МІНІСТЕРСТВО ОСВІТИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ

ТЕХНІЧНИЙ УНІВЕРСИТЕТ

ПРАКТИЧНЕ ПРОГРАМУВАННЯ МОВОЮ VISUAL PROLOG

Навчальний посібник

Рекомендовано Міністерством освіти і науки України
як навчальний посібник для студентів
вищих навчальних закладів, які навчаються за спеціальністю
«Програмне забезпечення систем»

Запоріжжя

2016

ББК 32.973.26-18.1
УДК 004.4'2+004.8
П 69

Рекомендовано до друку Вченою радою
Запорізького національного технічного університету
(Протокол №9 від 29.04.2013 р.)
Гриф надано Міністерством освіти і науки України
(лист №1/11-17594 від 18.11.13 р.)

Колектив авторів:

Дейнега Лариса Юрьевна – розділ 2;
Камінська Жанна Костянтинівна – розділ 6;
Левада Ірина Васильевна – розділ 3–5;
Сердюк Сергій Микитович – розділ 1

Рецензенти:

Пазюк М. Ю. – д-р техн. наук, проф., проректор з НПР, зав. каф. АУТП Запорізької державної інженерної академії;

Лавров Є. А. – д-р техн. наук, професор комп'ютерних наук Сумського національного аграрного університету;

Гоменюк С. І. – д-р тех. наук, проф., зав. каф. математичного моделювання Запорізького національного університету, декан математичного факультету

П69 Практичне програмування мовою Visual Prolog : навч. посіб. / [Дейнега Л. Ю., Камінська Ж. К., Левада І. В., Сердюк С. М] – Запоріжжя : ЗНТУ, 2016. – 236 с.

ISBN 978-617-529-115-3

Навчальний посібник розроблювався з метою розширення знань та практичних навиків програмування мовою Visual Prolog студентів після вивчення базового курсу «Функціональне та логічне програмування». Зміст навчального посібника орієнтований на теми розрахунково-графічних завдань, що виконуються студентами. Робота над книгою також пропонується тим, хто володіє елементарними знаннями з мови Visual Prolog і хоче застосувати їх практично для розробки програмних продуктів. Реалізація програм на Пролозі часто вимагає не тільки досвіду роботи, а і творчого підходу, кмітливості.

Існує багато літератури присвяченій застосуванню Прологу для розв'язування задач штучного інтелекту, в яких подається логічне подавання компонент систем штучного інтелекту. Але немає літератури, яка б показувала застосування можливостей сучасних реалізацій Прологу до програмного створення компонент систем штучного інтелекту. Посібник пропонує студентам різні способи реалізації компонент інтелектуальних систем мовою Visual Prolog показує переваги мови перед мовами традиційного програмування для створення експертних систем.

Проте, книгу присвячено не тільки розв'язуванню задач штучного інтелекту, а і розв'язуванню інших типів задач. У книзі показано способи реалізації програм традиційного програмування засобами мови Visual Prolog.

УДК 004.4'2+004.8
ББК 32.973.26-18.1

ISBN 978-617-529-115-3

© ЗНТУ, 2016
©Дейнега Л. Ю., Камінська Ж. К.,
Левада І. В., Сердюк С. М., 2016

ЗМІСТ

ПЕРЕДМОВА

Якість професійної підготовки фахівця визначається вмінням самостійно розв'язувати виробничі та наукові задачі, готовністю до самоосвіти. Такі властивості фахівець повинен набути під час навчання у вищому навчальному закладі. Особливо це стосується випускників кафедри «Програмне забезпечення», які працюють у різних галузях науки і виробництва.

Початок нашого віку характеризується бурхливим розвитком штучного інтелекту. Сьогодні слова «експертна система» вже не викликають подиву, а звучать звично. Для спілкування людини і програмних систем стає важливим створення інтелектуальних інтерфейсів. На підприємствах впроваджують інтелектуальні системи, що керують виробничими процесами. В різних країнах проводяться змагання по робототехнічному спорту. Серед господарських приладів все частіше зустрічаються прилади з штучним інтелектом. Тобто, штучний інтелект швидко поширюються у всіх сферах діяльності людини. Тому оволодіння мовами, які є інструментами створення інтелектуальних систем, є необхідною умовою випуску грамотного фахівця, а вміння самостійно застосовувати засоби мов штучного інтелекту до реалізації компонент інтелектуальних систем важливо. Звідси з'явилась ідея створити навчальний посібник, який би розширював та поглиблював знання з базової дисципліни «Функціональне та логічне програмування».

