнення".

*Метод продукцій, що використовують макрооператори* [Файкс і ін., 1973]. Макрооператори -це узагальнені розв’язання задач, одержувані методом **STRIPS**. Застосування макрооператорів дозволяє скоротити пошук розв’язання, однак при цьому виникає проблема спрощення застосовуваного макрооператора, суть якої полягає у виділенні по заданому розходженню його необхідної частини і виключенні з останньої непотрібних операторів.

### **Розв’язання задач дедуктивного вибору**

У дедуктивних моделях представлення й обробки знання розв'язувана проблема записується у вигляді тверджень формальної системи, у вигляді твердження, справедливість якого варто установити чи спростувати на підставі аксіом (загальних законів) і правил висновку формальної системи. Як формальну систему використовують числення предикатів першого порядку.

Відповідно до правил, встановлених у формальній системі, заключному твердженні-теореми, отриманої з початкової системи тверджень (аксіом, посилок), приписується значення ІСТИНА, якщо кожній посилці, аксіомі також приписане значення ІСТИНА.

Процедура висновку являє собою процедуру , що із заданої групи виразів виводить відмінний від заданих вираз.

Звичайно в логіці предикатів використовується формальний метод доказу теорем, що допускає можливість його машинної реалізації, але існує також можливість доказу неаксіоматичним шляхом : прямим висновком, зворотним висновком.

Метод резолюції використовується в якості повноцінного (формального) методу доказу теорем.

Для застосування цього методу вихідну групу заданих логічних формул потрібно перетворити в деяку нормальну форму. Це перетворення проводиться в кілька стадій, що складають машину висновку.

### **Розв’язання задач, що використовують немонотонні логіки, імовірнісні логіки**

Дані і знання, з якими доводиться мати справу в ІС, рідко бувають абсолютно точними і достовірними. Властива знанням невизначеність може мати різноманітний характер, і для її опису використовується широкий спектр формалізмів. Розглянемо один з типів невизначеності в даних і знаннях - їхня неточність. Будемо називати висловлення *неточним,* якщо його істинність (чи хибність) не може бути встановлена з певністю. Основним поняттям при побудові моделей *неточного висновку* є поняття імовірності, тому всі описувані далі методи зв'язані з імовірнісною концепцією.

Модель оперування з неточними даними і знаннями включає дві складові: мови представлення неточності і механізм висновку на неточних знаннях. Для побудови мови необхідно вибрати форму представлення неточності (наприклад, скаляр, інтервал, розподіл, лінгвістичне вираження, множина) і передбачити можливість приписування міри неточності усім висловленням.

Механізми оперування з неточними висловленнями можна розділити на два типи. До першого відносяться механізми, що носять "приєднаний" характер: перерахування мір неточності неначе супроводжує процес висновку, що ведеться на точних висловленнях. Для розробки приєднаної моделі неточного висновку в заснованій на правилах висновку системі необхідно задати функції перерахування, що дозволяють обчислювати:

а) міру неточності антецедента правила (його лівої частини) по мірах неточності складових його висловлень;

*б)* міру неточності консеквента правила (його правої частини) по мірах неточності правила і посилки правила*;*

в) об'єднану міру неточності висловлення по мірах, отриманим із правил.

Уведення міри неточності дозволить додати в процес висновку щось принципово нове - можливість об'єднання сили декількох свідчень, що підтверджують чи спростовують ту саму гіпотезу. Іншими словами, при використанні мір неточності доцільно виводити те саме твердження різними шляхами (з наступним об'єднанням значень неточності), що зовсім безглуздо в традиційній дедуктивній логіці. Для *об'єднання свідчень* потрібна функція перерахування*, що* займає центральне місце в перерахуванні. Зазначимо, що, незважаючи на "приєднаність" механізмів висновку цього типу, їхня реалізація в базах знань впливає на загальну стратегію висновку: з одного боку, необхідно виводити гіпотезу всіма можливими шляхами для того, щоб врахувати всі релевантні цій гіпотезі свідчення, з іншого - запобігти багаторазовий вплив сили тих самих свідчень.

Для механізмів оперування з неточними висловленнями другого типу характерна наявність схем висновку, спеціально орієнтованих на використовувану мову представлення неточності. Як правило, кожному кроку висновку відповідає перерахування мір неточності, обумовлений співвідношенням на множині висловлень (співвідношенням може бути елементарний логічний зв'язок*,* безвідносно до того, чи є це відношення фрагментом якого-небудь правила). Таким чином, механізми другого типу застосовуються не тільки до знань, виражених у формі правил. Разом з тим для них, як і для механізмів "приєднаного" типу, однією з головних є проблема об'єднання свідчень.

## 1.2. Експертні системи

На початку вісімдесятих років у дослідженнях зі штучного інтелекту сформувався самостійний напрямок, що одержав назву "експертні системи" (ЕС). Ціль досліджень з ЕС полягає в розробці програм, які при розв’язанні задач одержують результати, що не поступаються за якістю й ефективністю рішенням, які одержуються експертом. У ЕС часто використовується термін "інженерія знань", введений Е.Фейгенбаумом як "привнесення принципів і інструментарію досліджень з області штучного інтелекту в рішення важких прикладних проблем, що вимагають знань експертів".

Програмні засоби (ПЗ), що базуються на технології експертних систем, або інженерії знань одержали значне поширення у світі.

ЕС призначені для так званих неформалізованих задач, тобто ЕС не відкидають і не заміняють традиційного підходу до розробки програм, орієнтованого на рішення формалізованих задач.

Неформалізовані задачі звичайно мають наступні особливості:

- помилковістю, неоднозначністю, неповнотою і суперечливістю початкових даних;

- помилковістю, неоднозначністю, неповнотою і суперечливістю знань про проблемну область і розв'язувану задачу;

- великою розмірністю простору рішення, тобто перебір при пошуку рішення дуже великий;

- даними і знаннями, що динамічно змінюються .

Неформалізовані задачі представляють великий і дуже важливий клас задач. Багато фахівців вважають, що ці задачі є найбільш масовим класом задач, розв'язуваних ЕОМ.

Експертні системи і системи штучного інтелекту відрізняються від систем обробки даних тим, що в них в основному використовуються символьний (а не числовий) спосіб представлення, символьний висновок і евристичний пошук рішення (а не виконання відомого алгоритму).

Основні переваги, що дає використання ЕС:

1. Сталість. Людська компетенція з часом слабшає. Перерва в діяльності людини-експерта може серйозно відбитися на його професійних якостях.
2. Легкість передачі чи відтворення. Передача знань від однієї людини іншій — довгий і дорогий процес. Передача штучної інформації — це простий процес копіювання програми чи файлу даних.
3. Стійкість і відтворюваність результатів. Експерт-людина може приймати в тотожних ситуаціях різні рішення через емоційні фактори. Результати ЕС — стабільні.
4. Вартість. Експерти, особливо висококваліфіковані обходяться дуже дорого. ЕС, навпаки, порівняно недорогі. Їхня розробка дорога, але вони дешеві в експлуатації.

Разом з тим розробка ЕС не дозволяє цілком відмовитися від експерта-людини. Хоча ЕС добре справляється зі своєю роботою, проте, у визначених областях людська компетенція явно перевершує штучну. Однак і в цих випадках ЕС може дозволити відмовитися від послуг висококваліфікованого експерта, залишивши експерта середньої кваліфікації, використовуючи при цьому ЕС для посилення і розширення його професійних можливостей. На рис. 4 наведена структура ЕС.

Щоб проводити експертизу, комп'ютерна програма повинна бути здатна вирішувати задачі за допомогою логічного висновку й одержувати при цьому досить надійні результати. Програма повинна мати доступ до системи фактів, що називається базою знань.

Програма також повинна під час консультації виводити висновок з інформації, що знаходиться в базі знань. Деякі експертні системи можуть також використовувати нову інформацію, що додається під час консультації. ЕС, таким чином, складається з трьох частин:

1. База знань (БЗ).

2. Механізм висновку (МВ).

3. Система інтерфейсу користувача (СІК).

**розроблювач інструментарію**

**предметний експерт**

**експертна система**

**кінцевий користувач**

**інженер знань**

**засоби**

**побудови ЕС**

**клерки**

**будує**

**опитує**

використовує, розширює, перевіряє

**додає інформацію**

**розробляє, уточнює, тестує**

**Рис.4. Структура експертних систем**

Щоб проводити експертизу, комп'ютерна програма повинна бути здатна вирішувати задачі за допомогою логічного висновку й одержувати при цьому досить надійні результати. Програма повинна мати доступ до системи фактів, що називається базою знань.

Програма також повинна під час консультації виводити висновок з інформації, що знаходиться в базі знань. Деякі експертні системи можуть також використовувати нову інформацію, що додається під час консультації. ЕС, таким чином, складається з трьох частин:

1. База знань (БЗ).

2. Механізм висновку (МВ).

3. Система інтерфейсу користувача (СІК).

База знань - центральна частина експертної системи. Вона містить правила, що описують відносини або явища, методи і знання для розв’язку задач з області застосування системи. Можна представляти базу знань, що складається з фактичних знань і знань, що використовуються для висновку інших знань. Факти і правила в ЕС не завжди або вірні, або помилкові. Іноді існує деякий ступінь непевності у вірогідності факту або точності правила. Якщо цей сумнів виражений явно, то він називається "коефіцієнтом довіри".

Коефіцієнт довіри — це число, що означає імовірність чи ступінь упевненості, з яким можна вважати даний факт або правило достовірним чи справедливим.

Багато правил ЕС є евристиками, тобто емпіричними правилами або спрощеннями, що ефективно обмежують пошук рішення. ЕС використовують евристики, тому що задачі, які вона вирішує, важкі, не до кінця зрозумілі, не піддаються строгому математичному аналізу чи алгоритмічному розв’язуванню. Алгоритмічний метод гарантує коректне або оптимальне розв’язування задачі, тоді як евристичний метод дає прийнятне рішення в більшості випадків.

Знання в ЕС організовані так, щоб знання про предметну область відокремити від інших типів знань системи, таких як загальні знання про те, як вирішувати задачі чи знання про те, як взаємодіяти з користувачем. Виділені знання про предметну область називаються базою знань, тоді як загальні знання про розв’язання задач називаються механізмом висновку. Програмні засоби, що працюють зі знаннями, організованими таким чином, називаються системами, заснованими на знаннях. База знань містить факти (дані) і правила (чи інші представлення знань), що використовують ці факти як основу для прийняття рішень.

Існують різні види представлення даних. Наприклад,

* пари атрибут-значення (властивість-значення);
* триплети об’єкт-атрибут-значення
* записи;
* фрейми;
* семантичні мережі.

Найбільш фундаментальною є схема, яка використовує пари атрибут-значення. Вона ілюструється у системі ідентифікації птахів (див додаток). Прикладами є пари **color-white**, **size-large**.

Коли система має справу із складними об’єктами, необхідно в пару атрибут-значення включити об’єкт. Наприклад, система установки меблів в кімнаті могла б мати справу із складними стільцями, які мають різні атрибути, такі як розмір. У даному випадку представлення даних повинне включати об’єкт.

Раз уже об’єкти включені в систему, кожен із них міг би мати складні атрибути. Це веде до структури, яка базується на записах, де кожен пункт даних складається із імені об’єкта і всіх пов’язаних із ним пар атрибут-значення.

Фрейми – ще більш складний спосіб зберігання об’єктів і їх атрибутів та значень. Фрейми додають інтелектуальність до представлення даних і дозволяють об’єктам наслідувати значення від інших об’єктів. Більш того, кожен атрибут може бути пов’язаний із певною процедурою (яка називається демоном), що виконується при запиті або відновлюванні атрибута.

Механізм висновку містить принципи і правила роботи. Механізм висновку "знає", як використовувати базу знань так, щоб можна було одержувати розумно погоджені висновки з інформації, що знаходиться в ній.

Коли експертній системі задається питання, механізм висновку вибирає спосіб застосування правил бази знань для рішення задачі, поставленої в питанні. Фактично, механізм висновку запускає експертну систему в роботу, визначаючи які правила потрібно викликати й організувати до цих правил доступ у базу знань. Механізм висновку виконує правила, визначає, коли знайдене прийнятне рішення і передає результати програмі інтерфейсу з користувачем. Коли питання треба попередньо обробити, то доступ до бази знань здійснюється через інтерфейс із користувачем.

Механізм висновку містить:

\* інтерпретатор, що визначає як застосовувати правила для висновку нових знань на основі інформації, що зберігається в базі знань;

\* диспетчер, що встановлює порядок застосування цих правил.

Такі ЕС одержали назву статичних ЕС і мають структуру, зображену на рис. 5. Ці ЕС використовуються в тих додатках, де можна не враховувати зміни навколишнього світу за час розв’язку задачі.

**діалоговий компонент**

**пояснювальний компонент**

**компонент набуття знань**

**механізм висновку**

**інтерпретатор**

диспет

чер

**робоча пам’ять**

**база знань**

**інтерфейс з БД**

**Рис.5. Структура статичної ЕС**

Однак існує більш високий клас додатків, де потрібно враховувати динаміку зміни навколишнього світу за час виконання додатка. Такі експертні системи одержали назва динамічних ЕС і їхня узагальнена структура має вид, даний на рис.6.

У порівнянні зі статичною ЕС у динамічну вводиться ще два компоненти:

\* підсистема моделювання зовнішнього світу;

\* підсистема сполучення з зовнішнім світом.

Динамічні ЕС здійснює зв'язок із зовнішнім світом через систему контролерів і датчиків. Крім того компоненти бази знань і механізму висновку істотно змінюються, щоб відбити тимчасову логіку подій, що відбуваються в реальному світі.

**технічні пристрої**

**датчики**

локальна система

**підсистема сполучення з зовнішнім світом**

**елементи статичної ЕС**

**підсистема моделювання зовнішнього світу**

**динамічна ЕС**

**Рис. 6. Динамічна ЕС**

До розряду таких динамічних середовищ розробки ЕС відноситься сімейство програмних продуктів фірми **Gensym Corp**. (США). Один з таких продуктів система **G2** – базовий програмний продукт, що представляє собою графічне, об'єктно-орієнтоване середовище для побудови і супроводу експертних систем реального часу, призначених для моніторингу, діагностики, оптимізації, планування і керування динамічним процесом.

Існують два порядка механізма висновку – прямий (**forward chaining** ) і зворотній (**backward chaining**). Інша їх назва – міркування, що керуються данними (**Data driven reasoning** ) і міркування, що керуються ціллю (**Goal driven reasoning** ) Прямий порядок висновку будується від активних фактів до висновку, тобто за відомими фактами знаходиться висновок, який випливає із цих фактів. Цей механізм аналогічний тому, яким користується слідчий, який на основі певної кількості фактів “обчислює” злочинця. При зворотньому порядку висновкуоду низка готових висновків послідовно розглядається до тих пір, поки не будуть знайдені факти конкретної ситуації, які підтверджують який-небуть один висновок, тобто шляхом підбору фактів, які підходять під висновок. Цей механізм аналогічний процедурі, коли у слідчого є певна кількість підозрюваних і він по черзі перевіряє кожного з них, знаходячи злочинця.

Зворотний висновок ефективний у випадках так званих проблем “структурного вибору”. Ціль системи у цьому разі – зробити найкращий вибір із деякої кількості можливостей. Наприклад, під цю категорію підпадає задача ідентифікації. Цю модель використовують також системи діагностики, так як ціллю системи є поставити найточніший діагноз.

Для багатьох проблем неможливо перерахувати всі можливі відповіді даної задачі. Під цю категорію підпадають проблеми конфігурації. Наприклад, це може бути система вибору компонентів комп’ютера, система проектування монтажних схем, система, яка оптимально розміщує меблі в кімнаті і т. д. Так як тут вхідні дані дуже різноманітні і можуть комбінуватися у майже бескінечне число способів, зворотній висновок не спрацьовує. У даному випадку застосовується прямий висновок.

Системи, які використовують прямий висновок часто називаються продукційними системами (див. розділ “Експертна система, заснована на правилах”).

Система інтерфейсу з користувачем приймає інформацію від користувача і передає йому інформацію. Просто говорячи, система інтерфейсу повинна переконатися, що, після того як користувач описав задачу, уся необхідна інформація отримана. Інтерфейс, ґрунтуючись на виді і природі інформації, уведеної користувачем, передає необхідну інформацію механізму висновку. Коли механізм висновку повертає знання, виведені з бази знань, інтерфейс передає їх назад користувачу в зручній формі. Інтерфейс із користувачем і механізм висновку можуть розглядатися як "додаток" до бази знань. Вони разом складають оболонку експертної системи. Для бази знань, що містить велику і різноманітну інформацію, можуть бути розроблені і реалізовані кілька різних оболонок.