Мова логічного програмування Visual Prolog, що вивчається у дисципліні, є мовою штучного інтелекту. У посібнику розповідається про сучасні програмні способи розв'язування задач штучного інтелекту та методи розробки компонент інтелектуальних систем, про переваги мови Prolog перед алгоритмічними мовами при розв'язуванні задач певного типу.

Кожна тема супроводжується прикладами програм мовою Visual Prolog, які ілюструють поданий матеріал. Наприкінці кожного параграфу подаються резюме та контрольні питання, щоб звернути увагу студента на важливі питання. А вправи, що розташовані після них, допоможуть набути навичок у програмуванні за темою. Тобто, посібник допомагає розв'язати одну з основних задач дисципліни –

дати студенту знання та практичні навички з мови логічного програмування Visual Prolog.

Мова Visual Prolog розвинена до сучасного рівня і постійно удосконалюється. Вона гармонійно поєднує в собі дві парадигми логічного програмування і об'єктна-орієнтованого програмування.

Мова забезпечує розробку *клієнт-серверних застосувань* для Windows NT, Linux, Unix, підтримує основні протоколи TCP/IP, FTP, HTTP мережевого обміну. Visual Prolog дозволяє створювати сервери баз даних або логічні сервери. Програми на Візуальному Пролозі можуть пов'язуватися із зовнішніми базами даних за SQL – інтерфейсом, який базується на бібліотеках ODBC та OCI баз даних Oracle і DB2.

Мова Visual Prolog має потужні механізми пошуку розв'язку задач з механізмами звороту у випадку невдачі. Механізми дозволяють просто і компактно розв'язувати класи задач, що працюють з великою кількістю даних, вимагають швидкого пошуку та перебору даних за багатьма ключами. Мова має зручні засоби роботи із структурами.

В процесі викладання дисципліни треба розв'язати наступні задачі.

1. Ознайомити студентів з історією виникнення та сучасним станом логічного програмування.

2. Дати основи мови Visual Prolog, яка є прикладом реалізації напряму логічне програмування.

3. Показати достоїнства мови Visual Prolog та її переваги перед іншими мовами традиційного програмування для певних типів задач.

4. Навчити студентів практично застосовувати мову Visual Prolog до створення компонент систем штучного інтелекту та розв'язування інших прикладних задач.

Базовими знаннями для вивчення цього курсу є знання з основ програмування і з основ дискретної математики. Матеріал курсу є інструментальною основою дисциплін учбового плану, пов'язаних з проектуванням інтелектуальних систем. На знаннях курсу «Функціональне та логічне програмування» базується курс «Технології компонентного програмного забезпечення». У ньому вивчаються засоби модульного, об'єктна-орієнтованого та візуального програмування мовою Visual Prolog. Матеріал курсу застосовується під час дипломного проектування.

РОЗДІЛ І. ВИНИКНЕННЯ ТА СУЧАСНИЙ СТАН МОВИ ПРОЛОГ

Інтелектуальна діяльність людини це – пізнання навколишнього світу та застосування одержаних знань для розв'язування практичних задач. Для розв'язування задач людина створює алгоритми або адаптує відомі алгоритми.

Усього через 10 років після виникнення перших ЕОМ виникла ідея застосувати ЕОМ не тільки для виконання рутинних робіт при складних і громіздких обчисленнях, а для виконання інших видів інтелектуальної діяльності.

Зокрема, виникла ідея *знаходити програмно алгоритми для розв'язування задач на основі текстів задач.* Для реалізації ідеї необхідно було вирішити наступні проблеми.

1. Виділити і формалізувати зміст задачі, текст якої подається природною мовою.

2.Знайти узагальнений алгоритм, який міг би застосовуючи формалізований зміст задачі, розв'язувати задачу незалежно від її класу.

Реалізувати ідею можливо було двома способами: створенням інтелектуальних систем (ІС) або створенням мов програмування.

Сучасні спеціалізовані ІС можуть розв'язувати тільки задачі певного класу. ІС загального призначення розуміють текст задачі обмеженою природною мовою, перетворюють його зміст на внутрішню мову (опис задачі); вміють формувати за описом задачі алгоритми для розв'язування задач. Проте ІС загального призначення складні за структурою, великі за обсягом, а задачі розв'язують тільки кількох певних класів.

Реалізація ідеї створенням мови програмування виявилася більш продуктивною. Всі мови програмування базуються на двох поняттях – алгоритму і моделі. Деякі мови базуються тільки на алгоритмах, інші тільки на моделях, але більшість мов пов'язана з обома поняттями. Причому мови можна впорядкувати за зменшенням зв'язку з алгоритмами і збільшенням зв'язку з моделлю.

Ясно, що для реалізації ідеї – будувати алгоритми програмно, мови, що базуються тільки на алгоритмах, не підходять. Алгоритм розв'язування задачі знаходить людина і записує на мові програмування.

Слово модель означає зразок, який у спрощеній формі описує або показує який-небудь об'єкт або процес. Мови програмування, орієнтовані більше на модель, називають декларативними мовами. Слово декларація, з латинської declaration, означає об'яву, оголошення.

Програми, написані на декларативних мовах, описують основні властивості об'єкту, який створюється програмно. Причому мова програмування дозволяє гнучко описувати будь-які об'єкти.

До декларативних мов можна віднести в певному сенсі мову HTML. Опис на мові HTML Web-сторінки подає зміст сторінки, а не описує як відображувати сторінку на екрані.

Типовим прикладом декларативних мов програмування являються мови логічного програмування. Мови логічного програмування орієнтовані тільки на модель. На цих мовах створюються описи задач. Діапазон задач, які можна розв'язати на мовах логічного програмування, дуже великий.

Теоретичний напрям «Логічне програмування» засновано на ідеях математичної логіки. Напрям займається пошуком нових методів формального опису задач логічними формулами і пошуку методів формальних логічних виводів, що приводять до розв'язку задач. Наукові результати напряму «Логічне програмування» застосовують для створення нових мов програмування, машинних архітектур.

Мова Пролог є найбільш розповсюдженою мовою логічного програмування. Назва мови походить від слів «програмування в термінах логіки». Програми на мові Пролог являють собою опис задачі формалізованими твердженнями. Програми прозорі для розуміння, компактні. Виконуюча система мови Пролог застосовує опис задачі для виклику загальних механізмів Прологу, тим самим створюючи алгоритм, що розв'язує задачу.

Поява мови Пролог дозволила практично реалізувати ідею програмного створення алгоритму на основі опису задачі. Створення мови стало можливим завдяки появі конкретних передумов.

1.1 Передумови створення декларативної мови Пролог

1.1.1 Формалізація речень природною мовою на базі числення предикатів

Передумови створення декларативної мови Пролог виникли ще наприкінці XIX століття. В цей час були закладені основи математичної логіки.

Наукові праці Джорджа Буля вважаються початком алгебри логіки. Буль винайшов універсальну систему позначень і правил, застосовуючи яку до речень природної мови, він міг подавати їх у символічній формі. Така форма дозволяла виконувати над реченнями логічні операції подібно тому, як у алгебрі виконують операції над числами.

Істинні або хибні твердження називають *висловлюваннями*. Буль розглядав речення, які являють собою висловлювання. Основними логічними операціями у алгебрі висловлювань є *кон'юнкція* (логічне «І»), *диз'юнкція* (логічне «АБО»), *заперечення*(логічне «Ні»). Операції над висловлюваннями застосовувалися для доведення їх істинності.

У1879 році німецький логік, математик Фрідріх Людвіг Готліб Фреге ввів у алгебру логіки поняття: предметної змінної; предикату, як логічної функції; квантору. Це дало можливість побудувати нову гілку математичної логіки – числення предикатів. Готліба Фреге справедливо вважають родоначальником числення предикатів.

Саме поняття предикату було відомо задовго до Фреге. Слово «предикат» у вузькому значенні розуміли як – «властивість окремого предмета», наприклад «бути музикантом». У широкому значенні слово «предикат» розуміли як – відношення між кількома предметами, наприклад «бути сестрами». Аристотель, аналізуючи логічну структуру висловлювань природною мовою, застосовував поняття «предикат» у вузькому значенні.

Фреге розглядав предикат в широкому значенні. Він трактував *предикат, як особливий вид функції*, а замість предметів він застосував змінні.