Добре розроблені оболонки експертних систем звичайно містять механізм для додавання і відновлення інформації в базі знань.

В інтерфейсі з користувачем мовою спілкування є обмежена природня мова, а не формальна мова програмування. В ньому можуть використовуватися всі відомі форми діалогу:

* Директивна. Словник даної форми складається із ключових слів на природній мові, скорочень, чисел, мнемокодів.
* Таблична. Складається із таких типів: - вибір операції для виконання з меню; - заповнення і редагування шаблону.
* Фразова. Використовує обмежену природню мову.

Фразова форма використовується в багатьох існуючих ЕС, але на досить примітивному рівні. Допустимі вхідні повідомлення користувача обмежені набором понять, які містяться в базі знань.

Перспективним напрямком є розробка ЕС, орієнтоваеих на користувача не фахівця, які можуть спілкуватися із системою за допомогою повних речень, що містять будь-які частини мови.

Для зменшення часу набору фраз можуть застосовуватися скорочення, шаблони фраз, запрограмовані клавіші ключових слів і меню.

Для обробки фраз існують такі види аналізу:

* Морфологічний (лексичний) аналіз – обробка словоформ (відрізок тексту між двома сусідніми пробілами) без зв’язку з контекстом. Виділяють два метода:
* декларатвний – в словнику знаходяться всі можливі словоформи кожного слова і аналіз полягає у пошуку словоформи в словнику. При використанні даного методу забезпечують можливість обробки повідомлень, які складаються із малих і великих літер у довільній комбінації.
* процедурний – у поточній словоформі виділяється основа, яка потім ідентифікується.
* Синтаксичний аналіз – використовуючи інформацію, яка отримана після морфологічного аналізу будується синтаксична структура вхідного повідомлення, тобто здійснюється розбір речення.
* Семантичний аналіз – визначення смислових відношень між словоформами (знаходяться головні предикати). Стосовно даної форми можна виділити наступні функції інтерфейсу:
* здійснювати перетворення повідомлення із природньо-мовної форми в форму внутрішнього виду і зворотнє перетворення;
* аналіз і синтез повідомлень користувача і системи;
* відслідковувати і запам’ятовувати пройдений шлях.

### **Експертна система, заснована на правилах (прямий висновок - forward chaining)**

В усіх експертних системах існує залежність між вхідним потоком даних і даними в базі знань. Під час консультації вхідні дані зіставляються з даними в базі знань. Результатом зіставлення є негативна чи позитивна відповідь. У системі, що базується на правилах, стверджувальний результат є дією одного з продукційних правил. Ці продукційні правила визначаються вхідними даними. Експертна система також містить інтерпретатор у механізмі висновку, що вибирає й активізує різні модулі системи.

Роботу цього інтерпретатора можна описати послідовністю трьох кроків:

1. Інтерпретатор зіставляє зразок правила з елементами даних у базі знань.

2. Якщо можна викликати більш одного правила, то інтерпретатор використовує механізм вирішення конфлікту для вибору правила.

3. Інтерпретатор застосовує обране правило, щоб знайти відповідь на питання.

Цей процес інтерпретації є циклічним і називається циклом "розпізнавання-дія".

У системі, що базується на правилах, кількість продукційних правил визначає розмір бази знань. Деякі найбільш складні системи мають бази знань з більш ніж 5000 продукційних правил. При розробці таких ЕС необхідно:

1. Використовувати мінімально достатня множина умов при визначенні продукційного правила.

2. Уникати суперечних продукційних правил.

3. Конструювати правила, спираючи на структуру властивої предметної області.

### **Експертні системи, що базуються на логіці (зворотній висновок - backward chaining)**

В експертних системах, що базуються на логіці, база знань складається з тверджень у виді речень логіки предикатів.

Так само як і в системі, що базується на правилах, експертна система, що базується на логіці, має множину правил, що можуть викликатися за допомогою даних із вхідного потоку. Система має інтерпретатор, що може вибирати й активізувати модулі, що включаються в роботу системи.

Інтерпретатор виконує різні функції усередині системи на основі наступної схеми:

1. Система має речення в базі знань, що керують пошуком і зіставленням. Інтерпретатор зіставляє ці речення з елементами даних у базі даних.

2. Якщо може бути викликане більш одного правила, то система використовує можливості для вирішення конфлікту. Отже, користувачу не потрібно розглядати потенційно можливі конфлікти.

3. Система одержує результати уніфікаційного процесу автоматично, тому він може направлятися на потрібний пристрій виведення інформації.

Так само як і в системі, що базується на правилах, даний циклічний процес є процесом розпізнавання-дія.

Найбільш важливим аспектом для бази знань у системі, заснованій на логіці, є проектування бази знань, її тверджень і їхньої структури. База знань повинна мати недвозначну логічну організацію, і вона повинна містити мінімум надлишкової інформації. Так само як і в системі, що базується на правилах, мінімально достатня кількість даних утворять найбільш ефективну систему.

У роботі ЕС можна виділити два основних режими: режим набуття (накопичення) знань і режим розв’язку задачі (режим консультації або режим використання). У режимі *набуття знань* спілкування з ЕС здійснює експерт (за допомогою інженера знань).

Використовуючи компонент *набуття* знань, експерт описує проблемну область у виді сукупності фактів і правил. При традиційному підході програмування цьому процесу відповідають етапи: алгоритмізації, програмування і налагодження, виконувані *програмістом*. У випадку ЕС розробку програм здійснює не програміст, а *експерт*, що не володіє програмуванням.

У режимі *консультацій* спілкування з ЕС здійснює кінцевий користувач, якого цікавить результат і (чи) спосіб його одержання. У залежності від призначення ЕС користувач може:

не бути фахівцем у даній предметній області, і в цьому випадку він звертається до ЕС за результатом, що не вміє одержати сам;

бути фахівцем, і в цьому випадку він звертається до ЕС з метою прискорення одержання результату, покладаючи на ЕС рутинну роботу.

На відміну від традиційних програм ЕС при рішенні задачі не тільки виконує запропоновану алгоритмом послідовність операцій, але і сама попередньо формує її.

Добре побудована ЕС має можливість самонавчатися на розв'язуваних задачах, поповнюючи автоматично свою БЗ результатами отриманих висновків і рішень.

### **Модуль (компонент) пояснення**

Як показано на структурі ЕС ( мал. 5) експертна система повиння мати модуль пояснення. Без механізму пояснень користувач не довірятиме отриманим результатам і ЕС не матиме попиту. Призначення модуля пояснень – зробити ЕС “прозорою” для користувача, тобто надати можливість користувачу розуміти логіку дій ЕС, дати надійну гарантію вірності щодо отриманих результатів.

В деяких випадках важливість пояснення для користувача дещо переоцінюють, тому що воно не завжди корисне для нього. Це має місце через природу “інтелекту” ЕС.

Правила найчастіше відображають емпіричні, або “компільовані” знання. Вони є зведенням наближених правил експерта, а не його глибинних знань, що ведуть до цих наближених правил. Наприклад, розглянемо наступний діалог з ЕС, розробленої для надання порад при поломці автомобіля:

* Автомобіль заводиться? – **ні.**
* Двигун прокручується? – **так.**
* Пахне бензином? – **так.**
* Порада – Почекайте 5 хвилин і попробуйте завести знову.
* **Чому?.**
* Я використала правило: якщо автомобіль не заводиться і двигун прокручується і пахне бензином тоді порада така “Почекайте 5 хвилин і попробуйте завести знову.”

Це правило є вірним для автомобіля, в карбюратор якого попала вода, але в ньому немає ніяких знань про те, що таке карбюратор і чому допоможе п’ятихвилинна пауза. Якби користувач дійсно захотів зрозуміти що сталося, йому була би потрібна невелика дисертація на тему карбюратор, як він працює, і його зв’язок з педаллю газу.

Щоб зробити корисними пояснення для таких систем, необхідно не просто показувати якими правилами користувалася система. Один підхід полягає в тому, щоб анотувати правила більш глибокими поясненнями. Інший підхід - це розміщувати більш глибокі знання в систему і використовувати їх як для висновків, так і для пояснення.

З іншого боку існують системи в яких знання експерта є чисто емпіричними. У цьому випадку пояснення системи корисне для користувача. В цю категорію попадають системи класифікації, наприклад система ідентифікації птахів. Система могла б пояснити ідентифікацію laysan альбатрос правилами, які використовуються для його ідентифікації. Немає ніякої теорії, чому білий альбатрос є laysan альбатрос, а чорний є чорноногим альбатросом. Це просто правила для їх класифікації.

В той час як пояснення системи інколи не має користі для користувача, для розробника ЕС воно має дуже важливе значення. Пояснення ЕС відіграє ту ж діагностичну роль, як і трасування для звичайних програм. Якщо система поводить себе некоректно, експерт може використати пояснення для визначення помилкового правила. Інженер із знань використовує пояснення для кращого настроювання бази знань з метою більш реалістичного діалогу з користувачем.

Існує чотири типи пояснень, які звичайно застосовуються в ЕС:

* трассування правил, яке повідомляє про прогрес консультації;
* пояснення того, як система прийшла до даного висновку;
* пояснення того, чому система задає питання;
* пояснення того, чому система не дала висновку;

### **Модуль набуття знань (компонент набуття знань)**

Модуль набуття є сервісним модулем , який виконує різні допоміжні функції. Робота ЕС тільки тоді буде стабільною, якщо її база знань буде постійно поповнюватисть новими знаннями. До нових знань відносяться знання, які отримані на основі повідомлень з особливостей експлуатації системи, а також знання, що є результатом розвитку данного наукового напрямку.

Процес переносу знань в базу знань системи буже складний і містить багато функцій:

* Функція добування знань. Модуль накопичення в процесі діалогу з експертом повинен виявити нові правила і фрейми, а також виявити ті правила і фрейми, які необхідно видалити або модифікувати. Діалог не повинен вимагати від експерта знань мови програмування.
* Функція структурування знань. Добуті знання повинні бути перетворені з природньої форми, зручної для експерта, у внутрішню форму, прийняту в ЕС.
* Функція перевірки на існування. Перед виконанням додавання, модифікації або видалення із бази знань правила або фрейма, необхідно з'ясувати, чи існує це правило або фрейм.
* Функція додавання. Дає можливість експерту додавати знання в базу знань. Також може бути можливість використання даної функції в процесі роботи ЕС без спеціального звернення до модуля накопичення. Наприклад, якщо необхідна інформація в базі знань не знайдена, ЕС запитує її у користувача і додає до бази знань.
* Функція модифікації. У випадку правило або фрейм існує, користувач повинен мати можливість модифікувати їх.
* Функція видалення. Видаляє ті знання, які застаріли, вступили у протиріччя з новими знаннями або є невірними.
* Функція перевірки на несуперечність

### **Етапи проектування ЕС**

*1. Вибір предметної області. Формулювання мети. Визначення задачі. Розробка алгоритму функціонування експертної системи.*

На першому кроці необхідно провести оцінку предметної області на вірогідність і повноту накопичених знань. У результаті даної оцінки повинні бути конкретизовані джерела знань.

Експертну систему необхідно розглянути як семантичний об'єкт інформаційної семантичної системи. Даний підхід дозволяє сформулювати мету функціонування системи, що визначається конкретним набором елементів. Необхідно конкретизувати даний набір на самому початковому етапі.

При визначенні ресурсів слід уточнити, які види, форми представлення і перетворення семантичної інформації будуть використовуватися в експертній системі.

Визначення задачі, яка розв'язується ЕС, повинне проводитися з функціональної точки зору. Уся множина задач класифікується по наступним групах: задачі ідентифікації; конфігурації; інтерпретації; діагностика; контроль; прогнозування; планування; проектування.

На завершальному кроці даного етапу необхідно розробити узагальнений алгоритм функціонування ЕС.

*2. Добування знань.*

На початку необхідно зафіксувати природною мовою всю надану інформацію з різних джерел знань і уточнити основні поняття (тезаурус). Носіями знань можуть бути : людина-експерт; документація; магнітний носій і ін.

Далі варто виділити основні і допоміжні об'єкти, властивості і відносини для кожного рівня деталізації.

Кількість рівнів деталізації визначається метою створення експертної системи. (Наприклад, якщо в діагностичної ЕС необхідно визначати причину несправності тільки на рівні основних вузлів ЕОМ, то досить одного рівня. Далі всю множину об'єктів необхідно розбити на два види : матеріальні об'єкти, що представляють конструктивну частину ЕОМ і ідеальні об'єкти, що представляють функціональну частину ЕОМ).

Далі для кожного виду об'єктів вибирається спосіб представлення знань. (Наприклад, знання про конструктивну частину ЕОМ доцільніше представити за допомогою фреймів, де кожний із фреймів містить інформацію про адресу місця можливого несправного об'єкта. Знання функціональної частини можна представити продукціями чи семантичною мережею).

Далі визначаються зв'язки між ідеальними і матеріальними об'єктами для кожного рівня деталізації.

*3. Вибір інструментальних засобів проектування.*

На даному етапі необхідно зробити висновок про вибір інструментального засобу, з огляду на результати попередніх етапів. У випадку вибору спеціального програмного середовища, що автоматизують проектування ЭС, чи «оболонки» ЕС подальші етапи розробки помітно спрощуються.

*4. Формалізація знань у вигляді машинних процедур. Побудова бази знань .*

Наступним важливим кроком проектування ЕС є розробка структури бази знань і визначення її прикладної одиниці (базового елемента або структури). Від вирішення цієї задачі залежать: ефективність роботи механізму логічного висновку; дублювання і надмірність інформації бази знань; здійснення діалогу між експертною системою і користувачем; можливість підключення блоку пояснення безпосередньо в ході ведення діалогу; можливість створення блоку нагромадження знань для користувача непрограміста й ін.

Крім вищевказаного, наприклад, в окремому випадку, прикладна одиниця бази знань повинна враховувати зв'язок між функціональною і конструктивною частинами ЕОМ, яка діагностується. При цьому необхідно передбачити можливість участі користувача в процесі одержання висновку експертною системою.

При завершенні даного етапу розробляються вихідні тексти програм, що реалізують базу знань на обраному інструментальному засобі.

1. *Розробка семантичного інтерфейсу «ЕС - користувач».*

Розробка інтерфейсу «ЕС - користувач», основні функції якого: здійснювати перетворення повідомлення з природно-мовної форми у форму внутрішнього представлення і зворотне перетворення; аналіз і синтез повідомлень користувача і ЕС; відстеження і запам'ятовування пройденого шляху діалогу.

На виході даного етапу повинні бути розроблені можливі сценарії діалогу між ЕС і користувачем у виді блок-схем і текстів програм.

*6. Розробка механізму логічного висновку.*

Після того, як знання представлені за допомогою обраних методів, необхідно приступити до розробки механізму логічного висновку (МЛВ).

Основна задача МЛВ, наприклад для продукціної моделі, це реалізація стратегії вибору відповідного правила, факту . Зокрема необхідно розробити чотири процеси: вибір активних правил і фактів; зіставлення; вирішення конфліктів; виконання обраного зазначеного правила (дія).

Основною задачею при реалізації першого процесу є розробка різних схем породження нових фактів і формування правил, спрямованих на досягнення цільових фактів.

Після того, як була отримана множина активних правил виникає необхідність у їхньому зіставленні, тобто визначення які правила максимально охоплюють множину активних фактів і які правила доцільніше виконувати в першу чергу.

Далі необхідно розробити процес вирішення конфліктів. Даний процес у залежності від способу вибору правила повинний обробляти всі можливі конфліктні ситуації.

У результаті повинні бути розроблені алгоритми функціонування ядра експертної системи і тексти програм.

1. *Розробка модуля пояснень.*

Призначення даного модуля - зробити ЕС «прозорою» для користувача, тобто надати користувачу можливість розуміти логіку дій ЕС, дати надійну гарантію правильності отриманих результатів.

На виході даного етапу повинні бути розроблені вихідні тексти програм.

*8. Розробка модуля накопичення знань і маніпулювання зі знаннями.*

Основними функціями модуля накопичення знань є : автоматизація процесу наповнення бази знань; актуалізації бази знань. На цьому етапі розробляються алгоритми функціонування модуля накопичення і екранні форми, що дозволяють здійснювати операції маніпулювання зі знаннями стосовно до обраної структури бази знань. У результаті повинні бути розроблені вихідні тексти програм.