Ця ідея виникла при розгляданні речень природною мовою, у яких застосовуються неозначені займенники для поширення змісту на будь-які об'єкти, суб'єкти, дії тощо. Прикладами неозначених займенників можуть бути: хтось, щось, ким-небудь, чого-небудь, когось, чийсь, якоїсь, де з ким, дехто, і. т. п.. У тлумачних словниках

значення слів завжди подаються з використанням неозначених займенників. Для прикладу візьмемо означення слова «дата» з сучасного тлумачного словника української мови за загальною редакцією доктора філологічних наук, професора В. В. Дубічинського: «Точний календарний час (рік, місяць, число) якоїсь події.». Неозначений займенник «якоїсь» поширює означення на події будь-якої предметної області.

За аналогією із неозначеними займенниками Фреге ввів у предикат замість предметів змінні. Таку конструкцію предикату тепер називають пропозиціональною формулою. Наприклад, формулою є «сестри (*X*, *Y*)». В загальному вигляді *пропозиціональна формула містить предикати із змінними, логічні операції, дужки.*

Змінні пропозиціональної формули Фреге назвав предметними змінними. Замінюючи кожну змінну пропозиціональної формули значенням він одержував висловлювання. Наприклад, сестри (Людмила, Валентина)».

Кожне висловлювання має істинність у предметній області, до якої воно відноситься. Тому можна сказати, що множина висловлювань предметної області відображується на множину значень «істина» і «хибність», тобто предикат є логічною функцією.

Зацікавленість Фреге формалізацією речень природної мови пов'язана з мисленням людини. Він займався розробкою універсальної мови подавання форм раціонального мислення.

Природною мовою форми мислення подаються оповідальними реченнями. Фреге вважав, що речення з нереальним змістом не мають істинності. Такі типи речень він не розглядав. Фреге вважають родоначальником розділу математичної логіки «Логічна семантика», присвяченому відношенню речення або його частин до реальності.

У своїй роботі «Зміст і денотат» Готліб Фреге стверджував, що оповідальні речення можуть мати значення (денотат) і зміст. Значенням речення він вважав «істину» або «хибність». Змістом твердження він вважав думку, що виражалася в ньому.

Фреге також показав, що не всі оповідальні речення мають істинність. Так у складнопідрядних реченнях істинність має тільки головне речення, а висловлювання підрядних речень можуть мати або не мати істинність у всій предметній області (ПДО). Наприклад, у реченні «Вона була впевнена, що займе перше місце у змаганнях»

головне речення «Вона була впевнена» має істинність відносно предметної області незалежно від підрядного речення. Підрядне речення «вона займе перше місце у змаганнях» не має істинності відносно ПДО, але має істинність відносно головного речення.

Для відображення області дії істинності пропозиціональних формул Фреге застосовував квантори.

Поняття квантору було відомо ще Аристотелю. Слово «квантор» з латинської означає «скільки». До Фреге кванторні висловлювання розумілися інтуїтивно і подавалися словами у тексті. Фреге вперше формалізував кванторно-логічні конструкції. Проте його конструкції були громіздкими і працювати з ними було незручно.

У сучасному численні предикатів квантор є логічним оператором, який вказує при якій кількості значень змінної ПДО пропозиціональна формула істинна у цій області. У численні предикатів існує два квантори: квантор загальності (∀) і квантор існування (∃).

Застосування *квантору загальності* до пропозиціональної формули означає, що пропозиціональна формула істина, для будь-яких значень *X* з ПДО.

Наприклад:

$$\forall x\ (жінка(X) => людина(X)) \qquad (1.1)$$

Формула (1.1) читається: «Кожна жінка людина».

Застосування *квантору існування* до пропозиціональної формули вказує, що пропозиціональна формула істина хоча б для одного значення *X* з ПДО.

Наприклад:

$$\exists\ x\ (чоловік(X)) \qquad (1.2)$$

Формула (1.2) читається: «Існує хоча б один чоловік».

Формула може мати обидва квантори.

Наприклад:

$$\forall x\ \forall y\ (\exists z\ (батько(Z, X)\ \&\ батько(Z, Y)\ \&\ стать(X, чоловік)\ \&\ стать(Y, чоловік)) => брати(X, Y)) \qquad (1.3)$$

Формула (1.3) читається: «Кожні два чоловіки, що мають спільного батька є братами».

Фреге вважав, що подавання висловлювань в уніфікованій формі можна було б використовувати при формальному логічному виводі. Проте розробкою формального логічного виводу він не займався, а можливість такого використання універсальної системи позначень приймав на віру.