### **Відмінність експертних систем від традиційних програм**

Особливості ЕС, що відрізняють її від звичайних програм:

1. Компетентність:

Досягати експертного рівня рішень (тобто в конкретній предметній області мати той же рівень професіоналізму, що й експерти-люди).

Застосовувати знання ефективно і швидко, уникаючи непотрібних обчислень.

Мати адекватну здатність лише поступово знижувати якість роботи з наближенням до границь діапазону чи компетентності припустимої надійності даних.

2. Можливість до символьних міркувань:

Представляти знання в символьному виді

Переформулювати символьні знання. Символи поєднують, щоб виразити відносини між ними. Коли відносини представлені в ЕС вони називаються символьними структурами.

3. Глибина:

Працювати в предметній області, що містить важкі задачі

Використовувати складні правила (тобто використовувати або складні конструкції правил, або велика їхня кількість)

4. Самосвідомість:

Досліджувати свої міркування (тобто перевіряти їхню правильність)

Пояснювати свої дії

Існує ще одна важлива відмінність ЕС. Якщо звичайні програми розробляються так, щоб щораз породжувати правильний результат, то ЕС розроблені для того, щоб поводитися як експерти. Вони, як правило, дають правильні відповіді, але іноді, як і люди, здатні помилятися.

Традиційні програми для рішення складних задач, теж можуть робити помилки. Але їх дуже важко виправити, оскільки алгоритми, що лежать у їхній основі, явно в них не сформульовані. Отже, помилки нелегко знайти і виправити. ЕС, подібно людям, мають потенційну можливість учитися на своїх помилках.

# 2. Основи програмування на мові Visual Prolog

## 2.1. Загальний огляд мови Пролог

В 1972 р. французькі дослідники Колмерое і Руссель із Марселя, використавши результати теоретичних робіт Лавленда, Ковальскі і Куенера, створили мову логічного програмування, яку назвали Пролог – скорочення від “програмування в термінах логіки”. Розроблений ними інтерпретатор Прологу був написаний на мові Фортран і працював досить повільно. В 1977 р. Уоррен і Перейра із Едінбурзького університету створили ефективний інтерпретатор/компілятор мови Пролог для комп’ютера DEC-10 і сприяли підвищенню її популярності. З тих пір було створено багато діалектів мови Пролог: СіПролог, Еріті Пролог, Турбо Пролог, Strawberry Prolog, Visual Prolog та інші.

Visual Prolog 5.2 це середовище програмування для мови Visual Prolog, створене Prolog Development Center A/S (Данія). Воно містить все необхідне для створення великих комерційних додатків, включаючи компілятор, компоновщик і велику бібліотеку підпрограм.

Пролог дуже відрізняється від інших традиційних мов програмування, таких як Бейсік, Паскаль чи Сі. Програміст, створюючи програму, наприклад на Сі, вказує як програма повинна працювати , щоб привести до бажаної цілі (тобто визначає процедуру вирішення задачі). Якщо програміст пише програму на Пролозі, він спочатку описує структуру прикладної задачі, а потім вказує що треба знайти, тобто вказує ціль. Його не цікавить яким чином ця задача буде вирішуватися, це питання він перекладає на Пролог-систему. Тому Пролог відноситься до декларативних (описувальних) мов програмування, а Бейсік, Паскаль та Сі до процедурних мов. Якщо класифікувати мови програмування за рівнем, то Асемблер можна віднести до мов низького рівня, Сі до мов середнього рівня, Фортран, Бейсік, Паскаль до мов високого рівня, Пролог до мов надвисокого рівня.

## 2.2. Основні теоретичні відомості

### **Основні визначення мови Visual Prolog (далі Пролог)**

Пролог-програма складається із фраз (речень). Кожна фраза закінчується точкою. Фрази Прологу бувають трьох типів: факти, правила і питання.

1. Факти містять твердження, які є завжди, безумовно вірними. Факт – це твердження, яке визначає відношення між об’єктами, або описує ознаки об’єкта. Формат запису факту має такий вигляд:

**<ім’я відношення>(ім’я\_об’єкта1, ім’я\_об’єкта2,…, ім’я\_об’єкта).**

При запису факту треба додержуватися наступних правил:

* Імена всіх відношень і об’єктів повинні починатися із малої літери;
* Спочатку записується ім’я відношення, потім в круглих дужках через кому записуються імена об’єктів;
* Кожен факт повинен закінчуватися точкою;
* Імена об’єктів в дужках перераховуються в одному вибраному порядку.

Наприклад факти з одним та двома об’єктами можуть бути записані таким чином:

**car(black).**

**speak(liza, english).**

На природній мові ці факти означають: “Автомобіль – чорний” та “Ліза говорить по - англійськи”.

2. Правило - містить твердження, істинність якого залежить від деяких умов.

Формат запису правила має вигляд:

**голова правила>:- <тіло правила.**

Голова - це факт, який буде вірним, якщо виконаються умови, які знаходяться в тілі правила. Умови у тілі правила являють собою терми, зв`язані

між собою комою або крапкою з комою ("," означає "і", а ";" означає "або").

Символ ":-", який стоїть між головою та тілом правила означає "якщо".

Наприклад правило

**buy(peter, car) :- like(peter, car), have(peter, money).**

інтерпретується таким чином: "Пітер купує автомобіль, якщо автомобіль подобається Пітеру і Пітер має гроші".

3. Питання – складаються із одного або декількох цільових тверджень. Пролог-система розглядає питання як ціль, якої треба досягти. Питання можна розглядати як правило, яке має тільки тіло. У Visual Prolog питання розміщуються у розділі goal. Наприклад, питання: **buy(peter, car).** інтерпретується “Чи купив Пітер автомобіль?”.

1. Атом - це ім'я , число без знаку або символ: **liza, 12, z**.
2. Змінна – абстрактний об’єкт, ім’я якого починається з великої літери або символу підкреслювання: **X, Y, Man, \_number**.

6. Аргумент - ім'я об'єкта в круглих дужках.

7. Домен - тип даних.

1. Предикат - це символьне ім’я (ідентифікатор), яке пов’язує відношення з його аргументами:

**family(peter, joan, kate, bob)**, в даному випадку **family** – предикат, а **peter, joan, kate, bob** – аргументи.

1. Функтор - ім'я складного об'єкта:

**date(3, october, 2001)**, **date** - функтор.

1. База даних - в Пролозі означає сукупність фактів.
2. Уніфікація - це процес перевірки двох термів на відповідність одне одному.
3. Терми - всі об'єкти даних в Пролозі: **date(3, october, 2001), october**.

## 2.3. Структура програми на мові Visual Prolog

Програма на Пролозі складається із чотирьох основних розділів (секцій):

* Розділ **domains** призначений для оголошення доменів, які не є стандартними доменами Прологу.

В традиційному Пролозі є тільки один тип – домен. У Visual Prolog треба оголошувати домени аргументів предикатів. Розділ domains виконує дві корисні цілі. По-перше, можна давати смислові імена доменам, навіть якщо вони по-суті є стандартними доменами Прологу. По-друге, можна оголошувати структури даних, які не є стандартними доменами. Visual Prolog 5.2 підтримує такі домени:

1. Базові стандартні домени

- char (символи) – окремий символ, або символ після знаку “\” розміщений в апострофах, ‘t’, ‘0’, ‘\\’ (бекслеш “\”), ‘\”’ (лапки “), ‘\’’ (апостроф ‘), ‘\n’ (перехід на наступний рядок), ‘\t’ (табуляція), Після знаку “\” можна використовувати ASCI код символу: ‘\65’ (символ А), ‘\10’ (символ пробілу).

- real (дійсні числа) – довжина типу 8 байт.

- string (рядки) – послідовність символів в лапках, наприклад: “Bonny”, “1991”, “Київ”.

- symbol (символічні імена) – послідовність літер, цифр і знаків підкреслювання, яка починається із літери нижнього регістру, наприклад: cat, uncle\_aunt, sky5.

1. Інтегральні стандартні домени:

- integer (цілі числа) – довжина типу залежить від платформи. Для 32-бітної платформи довжина має 4 байти.

- byte, sbyte, short, ushort, word, unsigned, dword, long, ulong – див. довідку Visual Prolog 5.2.

3. Бінарний домен:

- binary - див. довідку Visual Prolog 5.2.

4. Особливо оброблювані визначені домени (Specially Handled Predefined Domains)

- accessmode, bgi\_ilist, bt\_selector, db\_selector, dbasedom, denymode, file, place, ref, and reg - див. довідку Visual Prolog 5.2.

* Розділ **predicates** призначений для оголошення предикатів та доменів аргументів предикатів.

Формат оголошення предикату такий:

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

Ставити крапку в кінці оголошення непотрібно. В якості типу аргументу використовуються стандартні домени, або домени, оголошені в розділі domains. Ім’я предикату повинне починатися з літери, за яким ідуть літери, цифри або знаки підкреслювання (\_). Максимальна довжина імені предикату – 250 знаків.

* Розділ **clauses**  - “серце” програми на Пролозі. В ньому розміщуються факти і правила, які використовуються коли Пролог намагається задовольнити ціль програми (запит).

Речення для даного предиката повинні розміщуватися в розділі clauses. Послідовність речень, які визначають предикат називається процедурою.

Намагаючись задовольнити ціль (питання в розділі **goal**) Пролог починає перевіряти речення з верху секції **clauses**. Ідучи вниз Пролог розміщує зовнішні покажчики на кожному реченні, які задовольняють поточну підціль. Якщо речення не є частиною логічного шляху, який веде до рішення, Пролог повертається до встановленого покажчика і шукає речення, яке задовольняє підціль (цей процес називається бектрекінг).

* Розділ **goal**, в якому розміщується стартова ціль програми.

По-суті, розділ goal це те саме, що і тіло правила, тобто просто список підцілей. Пролог автоматично виконує ціль, коли стартує програма. Якщо всі підцілі задовольняються, програма закінчується успішно. В іншому разі кажуть, що програма закінчується неуспіхом.

Наприклад:

**DOMAINS**

**author, name, publisher = symbol**

**price = byte**

**quantity = ulong**

**PREDICATES**

**nondeterm book(author,name,publisher,price,quantity)**

**CLAUSES**

**book(alexander\_abramov,riders\_from\_nowhere,raduga,15,200).**

**book(agatha\_christie,selected\_stories,progress,20,150).**

**book(jennifer\_niederst,html,o\_reilly,10,100).**

**GOAL**

**book(agatha\_christie,selected\_stories, \_ , \_ ,X).**

Дана програма знаходить у базі даних кількість книжок автора Агата Крісті з назвою “Вибрані оповідання”. Нас не цікавлять видавництво й ціна, тому вони позначені анонімними змінними – знаком підкреслювання. Значення анонімних змінних не виводяться при відповіді системи на запит.

Крім перелічених, Пролог має ще такі розділи.

* Розділ **database** (або **facts**) – динамічна база даних. В ній розміщуються факти, які можуть змінюватися, видалятися чи додаватися в базу даних під час виконання програми.
* Розділ **constants** містить оголошення символічних констант. Синтаксис оголошення:

**<Id> = <Macro definition>**

де **<Id> -** ім’я символічної константи, **<Macro definition>** - значення, яке присвоюється константі. Наприклад:

**CONSTANTS**

**zero = 0**

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

**pi = 3.141592653**

Існує декілька обмежень при використанні символьних констант: визначення констант не може посилатися саме на себе; система не відрізняє літери верхнього і нижнього регістрів, тому в розділі **clauses** перша літера імені символьної константи повинна бути нижнього регістру, щоб не сплутати константу зі змінною; у програмі може бути декілька розділів **clauses,** але оголошення символьної константи повинне бути до її використання.

* Розділи **global domains**, **global predicates**, and **global facts** дозволяють оголошувати глобальні домени, предикати та речення в програмі. Дані розділи розміщують на початку програми.

## 2.4. Предикати введення - виведення

Серед предикатів введення і виведення найчастіше використовуються такі:

* **readchar (CHAR CharVariable)** – зчитує з клавіатури один символ.
* **readint(INTEGER IntegerVariable) –** зчитує з клавіатури ціле число.
* **readln(STRING StringVariable) –** зчитує з клавіатури рядок символів.
* **readreal(REAL RealVariable) –** зчитує з клавіатури дійсне число.
* **readterm(<domainName>, <TERM> Term) –** зчитує з клавіатури терм даного домену.
* **nl –** перехід на наступний рядок.
* **write(e\_1, e\_2, e\_3, … , e\_N) –** виведення декількох аргументів на екран.
* **writef(STRING FormatString, Arg1, Arg2, Arg3, …. ) –** здійснює форматоване виведення на екран (аналог printf() у мові С).

З іншими предикатами введення виведення можна ознайомитись у розділі **Input/Output standard predicates** довідки Visual Prolog.

## 2.5. Об’єкти даних

Об’єкти даних можуть бути простими та складними (або структурами). Простими даними виступають *змінні* і *константи*. Константа може бути символьною (**char**), числовою (**integer**, **real**) або атомарною (**symbol, string**).

Складні об’єкти (далі структури) – це об’єкти які складаються із декількох компонент. Ці компоненти також можуть бути структурами. Наприклад, дату можна розглядати як структуру, яка складається із трьох компонентів: день, місяць та рік. Для об’єднання цих компонентів у структуру треба вибрати так званий функтор - ім'я складного об'єкта. У даному випадку можна використати функтор – data:

**date(1, april, 2000).**

Структури можна зображувати у вигляді дерев. Коренем дерева є функтор, гілками – компоненти. Якщо сам компонент є структурою, тоді він зображується у вигляді піддерева. Наприклад, якщо об’єкт – комп’ютер, представити як сукупність таких компонент як процесор, об’єм оперативної пам’яті і монітор, у свою чергу процесор представити як тип, тактова частота і об’єм кеш пам’яті, то структуру computer(processor(Pentium - 4, 3.0, 1024 ), 512, Samsung 913N) можна зобразити таким деревом:

computer

/ | \

processor 512 Samsung 913N

/ | \

Pentium 4 3.0 1024

### **Завдання 1.**

Використовуючи предикати **parent(string parent, string son-daughter), women(symbol)** і **man(symbol)**, розробити предикати, що визначають такі поняття:

1. Правнучка
2. Двоюрідний брат.
3. Внук
4. Дівер
5. Зять
6. Тесть
7. Свекруха
8. Теща
9. Свекор
10. Прадідусь
11. Племінник
12. Прабабуся
13. Племінниця
14. Свояк
15. Тітка
16. Своячка
17. Дядько
18. Дідусь
19. Правнук
20. Внучка
21. Бабуся

Написати програму для пошуку і видачі на екран відповідного родича в базі даних.

## 2.6. Вбудовані механізми мови Пролог. Управління бектрекінгом

В основі роботи Пролог-системи лежать два механізми – уніфікація та бектрекінг. Уніфікація – це процес перевірки на збіг двох об’єктів і конкретизація вільних змінних. Правила уніфікації такі:

1. Якщо два об’єкта – константи, то вони уніфікуються, якщо вони є одним і тим же об’єктом.
2. Якщо один об’єкт – змінна, а інший довільний об’єкт, то вони уніфікуються, і змінній приписується значення довільного об’єкта.
3. Якщо два об’єкта – структури, то вони уніфікуються тільки якщо: а) обидва вони мають однаковий головний функтор; б) усі їх відповідні компоненти уніфікуються.

Приклади об’єктів, які уніфікуються: константи 2 і 2, ‘s’ і ‘s’; змінна X і структура data(12, May, 2004), в результаті змінній X припишеться значення data(12, May, 2004); структура data(X, Y, 2004) і структура data(12, May, 2004), в результаті змінним X i Y припишуться значення 12 і May.

Бектрекінг – це механізм циклічного рекурсивного повернення для знаходження додаткових фактів і правил, необхідних для обчислення цілі, якщо поточна спроба обчислення цілі виявилася невдалою.

Таким чином роботу Пролог-системи можна назвати як рекурсивний циклічний процес уніфікації і обчислення підцілей.

Розглянемо для пояснення такий приклад.

**domains**

**parent = symbol**

**son\_daughter = symbol**

**facts**

**parent(parent, son\_daughter)**

**women(symbol)**

**man( symbol)**

**predicates**

**nondeterm son\_daughter(son\_daughter, parent)**

**clauses**

**parent(pam, bob).**

**parent(bob, ann).**

**women(pam).**

**women(ann).**

**man(bob).**

**son\_daughter(X, Y):- parent(Y, X), man(X).**

**son\_daughter(X, Y):- parent(Y, X), women(X).**

**goal**

**son\_daughter(ann, bob).**

Пролог-системі задано питання чи є Анн дочкою або сином Боба - **son-daughter(ann, bob).**. Пролог система починає з цілі і застосовуючи правила, підмінює поточні цілі новими, до тих пір, поки ці нові цілі не стануть простими фактами. Спочатку Пролог-система шукає таке речення в програмі голова якого уніфікується з ціллю, тобто з **son-daughter(ann, bob)**. Такими реченнями є два речення, які стосуються предикату **son-daughter**.

Спочатку Пролог-система пробує речення, яке стоїть в програмі першим:

**son-daughter(X, Y):- parent(Y, X), man(X).**

Після уніфікації структур **son-daughter(ann, bob)** та **son-daughter(X, Y)** значення змінних будуть **X=ann, Y=bob**. Тоді початкова ціль замінюється двома новими цілями **parent(bob, ann)** і **man(ann)**. Перша ціль досягається, оскільки вона уніфікується з фактом із програми, друга не досягається – вона не уніфікується з жодним фактом чи правилом програми. Ціль **man(ann)** виявляється неуспішною.

Тепер Пролог-система робить повернення до початкової цілі **son-daughter(ann, bob)** (бектрекінг), щоб спробувати другий варіант виводу цілі, тобто:

**son-daughter(X, Y):- parent(Y, X), women(X).**

Аналогічно, Пролог-система приходить до двох цілей parent(bob, ann) і women(ann). Так як обидві цілі уніфікуються з фактами програми, вони досягаються. Таким чином запит до Пролог-системи виявився успішним і вона дасть позитивну відповідь - **yes**.

Для управління процесом бектрекінгу існує два вбудованих предиката **fail**, та **cut** або **!** (предикат відсікання). Предикат **fail** завжди має хибне значення і тому запускає механізм бектрекінгу, а предикат **cut** або **!** навпаки не дозволяє бектрекінг. **fail** використовується для організації циклів (див. наступний розділ). Предикат відсікання **!** застосовується для обмеження автоматичного перебору, для підвищення ефективності програми, або якщо він не потрібен.

Розглянемо, наприклад, таку функцію:

якщо Х < 2, то Y = 10

якщо Х >= 2 i Х < 4, то Y = 20

якщо Х >= 4, то Y = 30

На Пролозі дана функція реалізується таким чином:

**f(X, 10) :- X < 2.**

**f(X, 20) :- X >= 2, X < 4.**

**f(X, 30) :- X >= 4.**

Якщо тепер зробити Пролог-системі такий запит - **f(0, Y), Y=5,** то при досягненні першої підцілі **f(0, Y)** за першим правилом **f(X, 10) :- X < 2 Y** конкретизується 10. Друга підціль **Y=5** виявиться хибною, так як 10 не дорівнює 5. Запуститься механізм бектрекінгу, Пролог-система знову повернеться до підцілі **f(0, Y)** і невдало намагатиметься задовольнити її за другим і третім правилом **f(X, 20) :- X >= 2, X < 4. , f(X, 30) :- X >= 4.** В решті решт запит **f(0, Y), Y=5** закінчиться невдачею (результат - **No**). Якщо проаналізувати роботу програми, виявиться, що намагання Пролог-системи задовольнити підціль **f(0, Y)** за другим і третім правилами даремні, тому що всі три правила взаємовиключні (якщо виконується перше правило, інші два виконуватись не будуть; **X** не може одночасно бути меншим за **2** і більшим або дорівнює **2** і т. д.). Тому в даному випадку бектрекінг не має сенсу і його можна заборонити таким чином:

**f(X, 10) :- X < 2, !.**

**f(X, 20) :- X >= 2, X < 4, !.**

**f(X, 30) :- X >= 4.**

Тепер, у випадку запиту **f(0, Y), Y=5,** після того як за першим правилом **Y** конкретизується 10, а потім підціль **Y=5** виявиться хибною Пролог-система не зможе повернутися далі точки поміченої в програмі предикатом **!.** Пролог-система зразу видасть результат запиту (як і в першому варіанті програми - **No**), не включаючи в роботу даремно друге і третє правила, що підвищить швидкість виконання програми. На декларативний смисл програми введення предикату **!** не вплинуло, змінився тільки процедурний смисл, тому результати запитів до програми не зміняться, якщо вилучити предикати **!** з програми.

Якщо застосування предикату **!** не впливає на декларативний смисл програми, то він називається “зелений **cut**”, в іншому випадку – “червоний **cut**”. Фахівці рекомендують конструювати програми таким чином, щоб не використовувати “червоний **cut**” (або використовувати якомога менше)

## 2.7. Організація циклів. Рекурсія

Циклічні процеси в Пролозі можна організовувати двома способами – використовуючи бектрекінг та за допомогою рекурсії.

Загальний вид правила, який виконує повторення за допомогою бектрекінгу наступний:

**rule :-**

**<предикати>,**

**fail.**

Наприклад, наступна програма друкує назви всіх міст, які знаходяться в базі даних (увага – щоб включити підтримку кирилиці, необхідної для запуску цієї програми, треба виконати наступні дії – вибрати пункт меню **Options – Global (.INI)-Environment**, далі у вікні **“Environment”** перейти на вкладку **“Fonts”** і у області **“Editor Windows”** натиснути кнопку **“Change Font”,** далі у вікні **“Шрифт”** у списку **“Набор символов”** вибрати **“Кириллический”** і **“OK”**).

**domains**

**town = string**

**facts**

**town(town)**

**predicates**

**print\_town**

**clauses**

**town(“Київ”).**

**town(“Одеса”).**

**town(“Харків”).**

**print\_town:-**

**town(X), write(X), nl, fail.**

**print\_town.**

**goal**

**print\_town.**

Процедура, яка реалізує предикат print\_town складається із двох речень. Перше речення **town(X), write(X), nl, fail**. говорить: “Знайти розв’язок предикату town(X) , надрукувати цей розв’язок, перейти на наступний рядок, потім повернутись і пошукати інший розв’язок”. Повернення і пошук іншого розв’язку забезпечує вбудований предикат **fail**, який завжди має хибне значення. Таким чином він форсує бектрекінг.

Без другого речення **print\_town.** виконання програми закінчувалося б неуспіхом і після друкування всіх міст на екрані з’являлося б слово **No**.

Інший цікавий спосіб застосування бектрекінгу для організації циклу є за допомогою предикату **repeat,** який визначається таким чином:

**repeat.**

**repeat :- repeat.**

Наприклад, наступна програма друкує на екрані всі символи, які вводить користувач, і закінчує роботу при вводі символу “.” (крапка).

**predicates**

**nondeterm repeat**

**nondeterm print**

**clauses**

**repeat.**

**repeat :- repeat.**

**print :- repeat,**

**readchar(X), write (X), X=’.’, nl.**

**goal**

**print.**

Процедура **print** виконується наступним чином. Спочатку виконується **repeat**, який нічого не робить, далі зчитується з клавіатури в змінну Х символ, друкується на екран, перевіряється на рівність крапці “.”. Якщо дорівнює, відбувається перехід на наступний рядок і процедура закінчується. Якщо ні, починається бектрекінг і перегляд альтернатив. Ні **write (X),** ні **readchar(X),** не породжують нових альтернатив, тому бектрекінг веде далі назад до **repeat,** який дозволяє отримати альтернативні розв’язки. Управління іде знову на початок, зчитується символ і т. д.

В даній програмі предикати **repeat** і **print** оголошуються із ключовим словом **nondeterm.** Це означає, що ці предикати можуть генерувати альтернативні множинні рішення. Такі предикати можуть також терпіти неуспіх і в цьому випадку не породжувати ніяких рішень.

Для організації повторів в Пролозі також використовується рекурсія. Як відомо, процедура називається рекурсивною, якщо вона безпосередньо або через інші процедури звертається сама до себе. Загальний вигляд рекурсивної процедури, яка складається із двох правил, в Пролозі такий:

**rule :- <predicates>. % нерекурсивне правило**

**rule :- <predicates>, % рекурсивне правило**

**rule.**

Нерекурсивне правило використовується для виходу із рекурсії, а рекурсивне – для організації рекурсії. Прикладом використання рекурсії може бути програма обчислення факторіалу числа – n!.

**domains**

**chyslo = integer**

**factorial = integer**

**predicates**

**factorial(chyslo, factorial)**

**clauses**

**factorial(1, 1):- !.**

**factorial(N, FactN):- N\_1=N-1,**

**factorial(N\_1, FactN\_1), FactN=N\*FactN\_1.**

**goal**

**factorial(5, N), write(N), nl.**

Перше правило процедури **factorial** формулюється так:

Факторіал числа один дорівнює одиниці (тобто **1!=1)**.

Друге, рекурсивне правило говорить:

Число **FactN** є факторіалом числа **N**, якщо воно є добутком числа **N** на факторіал числа **N-1** (тобто **N!=N\*(N-1)!)**.

Хоча, як видно з попереднього прикладу, рекурсію можна застосовувати для проведення обчислень, в Пролозі рекурсія частіше використовується для визначення рекурсивних відношень. Класичним прикладом рекурсивного відношення є відношення **предок**. Ось приклад процедури, що реалізує відношення **предок**:

**предок(X, Z):-**

**батько(X, Z).**

**предок(X, Z):-**

**батько(X, Y), предок(Y, Z).**

Перше і друге правила формулюється відповідно так:

Для всіх X і Z, X є предком Z, якщо X – батько Z.

Для всіх X і Z, X є предком Z, якщо існує такий Y, що X є батьком Y і Y – предок Z.

Як видно, перше правило рекурсивної процедури відображає окремий випадок відношення (предком є батько), а друге загальний випадок (предком є будь-який родич по висхідній лінії). Це – загальний підхід до визначення рекурсивних відношень.

## 2.8. Використання динамічної бази даних

Як було зазначено вище, динамічна база даних знаходиться у розділі програми **database** (або **facts**). В ній розміщуються факти, які можуть змінюватися, видалятися чи додаватися в базу даних під час виконання програми.

Найбільш поширені вбудовані предикати для роботи з динамічною базою даних такі:

* **assert(<facts\_domain> Fact)** – вставляє факт **Fact** в кінець динамічної бази даних, в якій зберігаються подібні факти.
* **asserta(<facts\_domain> Fact)** - вставляє факт **Fact** на початок динамічної бази даних, в якій зберігаються подібні факти.
* **assertz(<facts\_domain> Fact)** – аналог **assert(<facts\_domain> Fact)**.
* **consult(STRING OSFileName)** – зчитує факти з текстового файлу **OSFileName** в неіменовану динамічну базу даних.
* **nondeterm retract(<facts\_domain> Fact)** – видаляє факт **Fact** із динамічної база даних.
* **determ retractall(<facts\_domain> Fact)** - видаляє всі факти **Fact** із динамічної база даних.
* **save(STRING OSFileName)** – зберігає зміст неіменованої динамічної бази даних в текстовий файл **OSFileName.**

Повний список вбудованих предикатів для роботи з динамічною базою даних дивіться в довідці по Visual Prolog в розділі **Internal Facts Sections standard predicates**.

Наступна програма демонструє використання вбудованих предикатів для заповнення динамічної бази даних в момент виконання програми. В базі даних містяться назва, ціна та кількість товару, яка знаходиться в магазині. Ми можемо доповнити базу даних новими товарами під час виконання програми, а в кінці вся сукупність товару виводиться на екран.

**domains**

**dom\_tovar = tovar(symbol,real,integer)**

**database**

**tovar(symbol tovar,real price,integer quantity)**

**predicates**

**start**

**greeting**

**nondeterm input\_tovar**

**nondeterm output\_tovar**

**clauses**

**tovar(milk, 2.5, 100).**

**tovar(butter, 10.5, 50).**

**start:-**

**greeting, input\_tovar, output\_tovar,!.**

**greeting:-**

**write("Введіть назви товарів, ціну, вартість в базу даних, наприклад, tovar(\”tea\”, 3.4, 100).",'\n',**

**"Назву товару введіть в лапках.","Для закінчення введення введіть крапку."), nl.**

**input\_tovar:-**

**readchar(X),X='.', !.**

**input\_tovar:-**

**readterm(dom\_tovar, tovar(X,Y,Z)), assert(tovar(X,Y,Z)), input\_tovar.**

**output\_tovar:-**

**tovar(X,Y,Z), write(X," ",Y," ",Z), nl, fail.**

**goal**

**start.**

## 2.9. Рекомендації для створення програм на Пролозі

Для того,щоб написання програми проходило якомога легше і швидше, необхідно спочатку продумати задачу, яку треба вирішити. Частою помилкою є намагання написати програму навіть до того, як стала повністю зрозуміла докладна постановка задачі. На цьому етапі знаходиться загальний спосіб розв’язку задачі

Як рекомендує знаний фахівець Іван Братко [див. Братко И. “Программирование на языке Пролог для искусственного интеллекта”], трансформацію знайденого загального способу розв’язку задачі в програму можна здійснити використовуючи принцип покрокової деталізації. Загальний спосіб розв’язку задачі розглядається як “розв’язок верхнього рівня”, а остаточна програма – як “розв’язок найнижчого рівня”.

Остаточна програма виходить після серії “деталізацій” розв’язку. На кожному кроці деталізації розв’язок стає все більш і більш детальним, поняття стають більш повними, а їх представлення наближається до мови програмування. У випадку Прологу можна говорити про покрокову деталізацію відношень (або предикатів).

Розглянемо застосування викладених рекомендацій на прикладі створення програми даної в попередньому розділі.

Отже, дана постановка задачі – створити програму, в якій знаходиться база даних назв, цін та кількості товару і користувач може доповнювати цю базу даних під час виконання програми. Після заповнення бази даних весь її зміст виводиться на екран.

На нульовому кроці деталізації просто позначимо виконання нашої задачі через предикат **start**. Далі, видно, що дана програма повинна забезпечувати принаймі дві дії – введення користувачем нового товару в базу даних і виведення бази даних на екран. Можна, також, забезпечити і початкове вітання програми до користувача. Отже, на першому кроці деталізації закріпимо кожну з ціх трьох дій за своїми власними предикатами, наприклад, **greeting, input\_tovar, output\_tovar**. Таким чином, наш предикат **start** деталізується до трьох предикатів.

На другому кроці деталізуємо кожен із цих предикатів. Нехай предикат **greeting** виводить пояснюючий до програми напис, що можна реалізувати за допомогою вбудованого предикату виведення **write()**.

Предикат **input\_tovar** повинен вводити з клавіатури певні дані, в яких міститься інформація про товар і заносити їх в базу даних. Нехай дані про товар будуть знаходитись в структурі **tovar(symbol tovar,real price,integer quantity)**, де першим аргументом буде назва товару, другим – ціна, третім – кількість товару. Отже предикат **input\_tovar** буде вводити з клавіатури довільну кількість структур типу **tovar,** ми не можемо сказати заздалегідь скільки. Значить цей предикат треба реалізувати як циклічний процес, наприклад, через рекурсивну процедуру (див. розділ “Організація циклів. Рекурсія.”). Рекурсивна процедура, як правило, складається із двох правил – першого нерекурсивного, і другого рекурсивного. Нехай вихід із рекурсії буде здійснюватися при введенні крапки. Тоді перше правило запишеться так:

**input\_tovar:-**

**readchar(X),X='.', !.**

В другому, рекурсивному правилі, треба спочатку ввести терм **tovar** з клавіатури, потім вставити цей факт в базу даних і знову викликати предикат **input\_tovar**. Отже друге правило запишеться таким чином:

**input\_tovar:-**

**readterm(dom\_tovar, tovar(X,Y,Z)), assert(tovar(X,Y,Z)), input\_tovar.**

Предикат **output\_tovar** повинен знаходити всі факти **tovar** в базі даних і виводити їх на екран, тобто це також циклічний процес. Реалізуємо його за допомогою предикату **fail:**

**output\_tovar:-**

**tovar(X,Y,Z), write(X," ",Y," ",Z), nl, fail.**

### **Завдання 2.**

Вирішити задачі, аналогічні до тих, які дані попередньому завданні 1 (див. розділ “Об’єкти даних”), але родинне дерево з фактами повинне знаходитись в динамічній базі даних. Користувач має сам заповнити цю базу даних під час виконання програми, а потім записати її в файл. Можна потім зчитати її з файлу. Програма повинна мати меню, яке дозволяє виконувати команди користувача, наприклад:

1-заповнити базу даних

2- записати базу в файл

3- зчитати базу із файлу

4- знайти прадіда

5- вихід

## 2.10. Рекурсивні структури даних

Рекурсивними можуть бути не тільки правила, а й структури даних. Пролог належить до тих мов програмування, в яких дуже зручно можна організовувати, описувати і реалізовувати дані рекурсивного типу. Тип даних буде рекурсивним, якщо він являє собою структуру, яка в своєму визначенні використовує структуру, подібну собі.