Французький математик і логік Жак Ербран, норвезький математик Скулен і австрійський математик Курт Гедель незалежно один від одного довели, що система універсальних позначень Фреге повна. Повнота означає: універсальні позначення можна застосовувати для запису довільних логічно вірних формул, істинних при будь-якій інтерпретаціях та підстановках. Такий запис забезпечує формальний логічний вивід, тобто забезпечує доказ істинності висловлювання поза його змістом.

Сучасні логічні мови програмування, зокрема мова Пролог, базують опис задачі на численні предикатів, батьком якого був Готліб Фреге.

1.1.2 Створення алгоритму формального логічного виводу

Розвиток ідей Фреге привів до виникнення такої гілки математичної логіки, як обчислювальне числення предикатів, яка займається пошуком алгоритмів формального логічного виводу.

В 1930 році Жак Ербран придумав кілька версій алгоритму формального логічного виводу, які він застосовував для доведення теорем. Поліпшені алгоритми формального виводу Ербрана застосовуються до сих пір в мовах логічного програмування, зокрема в Пролозі. В одному з алгоритмів була використана процедура уніфікації.

Процедура уніфікації мала правила, за якими дві формули могли бути ототожнені. Правила уніфікації дозволяли пов’язувати різні логічні ланцюги через ототожнення формул.

Процедура уніфікації застосовувалася при логічному виводі. Процедура конкретизувала вільні змінні формул з різних логічних ланцюгів до тих пір, поки вони не ототожнювалися або виявлялось, що формули не можуть ототожнюватись.

Розглянемо правила уніфікації на прикладах *атомарних формул* (формул без логічних зв’язок).

Приклад 1. Нехай є два твердження «*X* чоловік» і «Петро чоловік».

Запишемо ці твердження в предикатній формі:

$$чоловік(X). \qquad (1.4)$$

чоловік(«Петро»). (1.5)

Якщо змінній *X* привласнити значення «*Петро»*, то формули (1.4) і (1.5) стануть тотожними. За рахунок ототожнення логічні ланцюги можна пов'язувати.

Приклад 2. Подамо речення «*Микола мислить*» і «*Петро мислить*» в предикатній формі:

мислить(«Микола»). (1.6)

мислить(«Петро»). (1.7)

Формули (1.6) і (1.7) не можуть бути ототожнені, аргументи предикатів подаються неоднаковими константами.

Приклад 3. Подамо речення «*X* мислить» і «*Y* мислить» в предикатній формі:

мислить(X). (1.8)

мислить(Y). (1.9)

Формули (1.8) і (1.9) можуть бути ототожнені, якщо змінні *X* і *Y* матимуть однакові значення.

У 1955 році датчанин Еверт Бет вперше зробив спробу запрограмувати алгоритми формального логічного виводу Жака Ербрана. Спроби продовжували і інші дослідники. Проте, алгоритми не підходили для обчислення на комп'ютері. При реалізації *алгоритму прямого формального виводу* початкова ціль (формула, істинність якої треба було довести) породжувала декілька нових цілей (формул). Нові цілі породжували знову декілька цілей. Продовжуючись, процес утворював таку кількість цілей, що комп'ютер не міг їх обробити. Тобто при спробах використання цього алгоритму, виникав комбінаторний вибух (рис. 1.1).

Невдалим був і алгоритм зворотного логічного виводу. У 1960 році швед Даг Правиця застосував у алгоритмі зворотного логічного виводу процедуру уніфікації Ербрана, проте комбінаторний вибух виникав.

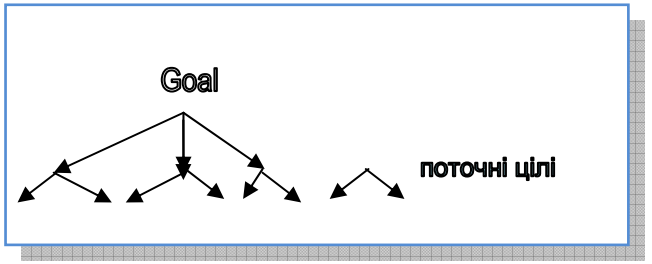


Рисунок 1.1 – Комбінаторний вибух

У 1961 році Дж. Робінсон придумав, яким чином поліпшити алгоритм формального логічного виводу, щоб обійти комбінаторний вибух. Робінсон винайшов *правило резолюції* (правило виводу), яке можна було застосовувати у алгоритмі на кожному кроці логічного виводу і яке дозволяло обмежувати кількість формул, що виводяться, застосовуючи процедуру уніфікації.