Найбільш вживаною рекурсивною структурою є список, але ми його розглянемо пізніше. В цьому розділі ми зосередимо свою увагу на структурі даних типу дерева і на її використанні.

### **Структура даних типу дерева**

Серед структур даних типу дерева можна виділити спеціальний клас найбільш вживаних дерев- бінарні дерева. Можна дати рекурсивне визначення бінарного дерева:

1. Порожнє дерево- бінарне дерево.

2. Кожний вузол бінарного дерева має не більше одного лівого бінарного піддерева і не більше одного правого бінарного піддерева.

Таку структуру даних в Пролозі можна визначити за допомогою двох предикатів:

**treetype = tree(string,treetype,treetype) та empty**

Останній використовується для позначення порожнього дерева. Тому структура даних для задання бінарного дерева може бути описана наступним чином:

**domains**

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

**empty**

Наприклад, дерево зображене на рис.7

**Перро**

**/ \**

**/ \**

**Грімм Кіплінг**

**/ \ / \**

**/ \ / \**

**Гауф Лондон Уайльд Лагерлеф**

**Рис 7. Приклад дерева.**

може бути описане:

**tree("Перро",tree("Грімм",tree("Гауф",empty, empty),**

**tree("Лондон" ,empty, empty)),**

**tree("Кіплінг", tree("Уайльд",empty, empty),**

**tree("Лагерлеф" ,empty, empty)))**

Зазначимо, що це не є прологівсьгою фразою; це є тільки складною структурою даних.

### **Обходи дерева**

Існує багато правил обходу дерев: прямий, зворотній, кінцевий і т.д.. Наприклад, розглянемо алгоритм прямого обходу.

1. Якщо дерево порожнє, тоді нічого не робити.

2. В іншому випадку, обробити поточний вузол, потім обійти ліве піддерево обробленого вузла, а потім обійти праве піддерево обробленого вузла.

В Пролозі цей алгоритм реалізується за допомогою двох фраз:

**traverse(empty).**

**traverse(tree(X,Y,Z):- do something with X, traverse(Y),**

**traverse(Z).**

Для практичної реалізації розглянемо програму, яка обходить дерево і друкує значення, що містяться у вузлах дерева.

**domains**

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

**empty()**

**predicates**

**print\_all\_elements(treetype)**

**clauses**

**print\_all\_elements(empty).**

**print\_all\_elements(tree(X,Y,Z)):- write(X), nl,**

**print\_all\_elements(Y),**

**print\_all\_elements(Z).**

**goal: print\_all\_elements(tree("Перро", tree("Грімм",**

**tree("Гауф", empty, empty),**

**tree("Лондон", empty, empty)),**

**tree("Кіплінг", tree("Вайльд", empty,**

**empty), tree("Лагерлеф", empty,**

**empty)))).**

Для багатьох задач виникає потреба створення структури даних типу дерева в пам'яті комп'ютера з подальшою її обробкою.

Створити один вузол дерева тривіально:

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

Цей предикат можна інтерпретувати: "Якщо N є елементом даних, tree(N,empty,empty) є вузлом дерева,який містить цей елемент.

Процес побудови дерева трохи складніший. Розглянемо фразу, яка містить предикат з трьома аргументами типу дерева. Він вставляє перше дерево як ліве піддерево другого дерева, використовуючи третє дерево як результуюче:

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

Припустимо, наприклад, що ми хотіли б вставити

**tree('Грім ' , empty, empty)** як ліве піддерево

**tree('Перро', empty, empty).** Це можна зробити, сформувавши запит:

**goal : insert\_left(tree(' Грім ',empty,empty),**

**tree('Перро ',empty,empty),**

**T)**

де **T** зразу ж прийме значення **tree('Перро', tree(Грім ',empty,empty), empty).**

Отже, ми поетапно можемо побудувати дерево. Наступна програма демонструє використання запропонованого підходу для побудови дерева. На практиці, елементи, які розміщуються у вузлах дерева, майже завжди беруться із зовнішнього файлу (увага - для виконання цієї програми необхідно у **Visual Prolog** вибрати в меню **Project- New project**, ввести ім’я проекту і ім’я файлу проекту, перейти на вкладку **target**, у списку **UI Strategy** вибрати **Easywin**, створити проект і скопіювати текст програми у файл із розширенням **.pro**, згенерований **Visual Prolog**'ом. Для запуску програми скористатися командою **R** на панелі інструментів)

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

**\* Прості процедури побудови дерева. \***

**\* create\_tree(A, B) вставляє A в поле даних дерева B \***

**\* insert\_left(A, B, C) вставляє A в як ліве піддерево B, \***

**\* отримуючи дерево C \***

**\* insert\_right(A, B, C) вставляє A в як праве піддерево B, \***

**\* отримуючи дерево С \***

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

**domains**

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

**empty()**

**predicates**

**create\_tree(string, treetype)**

**insert\_left(treetype, treetype, treetype)**

**insert\_right(treetype, treetype, treetype)**

**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)).**

**goal**

**/\* Спочатку створимо вузли (однорівневі дерева)... \*/**

**create\_tree("Гауф", Ch), create\_tree("Лондон", H),**

**create\_tree("Грімм", Mi), create\_tree("Вайльд", J),**

**create\_tree("Лагерлеф", E), create\_tree("Кіплінг",Me),**

**create\_tree("Перро", Ca),**

**/\* ... потім встановимо зв`язки \*/**

**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),**

**/\* ... і роздрукуємо результат. \*/**

**write(Ca3), nl.**

Так, як у Пролозі ми не можемо змінити значення змінної, а можемо лише зв'язати змінну з якимсь значенням, дана програма містиь так багато змінних.

### **Бінарний пошук на дереві**

Ми описали використання дерева для задання відношень між його елементами. Це не кращий спосіб використання дерева в Пролозі, оскільки люба фраза Прологу може зробити подібну роботу. Але використання дерева має інші переваги. Розглянемо спеціальний тип бінарного дерева, яке називають бінарним деревом пошуку.

Воно будується у такий спосіб:

1. Якщо поточний вузол є порожнім деревом, тоді вставити елемент в нього.

2. В іншому випадку, порівняти елемент, що вставляється з елементом, який зберігається в поточному вузлі. Якщо значення нового елементу менше, тоді вставити новий елемент в ліве піддерево поточного вузла, а в іншому випадку - вставити в праве піддерево поточного вузла.

Прологівська реалізація побудови бінарного дерева пошуку вимагає наявності трьох фраз. Перша:

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

позначає, "результатом вставки елементу NewItem в порожнє дерево буде дерево **tree(NewItem, empty,empty) .**

Друга й третя фрази реалізують відповідно лівосторонню або правосторонню вставку в не порожнє дерево:

**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).**

### **Сортування по дереву**

Якщо бінарне дерево пошуку вже побудоване, дуже просто розмістити всі його елементи в лексико-графічному порядку. Рекурсивний алгоритм буде включати зворотний обхід дерева:

1. Якщо дерево порожнє, тоді нічого не варто робити.

2. В іншому випадку, переглянути усі елементи лівого піддерева, потім поточний елемент, потім всі елементи правого піддерева.

Або ж у Пролозі:

**retrieve\_all(empty).**

**retrieve\_all(tree(Item,Left,Right)):- retrieve\_all(Left),**

**do\_something\_to(Item),**

**retrieve\_all(Right).**

Таким чином, використовуючи сортування по дереву, ви можете швидко впорядковувати елементи. Таке впорядкування одне з найкращих. Так, для N елементів, часова оцінка роботи алгоритму буде **О(N logN).**

### **Лексикографічне впорядкування**

В цій програмі ми будемо використовувати вмонтовані предикати Прологу для реалізації введення з клавіатури. Так, предикат READINT вводить цілі, а предикат READCHAR вводить символи. Стандартний предикат EXIT зупиняє процес виконання.

**Domains**

**chartree = tree(char, chartree, chartree);**

**end**

**predicates**

**nondeterm do(chartree)**

**action(integer, chartree, chartree)**

**create\_tree(chartree, chartree)**

**insert(char, chartree, chartree)**

**write\_tree(chartree)**

**nondeterm repeat**

**goal do(end).**

**clauses**

**do(Tree):-**

**repeat,**

**write("Enter 1 to create a tree\n"),**

**write("Enter 2 to show tree\n"),**

**write("Enter 7 to exit\n"),**

**readint(X),**

**action(X, Tree, NewTree),**

**do(NewTree).**

**action(1, Tree, NewTree):-**

**write("Enter characters or . to end: "),**

**create\_Tree(Tree, NewTree).**

**action(2, Tree, Tree):-**

**write\_Tree(Tree),**

**write("\nPress a key to continue\n"),**

**readchar(\_).**

**action(7, \_, end):- exit.**

**create\_Tree(Tree, NewTree):-**

**readchar(C),**

**C <> '.' , !,**

**write(C, " "),**

**insert(C, Tree, TempTree),**

**create\_Tree(TempTree,NewTree).**

**create\_Tree(Tree, Tree).**

**insert(New, end, tree(New, end, end)):- !.**

**insert(New,**

**tree(Element, Left, Right),**

**tree(Element, NewLeft, Right)):-**

**New < Element, !,**

**insert(New, Left, NewLeft).**

**insert(New, tree(Element, Left, Right),**

**tree(Element, Left, NewRight)):-**

**insert(New, Right, NewRight).**

**write\_Tree(end).**

**write\_Tree(tree(Item, Left, Right)):-**

**write\_Tree(Left),**

**write(Item, " "),**

**write\_Tree(Right).**

**repeat.**

**repeat:-repeat.**

## 2.11. Списки

Список – це структура даних, яка може зберігати довільне число однотипних елементів. Наприклад:

**[ ]**

**[1, 2, 3, 4, 5]**

**[tom, ann, john, bob]**

Першим представлено пустий список. Непустий список складається із двох елементів – голови і хвоста. Голова – перший елемент списку, інша частина списку – хвіст. Наприклад, в останньому списку **tom** – це голова, **[ann, john, bob]** – це хвіст списку. Список можна явно розділити на голову і хвіст за допомогою вертикальної риски:

**[tom | [ann, john, bob]]**

Насправді вертикальна риска має більш загальний смисл, нею можна виділити довільне число елементів: **[tom, ann | [john, bob]], [tom, ann, john, bob] | [ ]].**

Списки, як і всі структурні об’єкти в Пролозі це дерева. Ось як можна зобразити третій список:

**tom**

**ann**

**john**

**bob**

**[ ]**

**Рис.8. Зображення списку у вигляді дерева**

Списки зручно використовувати коли кількість компонентів у структурі змінна. Наприклад, якщо ми хочемо зробити базу даних, в якій зберігаються прізвища викладачів і предмети, які вони читають, її можна реалізувати таким чином:

**domains**

**teacher = string**

**courses\_list = string\***

**predicates**

**info(teacher, courses\_list)**

**clauses**

**info(“Шевченко О.В.”, [“Інформатика”, ”Чисельні методи”]).**

**info(“Нікольський А.С.”, [“Комп’ютерна графіка”]).**

**info(“Рябчук М.В.”, [“Фізика”, ”Хімія”, ”Астрономія”]).**

**goal**

**info(“Рябчук М.В.”, X), write (X), nl.**

Як видно, кількість предметів може варіюватись від нуля до довільного числа. Списки описуються за допомогою знаку \* (зірочка).

Основні операції зі списками такі:

* перевірка об’єкта на приналежність списку.
* конкатенація (зчеплення) двох списків.
* додавання або вилучення об’єкта зі списку.

Операції зі списками програмуються з використанням рекурсії. В процесі обробки списку він розбивається на голову і хвіст. Процедура, як правило, складається із двох правил – у першому описується окремий випадок, у другому – більш загальний випадок.

Приклади.

Перевірка об’єкта на приналежність списку реалізується виходячи із наступних міркувань:

1. об’єкт є головою списку, або
2. об’єкт належить списку, якщо об’єкт належить хвосту списку.

Ці речення на Пролозі записуються так:

**member(X, [X | Tail]).**

**member(X, [Head | Tail]):-**

**member(X, Tail).**

Конкатенація (зчеплення) двох списків, реалізується у вигляді предикату із трьома аргументами: перший і другий аргументи – два вхідні списки, третій аргумент – вихідний список. Наступні фрази показують як зчепити два списки:

1. Якщо перший аргумент пустий, то другий і третій аргументи являють собою один і той же список.
2. Якщо перший аргумент не пустий, то він має голову і хвіст, а результатом зчеплення буде список, який складається із голови першого списку і із списку, який є конкатенацією хвоста першого списку і всього другого списку.

На Пролозі ці фрази записуються таким чином:

**append([], List, List).**

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

**append(L1, List2, L3).**

Додати об’єкт до списку можна без будь-якої процедури, просто записуючи його у вигляді голови списку: **[X | List]**. Якщо ж треба явно описати процедуру додавання об’єкта до списку, її можна записати у вигляді факту:

**add(X, List, [X | List]).**

Вилучення об’єкту зі списку можна описати як предикат, що має три аргументи – перший аргумент – сам об’єкт, другий аргумент – початковий список, третій аргумент – результуючий список. Відношення вилучення записується двома реченнями:

1. Якщо Х є головою списку, тоді результатом вилучення буде хвіст цього списку.
2. Якщо Х знаходиться в хвості списку, тоді його треба вилучити звідти.

**remove(X, [X | Tail], Tail).**

**remove(X, [Y | Tail], [Y | Tail1]):-**

**remove(X, Tail, Tail1).**

Пролог має і вбудований предикат для роботи зі списками – **findall**. За допомогою нього можна зібрати всі розв’язки цілі в один список. Загальна форма:

**findall( ArgumentName, PredicateCall, ValuesList)**

**ArgumentName -** визначає, який параметр у зазначеному предикаті PredicateCall повинний бути зібраний у список ValuesList.

**PredicateCall -** вказує предикат, з якого будуть зібрані значення.

**ValuesList –** вихіднийсписок значень, зібраних за допомогою бектрекінга.

У наступній програмі за допомогою **findall** із бази даних збираются у список **TownList1** назви всіх міст, а у список **TownList2** назви міст до яких існує дорога з Київа.

**domains**

**town = symbol**

**town\_list=town\***

**facts**

**town(town).**

**predicates**

**nondeterm road(town,town)**

**clauses**

**town(kyiv).**

**town(lviv).**

**town(dnipro).**

**town(donetsk).**

**road(kyiv,lviv).**

**road(lviv,krakiv).**

**road(kyiv,dnipro).**

**road(dnipro,donetsk).**

**goal**

**findall(X, town(X), TownList1), write(TownList1), nl,**

**findall(X, road(kyiv,X), TownList2), write(TownList2), nl.**

## 2.12. Ігри

### **Ігри двох осіб з повною інформацією**

Прикладами таких ігор можуть бути шахи, шашки і т. п. У грі беруть участь два гравця, які ходять по черзі, причому обидва вони мають повну інформацію про поточну ігрову ситуацію (на відміну від більшості карточних ігор). Гра вважається закінченою, якщо досягнута позиція, яка є згідно з правилами гри «термінальною» (кінцевою), наприклад матова позиція в шахах. Правилами гри також визначається, який результат гри у цій термінальній позиції.

Для ігор такого роду можливе представлення у вигляді дерева гри (або ігрового дерева). Вершини цього дерева відповідають ситуаціям, а дуги ходам. Початкова ситуація гри – це коренева вершина; листками дерева представлені термінальні позиції.

У більшості ігор цього типу можливі наступні результати: виграш, програш і нічия. Розглянемо ігри, які мають тільки два результати – виграш і програш. Ігри, в яких можлива нічия, можна спрощено вважати іграми з двома результатами – виграш і не-виграш. Двох учасників гри будемо називати «гравцем» і «противником». Гравець може виграти в деякій не термінальній позиції з ходом гравця («позиції гравця»), якщо в ній існує який-небудь дозволений хід, який приводить до виграшу. З іншого боку деяка нетермінальна позиція з ходом противника («позиція противника») є виграною для гравця, якщо всі дозволені ходи із даної позиції ведуть до позицій, в яких можливий виграш. Ці правила є у повній відповідності з представленням задач у формі **І/АБО**-дерева. Між поняттями, які відносяться до **І/АБО**-дерева, і поняттями, які використовуються в іграх можна установити взаємну відповідність наступним чином:

Позиція гри – вершини, задачі

Термінальні позиції виграшу – цільові вершини, тривіальні задачі

Термінальні позиції програшу – задачі, які не мають розв'язання

Виграні позиції – задачі, які мають рішення

Позиції гравця – **АБО**-вершини

Позиції противника – **І**-вершини

Очевидно, що аналогічним чином поняття, які відносяться до пошуку в **І/АБО**-деревах, можна переосмислити в термінах пошуку в ігрових деревах.

Нижче приводиться проста програма, яка визначає, чи є деяка позиція гравця виграною.

**database**

**turn(symbol, symbol)**

**term\_v(symbol)**

**term\_p(symbol)**

**predicates**

**nondeterm vikt(symbol)**

**nondeterm turn\_not\_vikt(symbol, symbol)**

**clauses**

**vikt(P):-**

**term\_v(P).**

**vikt(P):-**

**not(term\_p(P)),**

**turn(P,P1),**

**not(term\_p(P1)),**

**not(turn\_not\_vikt(P1,\_)).**

**turn\_not\_vikt(P1,P2):-**

**turn(P1,P2), not(vikt(P2)).**

**turn(a, b).**

**turn(a, c).**

**turn(b, d).**

**turn(b, e).**

**turn(c, f).**

**turn(c, g).**

**term\_v(d).**

**term\_v(e).**

**term\_p(f).**

**term\_p(g).**

**goal**

**vikt(a).**

**a**

**b**

**c**

**d**

**c**

**e**

**f**

виграні терминальні позиції

**програні терминальні позиції**

**Рис. 9 Просте дерево гри**

Дерево гри представлене на рис. 9. Програма перевіряє,чи виграна позиція **а**. В даному випадку програма дасть відповідь **yes**, тобто позиція **а** виграна (як видно з малюнку існують ходи із позиції **а**, які приводять до виграних термінальних позицій **d** та **c**). Якщо запросити у системи чи виграна позиція **с – vikt(c)**, вона дасть відповідь **no** (не існує жодного дозволеного ходу із позиції **с**, який приводить до виграшу).

Тут правила гри вбудовані в предикат turn , який породжує всі дозволені ходи, а також в предикати **term\_v** і **term\_p**, які розпізнають термінальні позиції, які є, згідно з правилами гри, виграними або програними. Вираз **not(term\_p(P)** означає, що **Р** – не термінальна позиція програшу, а вираз **not(term\_p(P1))** означає, що і наступна після **Р** позиція теж не є термінальною позицією програшу. Вираз **not(turn\_not\_vikt(P1,\_))** говорить: не існує ходу противника, який веде до не виграної позиції. Іншими словами: всі ходи противника приводять до позицій, виграних с точки зору гравця.

Так як і аналогічна програма пошуку в **І/АБО**-графах, наведена вище програма використовує стратегію пошуку в глибину. Крім того, в ній не виключається можливість зациклювання в одних і тих же позиціях. Намагання виправити цей недолік може призвести до ускладнень, оскільки правила деяких ігор допускають таке повторення позицій. Правда, дозвіл повторяти позиції часто носить умовний характер, наприклад, за шаховими правилами після трикратного повторення позиції може бути оголошена нічия.

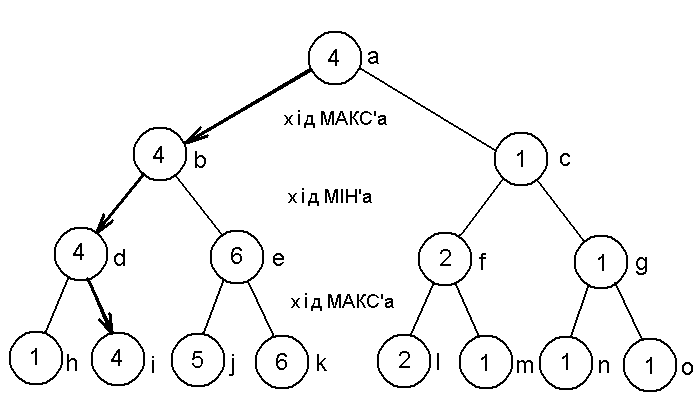
Вищенаведена програма демонструє основні принципи програмування ігор. Але практично прийнятна реалізація таких складних ігор, як шахи або го, змусила би до залучення значно більш потужних методів. Величезна комбінаторна складність таких ігор робить простий переборний алгоритм, який проглядає дерево аж до термінальних ігрових позицій, абсолютно неприйнятним. Наприклад, якщо в кожній шаховій позиції існує приблизно 30 дозволених ходів і термінальні позиції розміщені на глибині 40 ходів, то простір пошуку шахової партії має астрономічнв розміри – близько 10120 позицій (один хід (два напівходи) - 30×30≈1000 позицій, 40 ходів – 100040 позицій).

### **Мінімаксний принцип**

Для ігор, які представляють інтерес, повний перегляд ігрового дерева неможливий, тому були розроблені інші методи, які передбачають перегляд тільки частини дерева гри. Серед цих методів існує стандартний метод пошуку, який використовується в ігрових програмах і в основі якого лежить мінімаксний принцип. Дерево гри проглядається тільки до деякої глибини (звичайно на декілька ходів), а потім для всіх кінцевих вершин дерева пошуку обчислюються оцінки за допомогою деякої оціночної функції. Ідея полягає у тому, щоб, отримавши оцінки цих термінальних пошукових вершин, не просуватися далі і отримати таким чином економію часу. Далі, оцінки термінальних позицій розповсяджуються вгору по дереву пошуку у відповідності з мінімаксним принципом. В результаті всі вершини дерева пошуку одержують свої оцінки. І нарешті, ігрова програма, яка бере участь у деякій реальній грі, робить свій хід – хід, який веде із вихідної (кореневої) позиції до найбільш перспективного (с точки зору оцінки) її наступника.

Необхідно наголосити, що є різниця між “деревом гри” і “деревом пошуку”. Дерево пошуку – це тільки частина дерева гри (її верхня частина), тобто та її частина, яка була породжена у процесі пошуку. Таким чином, термінальні позиції зовсім не обов’язково повинні співпадати з термінальними позиціями самої гри.

Дуже багато залежить від оціночної функції, яка для більшості ігор, які представляють інтерес, є наближеною евристичною оцінкою шансів на виграш одного з учасників гри. Чим вища оцінка, тим більше у нього шансів виграти і чим нижча оцінка, тим більше шансів на виграш у його противника. Оскільки один із учасників гри завжди пргне до високих оцінок, а інший – до низьких, дамо їм імена **МАКС** і **МІН** відповідно. **МАКС** завжди обирає хід з максимальною оцінкою; на противагу йому **МІН** завжди обирає хід з мінімальною оцінкою. Користуючись цим принципом (мінімаксним принципом) і знаючи значення оцінок для усіх вершин “підніжжя” дерева пошуку, можна визначити оцінки всіх інших вершин дерева. На рис. 10 показано як це робиться. На цьому малюнку видно, рівні позицій з ходом **МАКС**’а чергуються з рівнями позицій з ходом **МІН**’а. Оцінки вершин нижнього рівня визначаються за допомогою оціночної функції. Оцінки всіх внутрішніх вершин можна визначити, рухаючись знизу вгору від рівня до рівня, поки ми не досягнемо кореневої вершини. У результаті, як видно із малюнка, оцінка корня виявляється рівною 4, і , відповідо, найкращим ходом **МАКС**’а із позиції **a – a-b**. Найкраща відповідь **МІН**’а на цей хід – **b-d**, і т. д. Цю послідовність ходів називають також основним варіантом. Основний варіант показує, яка “мінімаксно-оптимальна ” гра для обох учасників. Необхідно наголосити, що оцінки всіх позицій, які входять в основний варіант, співпадають.



**Рис. 10 . Статичні (нижній рівень) і мінімаксні робочі оцінки вершин дерева пошуку. Виділені ходи утворюють основний варіант, тобто мінімаксно-оптимальну оцінку з обох сторін.**

Розрізняють два види оцінок: оцінки вершин нижнього рівня і оцінки внутрішніх вершин (робочі оцінки). Перші із них називаються також “статичними”, так як вони обчислюються за допомогою “статичної” оціночної функції., на відміну від робочих оцінок , які одержуються “динамічно” при розповсюдженні статичних оцінок вгору по дереву.

Правила розповсюдження оцінок можна сформулювати наступним чином. Будемо позначати статичну оцінку позиції **P** через **v(P)**, а її робочу оцінку - через **V(P).** Нехай **P1,…, Pn** – дозволені наступники позиції **P.** Тоді співвідношення між між статичними і робочими оцінками можна записати таким чином:



Якщо **P** – термінальна позиція дерева пошуку (n=0).



якщо **P** – позіція з ходом **МАКС’**а



якщо **P** – позиція з ходом **МІН**’а.

Далі приводиться програма на **Visual Prolog**, яка обчислює мінімаксну робочу оцінку для деякої заданої позиції (у даному випадку позиції a для дерева гри, зображеного на рис. .10.).

**domains**

**position = symbol**

**position\_list = symbol\***

**mark = integer**

**database - data**

**turns(position,position\_list) %задає дозволені ходи ігри**

**max\_turn(position) %позиції, в які робить ходи ігрок**

**min\_turn(position) %позиції, в які робить ходи противник**

**stat\_o(position,mark) %статичні оцінки для термінальних позицій**

**predicates**

**nondeterm minimax(position,position,mark)**

**%обчислює найкращий хід і мінімаксну оцінку для певної позиції**

**nondeterm best(position\_list,position,mark)**

**%обирає із списку позицій - кандидатів найкращу позицію.**

**% mark - оцінка найкращої позиції**

**choice(position,mark,position,mark,position,mark)**

**% обирає або першу, або другу позицію**

**% в залежності від ходу ігрока або противника**

**clauses**

**minimax(Pos,BestPos,O):-**

**turns(Pos, PosList),!, %PosList - список дозволених ходів**

**best(PosList, BestPos, O);**

**stat\_o(Pos,O). %Pos - не має продовжень**

**best([Pos],Pos,O):-**

**minimax(Pos,\_,O),!.**

**best([Pos1|PosList],BestPos,BestO):-**

**minimax(Pos1,\_,O1),**

**best(PosList,Pos2,O2),**

**choice(Pos1,O1,Pos2,O2,BestPos,BestO).**

**choice(Pos0,O0,Pos1,O1,Pos0,O0):-**

**min\_turn(Pos0), O0<O1,!;**

**max\_turn(Pos0), O0>O1,!.**

**choice(Pos0,O0,Pos1,O1,Pos1,O1).**

**turns(a, [b,c]).**

**turns(b, [d,e]).**

**turns(c, [f,g]).**

**turns(d, [h,i]).**

**turns(e, [j,k]).**

**turns(f, [l,m]).**

**turns(g, [n,o]).**

**max\_turn(b).**

**max\_turn(c).**

**max\_turn(h).**

**max\_turn(i).**

**max\_turn(j).**

**max\_turn(k).**

**max\_turn(l).**

**max\_turn(m).**

**max\_turn(n).**

**max\_turn(o).**

**min\_turn(d).**

**min\_turn(e).**

**min\_turn(f).**

**min\_turn(g).**

**stat\_o(h,1).**

**stat\_o(i,4).**

**stat\_o(j,5).**

**stat\_o(k,6).**

**stat\_o(l,2).**

**stat\_o(m,1).**

**stat\_o(n,1).**

**stat\_o(o,1).**

**goal**

**minimax(a, X, M).**

Дана програма на запит **minimax(a, X, M),** як і слід чекати відповідає **X=b, M=4**, тобто найкращий хід із позиції **а** є **a-b**, а оцінка цього ходу і всієї позиції **а** - 4.

Основне відношення цієї програми –

**minimax(Pos, BestPos, O)**

де **O** – мінімаксна оцінка позиції **Pos**, а **BestPos** – найкраща позиція-наступник позиції **Pos** (найкращий хід, який дозволяє досягнути оцінки **О**). Відношення

**turns(Pos, PosList)**

задає дозволені ходи гри: **PosList** – це список дозволених позицій-наступників позиції **Pos**. Передбачається, що ціль **turns** має неуспіх, якщо **Pos** є термінальною пошуковою позицією (листком дерева пошуку). Відношення

**best(PosList, BestPos, O)**

вибирає із списку позицій-кандидатів **PosList** найкращу позицію **BestPos**. **О** – оцінка позиції **BestPos** і відповідно, позиції **Pos**. Під найкращою оцінкою розуміється або максимальна, або мінімільна оцінка, в залежності від того, з якого боку очікується хід.

## 2.13. Реалізація деяких методів пошуку в просторі станів на Пролозі

Простір станів на Пролозі можна представити у вигляді відношення **after(x, y).** Наприклад, якщо простір станів зображується у вигляді такого дерева:

**a**

**b**

**d**

**e**

**c**

**Рис. 11. Приклад дерева**

то його записують таким чином:

**after(a,b).**

**after(b,c).**

**after(a,d).**

**after(d,e).**

Пошук у глибину реалізується на Пролозі за допомогою наступної ідеї:

Для того, щоб знайти шлях **Solution** із даної вершини **A** в деякі цільову вершину, необхідно:

* якщо **A** – це цільова вершина, то установити **Solution = A**, або
* якщо для вихідної вершини **A** існує вершина спадкоємець **A1**, така, що можна провести шлях **Solution1** із **A1** в цільову вершину, то установити **Solution = [A | Solution1].**

На Пролозі це правило записується так:

**solve(A, [A]) :-**

**destination(A).**

**solve(A, [A|Solution]) :-**

**after(A, A1),**

**solve(A1, Solution).**

Дана програма має недолік – вона може зациклитися якщо в просторі станів є цикли. Для її удосконалення треба добавити механізм виявлення циклів.

Пошук в ширину програмується важче, ніж пошук в глибину, тому що доводиться зберігати всю множину вершин-кандидатів, а не тільки одну. Крім того, якщо нам треба отримати розв’язувальний шлях, то однієї множини вершин не достатньо. Тому будемо зберігати не множину вершин-кандидатів, а множину шляхів-кандидатів. Простір станів у цьому випадку зручно представляти за допомогою відношення **children(x, [y, z]).** Наприклад, дерево на рис. 12

**a**

**b**

**e**

**f**

**c**

**d**

**Рис. 12. Приклад дерева**

записується таким чином:

**children(a, [b, e]).**

**children(b, [c, d]).**

**children(e, [f]).**

Будемо шукати відношення пошуку в ширину **breadth\_star(Roads, Answer).** Це відношення буде істинним, якщо існує шлях із множини кандидатів **Roads**, який може бути продовжено до цільової вершини. Цей продовжений шлях і є **Answer.**

В нашій реалізації множина шляхів-кандидатів буде представлена списком шляхів, а кожен шлях – списком вершин, перечислених у зворотньому напрямі, тобто головою списку буде остання із породжених вершин. Пошук починається із одноелементногї множини кандидатів **[[Start]].**

Загальні принципи пошуку в ширину такі:

Для того, щоб виконати пошук в ширину при заданній множині шляхів-кандидатів, необхідно:

* якщо голова першого шляху – це цільова вершина, то взяти цей шлях як розв’язок, інакше
* видалити перший шлях із множини кандидатів і породити множину всіх можливих продовжень цього шляху на один крок; множину продовжень добавити в кінець множини кандидатів, а потім виконати пошук в ширину з отриманою новою множиною.

Наступна програма реалізує цей алгоритм:

**breadth\_first(Start,Answer):-**

**breadth\_star( [[Start]], Answer).**

**breadth\_star( [[H|Road]|\_], [H|Road] ):-**

**destination(H).**

**breadth\_star( [[H|T]|Roads], Answer ) :-**

**children(H, Children), make\_road(Children, [H|T], Child\_Roads),**

**append1( Roads, Child\_Roads, New\_Roads ),!,**

**breadth\_star( New\_Roads, Answer ).**

**breadth\_star(Roads, Answer). % випадок, якщо у Н немає спадкоємця**

Множина продовжень шляху породжується двома відношеннями **children(H, Children) і make\_road(Children, [H|T], Child\_Roads),** перше відношення спочатку породжує список вершин-спадкоємців **Children** вершини **H**, а друге – список шляхів-спадкоємців **Child\_Roads**, які ведуть із вершини **H.** Предикат **make\_road** можна реалізувати так:

**make\_road([], \_, []).**

**make\_road([H|T], L, [New\_L|T1]):-**

**append([H], L, New\_L),**

**make\_road(T, L,T1).**

Предикат **append1( Roads, Child\_Roads, New\_Roads )** додає множину шляхів-продовжень в кінець множини шляхів-кандидатів:

**append1([],L,L).**

**append1([X|L1],L2,[X|L3]):-**

**append1(L1,L2,L3).**

Якщо у вершини **H** немає вершин-спадкоємців і відношення **children(H, Children)** отримує невдачу забезпечується альтернативний запуск процедури **breadth\_star(Roads, Answer).**

### **Завдання 3.**

Розв’язати задачу про комівояжера, використовуючи такі методи пошуку:

1. Метод перебору.
2. Метод пошуку в глибину.
3. Метод пошуку в ширину.
4. Метод найкоротшого шляху.
5. Метод Мура.
6. Метод Дейкстри.
7. Метод гілок і меж.
8. Метод Нільсона.