Правило резолюції стверджує наступне. Нехай є дві пропозиціональні формули *A* та *B*, такі що:

$$A=X_1 \vee A_1$$

$$B= \neg X_2 \vee B_1,$$

де A_1 і B_1 формули або порожні формули, а X_1 і X_2 *пропозиціональні змінні* (змінні, що конкретизується висловлюваннями).

При тотожності значень змінних X_1 і X_2 формула $A_1 \vee B_1$ є логічним наслідком формул *A* і *B*.

Сенс правила резолюції в тому, що при конкретизації змінних X_1 і X_2 одним висловлюванням істинність диз'юнкта $A \vee B$ залежить тільки від диз'юнкт $A_1 \vee B_1$, бо диз'юнкт $X_1 \vee \neg X_2$ невірний.

Формулу $A_1 \vee B_1$ (1.10)

називають *резольвентою* (з лат. «resolver» означає розв'язок).

Тотожність значень змінних визначалася процедурою уніфікації. Процедура уніфікації конкретизувала змінні X_1 і X_2 формул *A* і *B* до тих пір, поки значення змінних X_1 і X_2 не ставали тотожними або виявлялось, що їх тотожність неможлива.

Алгоритм формального логічного виводу являв собою алгоритм доведення теорем. Тобто, є формула-теорема F. Треба виявити чи буде формула F логічним наслідком формул(посилок) $F_1, F_2, \ldots F_n$.

Розглянемо доведення теореми методом від супротивного, де розглядається множина формул-посилок і формула протилежна теоремі $\{F_1, F_2, \ldots Fn, \neg F\}$. При цьому застосовується закон «виключеного третього», а саме, якщо теорема $\neg F$ невірна, то-теорема F істинна. Тобто, якщо при додаванні до множини посилок формули $\neg F$ одержується протиріччя, то теорема F істина. Алгоритм доведення подається нижче.

Записується множина формул $\{F_1, F_2, \ldots Fn, \neg F\}$ і перетворюється у кон'юнктивну нормальну *форму* (кон'юнкцію диз'юнкцій).

Далі послідовно розглядається множина *диз'юнктів* (без кон'юнкцій), що входять в формулу. До кожної пари диз'юнктів, що мають протилежні змінні X_1 і $\neg X_2$, застосовується правило резолюцій.

Якщо після застосування правила резолюції значення змінних X_1 і X_2 тотожні за процедурою уніфікації, то до посилок додається нова посилка - диз'юнкт (резольвента). При неможливості зіставлення значень змінних X_1 і X_2 виведений диз'юнкт не поповнює посилки. Тим самим кількість диз'юнктів (цілей, що породжуються), зменшується. Правило резолюції не застосовується повторно для двох тих самих диз'юнктів.

Дії повторюються для одержаної множини диз'юнктів до тих пір, поки не буде одержане протиріччя – порожній диз'юнкт X_1 і $\neg X_2$. Цей факт означає, що формула F є логічним наслідком посилок.

Якщо порожній диз'юнкт не може бути отриманий, то формула F не є логічним наслідком посилок $F_1, F_2, \ldots Fn$.

Цей метод одержав назву *метод резолюції*. Недоліком методу було те, що на великих задачах комбінаторний вибух залишався. Робінсон був першим, хто реалізував метод резолюції на комп'ютері.

У 60-і роки метод резолюційного програмування отримав широке поширення. Метод застосовували для доведення істинності теорем. На початку 70-років багато спеціалістів продовжували займатися проблемою опису задачі на базі логіки.

В той же час польський вчений *Р.* Ковальський та інші спеціалісти продовжували шукати стратегії удосконалення методу резолюції. Однією з вдалих стратегій виявилася стратегія лінійної

резолюції або SLD резолюції (Linear resolution with Selection function for Definition clauses). Лінійна резолюція є окремим випадком методу резолюції створеним для множини хорновських диз'юнктів.

До хорновських диз'юнктів відносять диз'юнкти виду:

$$F \vee \neg (X_1 \,\&\, X_2 \,\&\, X_3) \qquad (1.11)$$

Вираз (11) еквівалентний виразу (12).

$$X_1 \,\&\, X_2 \,\&\, X_3 \rightarrow F \qquad (1.12)$$

Тобто, з істинності кон'юнкції $X_1 \,\&\, X_2 \,\&\, X_3$ виходить, що формула F теж істина.