Розв’язати задачу про туриста, використовуючи такі методи пошуку:

1. Метод перебору.
2. Метод пошуку в глибину.
3. Метод пошуку в ширину.
4. Метод найкоротшого шляху.
5. Метод Мура.
6. Метод Дейкстри.
7. Метод гілок і меж.
8. Метод Нільсона.

# 3. Короткі відомості про засоби програмування інтерфейсів (VPI) Visual Prolog 5.2

Одним із секретів потужності й елегантності програмування в Visual Prolog є використання Visual Programming Interface (бібліотека підтримки програмування інтерфейсів користувача), чи VPI. З використанням VPI створюються додатки, які керуються подіями (event-driven applications). VPI - основа для програмування інтерфейсів користувача, що була спеціально передбачена як засіб програмування GUI (графічних інтерфейсів користувача) для Прологу. VPI у даний час підтриманий у Windows NT і OS/2 платформах.

## 3.1. Створення найпростішого додатку

Розглянемо використання VPI на прикладі створення найпростішого додатку з використанням стандартного діалогового вікна. Додаток буде мати меню, при виборі відповідного пункту меню з’явиться стандартне діалогове вікно з повідомленням “Hello World!”.

Для створення нового проекту скористаємося експертом додатків, виконуючи у вікні інтегровного середовища розробки команду : **Project | New** **Project**. У вікні **Application Expert** на вкладці **General** вводимо назву проекту, наприклад, Hello\_World. Можна вибрати також і основний каталог для проекту, або згодитись із запропонованим за умовчуванням. Всі опції на інших вкладках вікна залишаємо без змін і натискаємо кнопку **Create**. З’являється вікно проекту, у якому відображені два вихідних модулі – **Hello\_World.pro i VPITools.pro**. Після створення проекту, натискуючи кнопку **Run,** компілюємо, компонуємо і виконуємо проект. Повинне з’явитися вікно нашого проекту з назвою **Hello\_World** , у якому є меню, панель інструментів і вікно **Messages**. Цей, створений за умовчуванням проект, є каркасом для формування нашого власного додатку. Для його формування потрібно виконати три етапи.

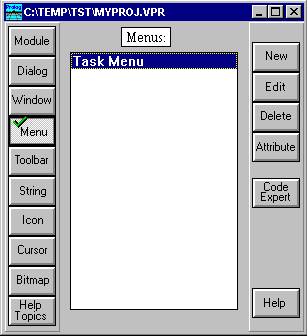
- додати новий пункт меню

- додати речення для реагування на вибір цього пункту меню

- у цьому реченні вставити предикат **dlg\_Note**, який виводить на екран вікно MessageBox (реакція на вибір пункту меню).

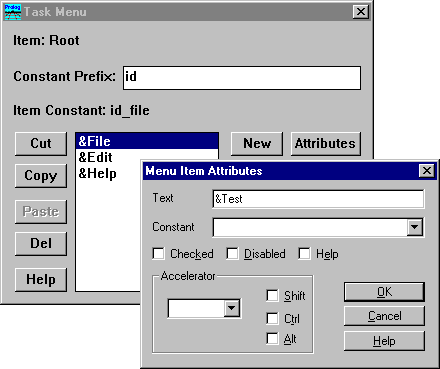
### **Додавання пункту меню**

У вікні Проекту натисніть кнопку Menu в лівій панелі інструментів (рис.13).



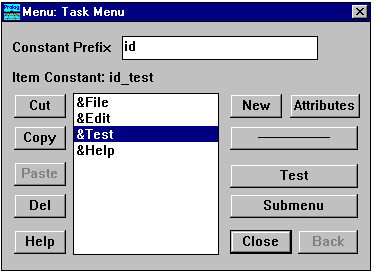
**Рис.13.** Вікно Проекту

Щоб активізувати редактор меню зробіть подвійне клацання мишкою Task Menu.



**Рис.14. Додавання пункту меню Test**

Виділіть пункт меню Edit і додайте новий пункт меню верхнього рівня з ім’ям Test, натискуючи кнопку New (рис.14). Константа id\_test додається автоматично. Якщо бажаєте визначити акселератор перед ім'ям пункту меню уведіть символ & і виберіть будь-яку клавішу в полі Accelerator , а також поставте прапорець перед Shift, Ctrl, або Alt.



**Рис.15. Додавання підменю Hello World в пункт меню Test**

Тепер створимо підменю пункту меню Test. Для цього виділимо пункт меню Test і натиснемо кнопку Submenu. Далі у вікні, що з'явилося, натискаємо кнопку New. У поле Text уводимо Hello World, а в поле Constant не погоджуємося із запропонованою константою, а уводимо свою константу id\_hello. і натискаємо ОК.

Можна перевірити створене меню, для чого натискаємо кнопку Test. Потім знову натискаємо Test і закриваємо редактор меню, натискаючи кнопку Close.

### **Додавання речення для реагування**

### **на вибір пункту меню**

Для цього можна було б використовувати редактор тексту VPI і додати потрібний фрагмент коду. Але краще й зручніше це зробити за допомогою так званих експертів коду (Code Experts). Для цього натисніть у вікні проекту кнопку Window на лівій інструментальній панелі й кнопку Code Expert на правій інструментальній панелі. З'явиться один із видів експертів коду - експерт діалогів і вікон (Dialog/Window Expert). Потім виконайте наступні кроки:

- виберіть у вікні Event Type пункт Menu.

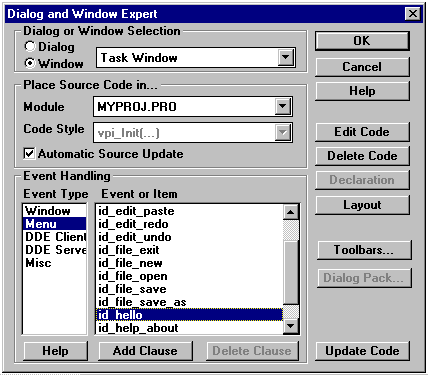
- виберіть у вікні Event or Item пункт id\_hello.

- натисніть кнопку Add Clause.

- натисніть кнопку Edit Clause.

Select Menu Event Type

**Select id\_hello**



**Рис. 16. Вікно Експерта Коду**

Вставка предикату dlg\_Note в речення.

Натиснення кнопки Edit Clause видасть редактор, в якому уже додано таке речення:

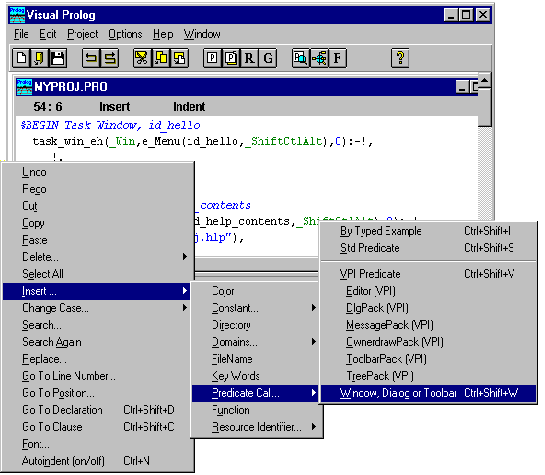
**task\_win\_eh(\_Win,e\_Menu(id\_hello,\_ShiftCtlAlt),0):-!,**

**!.**

**task\_win\_eh** – це предикат оброблювача події для вікна **Task**.

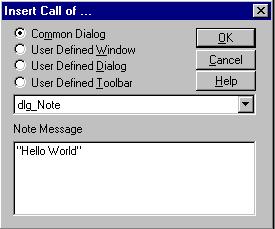
Зверху й знизу даного речення знаходяться коментарії **%BEGIN Task Window, id\_hello** , які теж додані експертом діалогів і вікон. Змінна **\_Win** містить дескриптор вікна – у даному випадку вікна **Task.**Змінна **\_ShiftCtlAlt** містить комбінацію клавіш, якими ми можемо викликати меню.

Тепер для вставки предикату **dlg\_Note** скористаємося контексним меню. Встановимо курсор у речення після першого предикату **!,** натискаємо праву клавішу миші й вибираємо в меню пункти **Insert | Predicate Call | Window, Dialog or Toolbar.**



**Рис. 17. Додавання предикату dlg\_Note через контекстне меню**

Далі у вікні **Insert Call of…**, вибираємо **Common Dialog** та **dlg\_Note**, а також у полі вводу **Note Message** друкуємо фразу “Hello World!”, включаючи лапки.



**Рис. 18. Вікно Insert Call of…**

Тепер кінцевий код речення виглядатиме таким чином:

**task\_win\_eh(\_Win,e\_Menu(id\_hello,\_ShiftCtlAlt),0):-!,**

**Title="Title",**

**dlg\_Note(Title,"Hello World!"),!.**

Перевіряємо роботу додатку. Натискаючи кнопку **Run** на панелі інструментів компілюємо, компонуємо і виконуємо проект. Тепер, якщо в меню нашого додатку вибрати команду **Test | Hello World**, з’явиться вікно MessageBox із повідомленням Hello World!.

### **Використання діалогових вікон, створених користувачем**

Приклад показує як створювати діалогові вікна (діалоги) і використовувати елементи управління діалогу. Для прикладу, створимо діалог для знаходження косинусу числа.

Для створення нового діалогу треба у лівій частині вікна проекту вибрати кнопку **Dialog**, а в правій частині вікна натиснути кнопку **New**. Відкривається вікно **Dialog Attributes**, в якому у відповідні рядки вводимо ім`я діалогу, наприклад My, константу (вона вводиться автоматично), заголовок, якщо потрібно прапори стилю, розміри та тип (модальний або немодальний). Після натиснення **ОК**, з`являється вікно діалогу і вікна елементів управління та шарів. Редагуємо вікно діалогу, наприклад, розміщуємо у ньому рядок вводу та кнопку. Після редагування вікна, виділяємо його назву у вікні проекту і натискуємо кнопку **Code Expert**. У вікні **Dialog and Window Expert** у групі **Dialog or Window Selection** перемикач поставити на **Dialog**, у списку знайти ім`я діалогу - **My**. У групі **Place Code Source** in ... у списку вибрати назву проекту (наприклад **New.pro**). Натиснути кнопку **Default Code**, при цьому у файл (наприклад **New.pro**) вставляється код, що стосується данного діалогу. Для виклику діалогу створимо пункт меню. У лівій частині вікна проекту вибрати кнопку **Menu**, а в правій **Edit**. У вікні **Task Menu** натиснути кнопку **New**, ввести текст, наприклад, **My\_dialog**, константа вводиться автоматично – **id\_My\_dialog**. Далі, натискаємо кнопку **Code Expert**, у групі **Dialog or Window Selection** перемикач поставити на **Window**, у списку **Event Type** вибрати **Menu**, у списку **Event or Item** вибрати ім`я нашого пункту меню - **My\_dialog** і натиснути кнопку **Add Clause і Edit Clause**. У вікні редактора у тілі правила task\_win\_eh ставимо курсор після першого предиката ! та коми і за допомогою контексного меню виконуємо команду **Insert - Predicate Call - Window, Dialog or Toolbar.**

З`являється вікно **Insert Call of ...**, у якому ставимо перемикач на **User Defined Dialog** у списку знаходимо ім`я нашого діалогу **My** і натискуємо **ОК**. Далі, викликаємо **Code Expert**, ставимо перемикач на **Dialog** у списку знаходимо ім`я **My**, в списку **Event Type** вибираємо **Control**, у списку **Event or Item** вибираємо pb: **idc\_Push\_Button**, натискуємо кнопку **Add Clause i Edit Clause.**

Далі у редакторі в тілі правила **dlg\_my\_eh** необхідно спочатку отримати дескриптор елементу управління поля вводу для доступу до його атрибутів. Для цього у контексному меню виконуємо команду **Insert - Predicate Call - VPI Predicate**, у списку вибираємо предикат **win\_GetCtlHandle** натискаємо **ОК**. **CtrlId** змінюємо за допомогою контексного меню на idc\_edit (команда **Insert - Resource Identifier - Dialog Controls**). Змінна **CtrlWin** буде зберігати дескриптор поля вводу **idc\_edit.**

Далі за допомогою предикату **win\_GetText** отримуємо текст із поля вводу **idc\_edit** (команда **Insert - Predicate Call - VPI Predicate** ), **\_Win** змінюємо на дескриптор поля вводу - **CtrlWin**. Потім вводимо послідовно за допомогою контексного меню предикат **str\_real** (команда **Insert - Predicate Call - Standart Predicate** ), арифметичний вираз і знову предикати **str\_real** та **win\_SetText** (який установлює текст у поле вводу). Код для правила **dlg\_my\_eh** буде таким:

**dlg\_my\_eh(\_Win,e\_Control(idc\_push\_button,\_CtrlType,\_CtrlWin,\_CtlInfo),0):-!, CtrlWin = win\_GetCtlHandle(\_Win, idc\_edit),**

**Text = win\_GetText(CtrlWin),**

**str\_real(Text,RealParam), Y=cos(RealParam),str\_real(TextY,Y),**

**win\_SetText(CtrlWin, TextY),**

**!.**

### **Завдання 4.**

Розробити експертну систему згідно із варіантом та темою завдання. Реалізувати функцію згідно варіанту завдання.За результатами роботи представити звіт. Структура звіту така:

* титульний лист;
* зміст;
* вступ ( коротко описати історію виникнення та розвиток експертих систем);
* постановка задачі ( представити тему та варіант даної задачі);
* теоретична частина ( описати предметну область експертої системи й механізми, засоби та функції, які в ній використовуються;обґрунтувати вибір механізму логічного виводу);
* програма;
* опис програми(особливості програми, узагальнений алгоритм програми, опис предикатів);
* висновки ( проаналізувати переваги та недоліки розробленої програми);
* список використаних джерел.

**Варіанти завдання:**

|  |  |  |
| --- | --- | --- |
| Варіант | Тема | Модуль, функції експертної системи |
| 1 | 1 | Модуль пояснення. Таблична форма діалогу. |
| 2 | 1 | Модуль нагромадження знань.Фразова форма діалогу. |
| 3 | 1 | Модуль пояснення. Фразова форма діалогу. |
| 4 | 1 | Модуль нагромадження знань.Таблична форма діалогу. |
| 5 | 2 | Модуль пояснення.Таблична форма діалогу. |
| 6 | 2 | Модуль нагромадження знань.Таблична форма діалогу. |
| 7 | 2 | Модуль пояснення.Фразова форма діалогу. |
| 8 | 2 | Модуль нагромадження знань.Фразова форма діалогу. |
| 9 | 3 | Модуль пояснення.Таблична форма діалогу. |
| 10 | 3 | Модуль нагромадження знань.Таблична форма діалогу. |
| 11 | 3 | Модуль пояснення.Фразова форма діалогу. |
| 12 | 3 | Модуль нагромадження знань.Фразова форма діалогу. |
| 13 | 4 | Модуль пояснення.Таблична форма діалогу. |
| 14 | 4 | Модуль нагромадження знань.Таблична форма діалогу. |
| 15 | 4 | Модуль пояснення. Фразова форма діалогу. |
| 16 | 4 | Модуль нагромадження знань. Фразова форма діалогу. |
| 17 | 5 | Модуль пояснення. Таблична форма діалогу. |
| 18 | 5 | Модуль нагромадження знань. Таблична форма діалогу. |
| 19 | 5 | Модуль пояснення. Фразова форма діалогу. |
| 20 | 5 | Модуль нагромадження знань. Фразова форма діалогу. |
| 21 | 6 | Модуль пояснення. Таблична форма діалогу. |
| 22 | 6 | Модуль нагромадження знань. Таблична форма діалогу. |
| 23 | 6 | Модуль пояснення. Фразова форма діалогу. |
| 24 | 6 | Модуль нагромадження знань. Фразова форма діалогу. |
| 25 | 7 | Модуль пояснення. Можливість роботи з невизначенністю. |
| 26 | 7 | Модуль нагромадження знань. Таблична форма діалогу.Можливість роботи з невизначенністю. |
| 27 | 7 | Модуль пояснення. Фразова форма діалогу.Можливість роботи з невизначенністю. |
| 28 | 7 | Модуль нагромадження знань. Фразова форма діалогу.Можливість роботи з невизначенністю. |
| 29 | 8 | Модуль пояснення. Таблична форма діалогу. |
| 30 | 8 | Модуль нагромадження знань. Таблична форма діалогу. |
| 31 | 8 | Модуль пояснення. Фразова форма діалогу. |
| 32 | 8 | Модуль нагромадження знань. Фразова форма діалогу. |
| 33 | 9 | Модуль пояснення. Таблична форма діалогу. |
| 34 | 9 | Модуль нагромадження знань. Таблична форма діалогу. |
| 35 | 9 | Модуль пояснення. Фразова форма діалогу. |
| 36 | 9 | Модуль нагромадження знань. Фразова форма діалогу. |

**Тема 1**. Консультуюча інтерактивна експертна система по визначенню оптимальної конфігурації ПЕОМ.

Основними вхідними фактами (даними) є:

Ціль, для якої призначається ПЕОМ;

Межі припустимої суми ($ - $);

Фірма (країна) виготовлювач.