Якщо подати у виразі (1.11) диз'юнкцію двокрапкою (:), а заперечення рискою, то вираз запишеться у вигляді правила на Пролозі:

$$F: - X_1 \,\&\, X_2 \,\&\, X_3 \qquad (1.13)$$

Запис факту на Пролозі – це окремий випадок запису хорновських диз'юнктів:

$$F \vee \neg (X_1 \,\&\, \neg X_1), \qquad (1.14),$$

де вираз (X_1 & $\neg X_1$) завжди невірний і тому вираз (14) завжди вірний.

Ціль програми на Пролозі також подається хорновським диз'юнктом, якщо розглядати тільки праву частину загального виду хорновського диз'юнкту (1.11):

$$\neg (X_1 \,\&\, X_2 \,\&\, X_3) = \neg X_1 \vee \neg X_2 \vee \neg X_3 \qquad (1.15)$$

До того ж істинність такого диз'юнкту невідома і визначається роботою програми на Пролозі.

Важливою властивістю хорновських диз'юнктів є той факт, що хорновські диз'юнкти породжують тільки хорновські диз'юнкти. Тому застосування хорновських диз'юнктів спростило метод резолюції. В стратегії лінійної резолюції процес доведення повинен мати деревовидну

структуру і мати одну головну гілку, на якій розміщуються всі резольвенти (10). Тому дерево логічного виводу лінійно впорядковано. На кожному етапі застосовування правила резолюції послідовно обирається одна формула – посилка і остання одержана резольвента. Кожна посилка та одержані резольвенти являють собою поточні цілі. Тому дерево логічного виводу – це дерево цілей.

Метод лінійної резолюції цілком усунув проблему комбінаторного вибуху при логічному виводі.

1.1.3 Діалогова система, як прообраз функціонування систем логічного програмування

Великий вплив на створення мови Пролог мала також створена наприкінці 60-х років діалогова система Фостера і Елкока.

Вони розробили діалогову систему, яка не застосовувала логічний вивід на базі числення предикатів. Автори використовували звичайні математичні позначення. Для діалогової системи була розроблена мова, на якій програміст міг записувати у формалізованому вигляді твердження, що описують задачу. Такі твердження вважалися аксіомами. Використовуючи введені аксіоми, виконуюча система робила вивід про істинність питання поставленого до задачі (цілі задачі). Ця система була призначена для розв'язування широкого кола задач.

Система Фостера і Елкока була прообразом того, чим займається сучасне логічне програмування.

Твердження програми на Пролозі розглядаються як аксіоми. Ціль програми – твердження, яке визначає що треба знайти у задачі. Алгоритм формального логічного виводу застосовує аксіоми і знаходить розв'язок задачі, якщо це можливо.

1.1.4 Методи пошуку та перебору варіантів

У массачусетському технологічному інституті теж почали шукати інші шляхи опису задачі та її розв'язування, не на основі логіки. У 1967–1971 роках дослідник лабораторії штучного інтелекту цього інституту Карл Хюітт розробив мову програмування PLENER [2].

Мова призначалася для реалізації систем планування дій робота в певних умовах. В той же час механізми мови PLENER дозволяли розв'язувати і інші задачі штучного інтелекту: розуміння тексту

природною мовою, автоматичного доведення теорем, ведення діалогу між системою і користувачем тощо.

Мова базувалася на можливостях функціональної мови LISP – найбільш популярної мови штучного інтелекту. У мові PLENER крім відомих на той час методів розв'язування задач штучного інтелекту застосовувалися нові методи, які одержали широке розповсюдження пізніше.

До появи мови PLENER пошук алгоритму розв'язування задач часто виконувався *методом перебору варіантів*. У мові PLENER метод перебору варіантів застосовувався до тверджень, які описували всілякі стани об'єкту та відповідні станам дії, які може виконувати об'єкт у кожному стані. Множину таких тверджень будемо називати *моделлю середовища* об'єкту. Множину послідовностей дій, що веде від вхідного стану об'єкту до його певного кінцевого стану, назвемо *варіантом*. Якщо у варіанті кінцевий стан об'єкту збігається із ціллю задачі, то послідовність дій з тверджень варіанту є розв'язком задачі.

Розглядаючи задачу планування дій робота під час пожежі, можна сказати, що початковим станом робота може бути опис твердженнями наявності пожежі, місця знаходження робота і наявності у нього необхідних інструментів. Ціллю задачі є відсутність пожежі. Розв'язком задачі вважатиметься послідовність дій: рух до інструменту, взяти інструмент, рух до місця пожежі, знищення пожежі.