**Тема 2**. Діагностична інтерактивна експертна система пошуку причини й усунення несправності кольорового телевізора LG CF-20F60K.

Основними вхідними фактами (даними) є ознаки несправності.

Використовувати наступний порядок пошуку:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Ознака несправності** | | | | | | | | | **Вид перевірки чи регулювання** |
| **немає зображення, немає звуку** | **звук нормальний, зображення погане** | **зображення нормальне, звук поганий** | **нечітке зображення** | **лінії або смужки на зображенні** | **поганий прийом деяких каналів** | **чорно-біле изображення** | **погана передача кольору** | **не працює ПДУ** |
|  |  |  |  |  |  |  |  |  | **Чи уткнута** **вилка шнура живлення в розетку?** |
|  |  |  |  |  |  |  |  |  | **Чи включене живлення телевізора?** |
|  |  |  |  |  |  |  |  |  | **Спробуйте включити інший канал (слабкий сигнал)** |
|  |  |  |  |  |  |  |  |  | **Перевірте антену ( чи підключена антена до антенного гнізда телевізора?)** |
|  |  |  |  |  |  |  |  |  | **Перевірте антену ( чи обірваний провід?)** |
|  |  |  |  |  |  |  |  |  | **Перевірте антену** |
|  |  |  |  |  |  |  |  |  | **Немає чи поблизу джерела перешкод?** |
|  |  |  |  |  |  |  |  |  | **Відрегулюйте контраст** |
|  |  |  |  |  |  |  |  |  | **Відрегулюйте яскравість** |
|  |  |  |  |  |  |  |  |  | **Відрегулюйте насиченість кольору** |
|  |  |  |  |  |  |  |  |  | **Відрегулюйте голосність** |
|  |  |  |  |  |  |  |  |  | **Перевірте батарейки в ПДУ** |
|  |  |  |  |  |  |  |  |  | **Перевірте гнізда Audio/Video (тільки при роботі з відеомагнітофоном)** |

**Тема 3**. Консультуюча експертна система для вибору породи собаки.

Основними вхідними фактами (даними) є:

вовна (довга, коротка);

офарблення (чорна, біла і т.п.);

ріст;

хвіст (низькопосаджений і т.п.);

вуха (довгі, короткі і т.п.);

характер;

вага;

для яких цілей.

Наприклад,

Англійський бульдог (коротка вовна, ріст менше 22 дюймів, низькопосаджений хвіст, гарний характер і т.д.);

Гонча (коротка вовна, ріст менше 22 дюймів, довгі вуха, гарний характер і т.д.);

Дог (коротка вовна, ріст менше 30 дюймів, довгі вуха, гарний характер, вага більш 100 фунтів і т.д.);

Американська гонча (коротка вовна, низькопосаджений хвіст, довгі вуха, гарний характер і т.д.);

Коккер-спаніель (довга вовна, ріст менше 22 дюймів, низькопосаджений хвіст, довгі вуха, гарний характер і т.д.);

Ірландський сеттер (довга вовна, ріст менше 30 дюймів, довгі вуха і т.д.);

Колі (довга вовна, ріст менше 22 дюймів, , низькопосаджений хвіст, гарний характер і т.д.);

Сенбернар (довга вовна, низькопосаджений хвіст, гарний характер, вага більш 100 фунтів і т.д.) і т.д.

**Тема 4**. Медична консультуюча експертна система на вибір лікарських трав.

Основними вхідними фактами (даними) є симптоми хвороби.

Нижче приведені приклади деяких рослин:

(Ім'я рослини, показання до застосування, дія на організм)

Шувар звичайний(Acorus Calamus ).

Виразка шлунка, гастрит зі зниженою шлунковою секрецією.

Поліпшує апетит, підсилює відділення шлункового соку.

2. Аскоміцети чи сумчасті гриби (Ascomicetis).

Гіпертонія , підвищена дратівливість, спазми судин

Зупиняє внутрішню кровотечу, седативна дія

на порушення ЦНС(Центральна Нервова Система), знижує кров'яний тиск.

3. Глід відігнутий чашолистиковий(Cratequs сurvіsераlа Lindin).

Спонтанні болі в серце, початкова гіпертонія.

Знижує збудливість ЦНС, тонізує м'яз серця, підсилює кровообіг у коронарних судинах, знижує А/Д.

4. Лопух великий(Arctium lарра ).

Набряки ниркового і серцевого походження, відсутність поту, загострення геморою, дитячий діатез, злоякісні пухлини, більмо, бородавки, хронічні хвороби печінки.

Сприяє збільшенню діурезу, знімає запальні процеси, при обробці волосся - зміцнює волосяні сумки, допомагає при екземі, поліпшує функцію бруньок.

5. Ялiвець звичайний(Juniреrus communis).

Набряки ниркового походження, гіпотонія, інфекція сечових шляхів, кашель,

Здійснює сечогінну дію, дезінфекція сечових шляхів, має відхаркувальний ефект, поліпшує травлення."

6. Вільха чорна (Alnus glutinosa),вільха сіра (Volrha sehera)",

Ентероколіти, опіки, дерматити, кровотечі з носа, поноси, геморой.

Протизапальні і бактерицидні, особливо при поверхневих гнійних процесах."

7. Кульбаба лікарський (Taraxacum officinalis Wigg).

Порушення апетиту, запори, холіцистити, порушення ЦНС.

Збуджує апетит; як жовчогінний засіб, як заспокійливий засіб. робить рідким стілець.

8. Вівчарська сумка(Сарsеllа bursa- раstоrіs).

Гіпертиреоз, при капілярній кровотечі.

Підвищує згортуваність крові, знижує тиск крові, приймають при поносі, хворобах печінки.

9. Бузок звичайний (Syrtuga Vulgaris).

Невралгії, запальні процеси, виразка шлунка, кашель, пов'язки на гнійні рані, набряки ниркового походження.,

Знімає біль при місцевих запаленнях, поліпшує стан при загальних застудах, запальних процесах, поліпшує ниркові набряки, сечогінне.

10. Хрін звичайний.

Відсутність апетиту ,авітаміноз, зниження шлункової секреції, погіршення травлення, вірусний гепатит,ангіни.

Підвищує апетит, поліпшує травлення, підсилює секрецію травних залоз, жовчогінний засіб, відволікаючий засіб.

11. "Полинь гірка(Artemisia absintinum).

"Знижений апетит.

Поліпшує апетит, травлення ,жовчогінну дію, спазмолітична дія, аплікації при рентгенівських опіках, екземах, бронхіальній астмі.

12. Волоський горіх(Iuglands regia).

Гіповітаміноз C, поноси, шлункова гемороїдальна кровотеча, шкірний туберкульоз, інші шкірні захворювання.

В'яжуча дія на подразнену слизову оболонку, кровоспинна дія на ясна, геморой, виразку шлунка.

**Тема 5**. Експертна система для визначення мінерального добрива.

Основними вхідними фактами (даними) є певні властивості добрива.

Нижче поданий порядок визначення добрива.

Для визначення дибрива його досліджують за пунктами, починаючи з першого. У тому разі, коли певна властивість досліджуваного добрива збігається з властивістю, зазначеною в одному із підпунктів, наступне дослідження слід проводити за пунктом, зазначеним цифрою після цього підпункту. Якщо всі властивості досліджуваного добрива збігаються із зазначеними в підпунктах, то після останнього з них дається назва добрива.

1. а) кристали або гранули, добре розчинні у воді ........................................... ..2

б) кристали або гранули,погано розчинні у воді ..............................................11

1. а) на розжареному вугіллі спалахує...................................................................3

б) на розжареному вугіллі димить........................................................................5

в) на розжареному вугіллі не змінюється............................................................8

1. а) при нагріванні з розчином лугу утворюється амміак –

*амміачна селітра* NH4NO3

б) при нагріванні з розчином лугу аміак не утворюється............................4

1. а) забарвлює полум`я у жовтий колір –

*натрієва селітра* NaNO3

б) забарвлює полум`я у фіолетовий колір

*калійна селітра* KNO3

в) забарвлює полум`я у оранжевий колір

*кальцієва селітра* Ca(NO3)2

1. а) при нагріванні з розчином лугу утворюється амміак ...................................6

б) при нагріванні з розчином лугу амміак не утворюється.-

*карбамід* CO(NH2)2

1. а) з розчином хлориду барію утворює білий осад, нерозчинний у сильних кислотах - *сульфат амонію* (NH4)2SO4

б) з розчином хлориду барію осад не утворюється, а якщо й утворюється, то він розчинний у сильних кислотах.......................................................................7

1. а) з розчином нітрату срібла утворює білий осад, нерозчинний в азотній кислоті – *хлорид амонію* NH4Cl

б) з розчином нітрату срібла утворює жовтий осад, розчинний в азотній кислоті – *амофос* NH4H2PO4 i (NH4)2HPO4

1. а) з розчином хлориду барію утворює білий осад, розчинний у кислотах, або не утворює осаду.............................................У визначенні допущено помилку.

б) з розчином хлориду барію утворює білий осад, нерозчинний у сильних кислотах – *сульфат калію* K2SO4

**Тема 6.** Консультуюча інтерактивна експертна система,яка допомагає директору фірми при прийомі кандидата на роботу. Експертна система дозволяє визначити яку посаду у фірмі може зайняти кандидат.

Вхідними фактами (даними) є відповіді кандидата на питання, що задаються експертною системою.

Приблизне дерево рішень:

**<2**

**<10**

**>=10**

Відвідувачу відмовити

Запропонувати відвідувачу посаду інженера з експлуатації

**ні**

**Відвідувач має вчений ступінь?**

**так**

**Відвідувач зробив важливе відкриття?**

Запропонувати відвідувачу посаду інженера-конструктора

**Середній**

**бал відвідувача під час навчання**

**ні**

**так**

**Запропонувати відвідувачу посаду наукового співпрацівника**

Скільки років співробітник працював за

**фахом?**

**>=2**

**Відвідувачу відмовити**

Запропонувати відвідувачу посаду інженера з експлуатації

**Рис.19. Дерево рішень консультуючої експертної системи**

**Тема 7.** Консультуюча експертна система прогнозу повені і необхідності евакуації міста N.

Основними вхідними фактами (даними) є фактори погоди:

Рівень води.

Дощ в околицях міста і вище по ріці

Температура. Якщо температура висока і з гір у ріку розтануло багато снігу, а рівень води в річці високий, є небезпека повені.

Сніг. У розрахунок приймається кількість снігу в горах.

**Тема 8**. Діагностична медична експертна система.

Основними вхідними фактами (даними) є відповіді пацієнта на питання, що задаються експертною системою.

Пацієнт повинний виразити ступінь згоди на нижче перераховані твердження.

'У загальному я нервовий';

'Я дуже турбуюся про свою роботу';

'Я часто відчуваю нервову напругу';

'Моя повсякденна діяльність викликає велику напругу';

'Спілкуючись з людьми, я часто відчуваю нервову напругу';

'До кінця дня я зовсім виснажений фізично і психічно';

Ступінь згоди може бути виражена одним з наступних чотирьох варіантів:

1. "ТАК, ЗГОДНИЙ"

2. "СКОРІШЕ, ЗГОДНИЙ"

3. "СКОРІШЕ, НЕ ЗГОДНИЙ"

4. "НІ, НЕ ЗГОДНИЙ"

Кожен варіант відповіді має своя вагу відповідно до його порядкового номера, тобто дорівнює йому.

Система може видати два рішення:

1: Підвищена психоемоційна напруга.

2: Психоемоційна напруга в нормі.

Рішення видається на основі середньої ваги в такий спосіб:

якщо пацієнт чоловік і (A1+A2+A3+A4+A5+A6)/6<=2,

то 1 варіант рішення;

якщо пацієнт жінка і (A1+A2+A3+A4+A5+A6)/6<=1.83,

то 1 варіант рішення.

В всіх інших випадках видається другий варіант рішення.

# Список рекомендованої літератури.

Братко И. Программирование на языке Пролог для искусственного интеллекта. – М.: Мир, 1990 – 560 с.

Глибовець М.М., Олецький О.В. Штучний інтелект: підручник.-К.: Видавничий дім "КМ Академія", 2002. – 366 с.: іл...

Дж. Малпас. Реляционный язык Пролог и его применение. М.: Мир, 1990 – 430 с.

Клоксин У., Меллиш К. Программирование на языке Пролог. 334 с.

Левин Р., Дранг Д., Эделсон Б. Практическое введение в технологию искусственного интеллекта и экспертных систем с иллюстрациями на Бейсике. М.: Финансы и статистика, 1991 – 239 с.

Макаллистер Дж. Искусственный интеллект и Пролог на микроЭВМ. – М.: Машиностроение, 1990 – 240 с.

Пугачев Е.К. Язык программирования Prolog применительно к системам искусственного интеллекта: Методические указания по выполнению лабораторных работ по дисциплине "Системы искусственного интеллекта” Часть 1, 2 - Московский государственный технический университет имени Н.Э. Баумана, 2000 – 61 c.

**Зміст**

1. Теоретична частина 3

1.1. Методи розв’язання задач 3

Розв’язання задач методом пошуку в просторі станів 4

Загальна схема алгоритму Харта, Нільсона і Рафаеля 5

Розв’язання задач методом редукції 9

Розв’язання задач дедуктивного вибору 12

Розв’язання задач, що використовують немонотонні логіки, імовірнісні логіки 12

1.2. Експертні системи 13

Експертна система, заснована на правилах (прямий висновок - forward chaining) 20

Експертні системи, що базуються на логіці (зворотній висновок - backward chaining) 21

Модуль (компонент) пояснення 22

Модуль набуття знань (компонент набуття знань) 23

Етапи проектування ЕС 24

Відмінність експертних систем від традиційних програм 26

2. Основи програмування на мові Visual Prolog 28

2.1. Загальний огляд мови Пролог 28

2.2. Основні теоретичні відомості 28

Основні визначення мови Visual Prolog (далі Пролог) 28

2.3. Структура програми на мові Visual Prolog 30

2.4. Предикати введення - виведення 32

2.5. Об’єкти даних 32

Завдання 1. 33

2.6. Вбудовані механізми мови Пролог. Управління бектрекінгом 34

2.7. Організація циклів. Рекурсія 36

2.8. Використання динамічної бази даних 39

2.9. Рекомендації для створення програм на Пролозі 40

Завдання 2. 42

2.10. Рекурсивні структури даних 42

Структура даних типу дерева 42

Обходи дерева 43

Бінарний пошук на дереві 45

Сортування по дереву 46

Лексикографічне впорядкування 47

2.11. Списки 48

2.12. Ігри 51

Ігри двох осіб з повною інформацією 51

Мінімаксний принцип 54

2.13. Реалізація деяких методів пошуку в просторі станів на Пролозі 58

Завдання 3. 61

3. Короткі відомості про засоби програмування інтерфейсів (VPI) Visual Prolog 5.2 62

3.1. Створення найпростішого додатку 62

Додавання пункту меню 62

Додавання речення для реагування 64

на вибір пункту меню 64

Використання діалогових вікон, створених користувачем 67

Завдання 4. 68

Список рекомендованої літератури. 78