На рис. 1.2 показаний приклад схеми розв'язування задачі методом перебору варіантів. Числа на рисунку позначають вхідний стан об'єкту, проміжні та кінцеві стани, а стрілками позначають дії, що переводять об'єкт з одного стану в інший. Вхідний стан та ціль задачі позначені 1 і 10 відповідно.

Для певного стану об'єкту можуть вказуватися альтернативні дії, при виконанні яких об'єкт може переходити у різні стани. Такий стан називають альтернативною точкою. На рис. 1.2 всі точки альтернативні за винятком точки 5, та кінцевих точок 8, 9, 10, 11, 6, 12, 13, 14. Розглядаючи всі варіанти дій від початкового стану до кінцевого стану, треба знайти один варіант послідовності дій, який приводить до розв'язку задачі. Пошук послідовності дій одного варіанту називають *пошуком вглиб*.

Перебір варіантів виконується за таким алгоритмом: якщо обраний варіант не приводить до цілі, то виконується зворот до

найближчої альтернативної точки. Обирається не розглянута альтернатива і продовжується рух до кінцевої точки. Перебір нерозглянутих альтернатив називають *пошуком вшир.* По закінченню всіх альтернатив точки, що розглядалася, і недосягненні цілі, знову виконується рух назад до попередньої альтернативної точки. Після чого дії повторюються для попередньої альтернативної точки. Перебір закінчується, коли буде осягнена ціль або виявиться, що ціль осягнена не може бути.

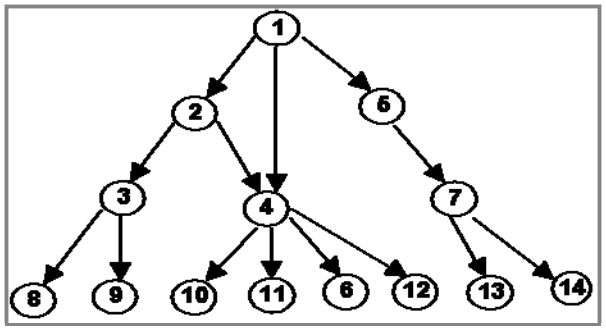


Рисунок 1.2 – Приклад схеми розв'язування задачі методом перебору варіантів

На рис. 1.2 перша альтернатива точки 2 веде до альтернативної точки 3, але всі її альтернативи не ведуть до цілі задачі. Зворот до другої альтернативи точки 2 приводить до цілі задачі.

До цілі задачі нашого прикладу ведуть такі варіанти: 1, 2, 4, 10; 1, 4, 10. Метод перебору варіантів не шукає найкращий варіант розв'язку, а обирає перший підходящий варіант. Вибір розв'язку залежить від того, яким методом виконується рух по дереву.

Для спрощення програмування метода перебору варіантів у мові PLENER вперше була реалізована функція, яка створювала зворот у альтернативну точку. Функція ліквідувала результати розглянутої альтернативи і підготовлювала роботу для іншої альтернативи або повертала виконання програми до попередньої альтернативної точки. Функція є прообразом сучасного *механізму зворотy* у Пролозі.

Мова PLENER мала спеціальні засоби опису задачі у вигляді формалізованих тверджень, але засоби не базувалися на логіці. Опис задачі твердженнями називали базою даних. Проте твердження не

можна було називати базою даних у звичайному сенсі. Кожне твердження не мало адреси за якою можна було звертатися до твердження. Доступ до твердження здійснювався тільки *методом зіставлення із зразком*.

До появи мови PLENER для пошуку твердження за введеними ключами виконувався процедурний аналіз значень компонент кожного твердження. Якщо пошук твердження виконувався за кількома ключами, то у процедурі необхідно було аналізувати різні можливі комбінації значень компонент тверджень.

Наприклад, нехай існують твердження з прізвищами та адресами людей міста: прізвище, вулиця, дім, квартира. Програмна реалізація пошуку повної адреси за вулицею і домом буде відрізнятися від реалізації пошуку за прізвищем і вулицею. Тому у процедурі треба передбачити всі можливі комбінації ключів, за якими буде здійснюватися пошук повної адреси.

У мові PLENER виконувався пошук тверджень, що описували різноманітні дії робота в певних умовах. Пошук реалізовувався стандартною функцією, яка накладала зразок на кожне твердження поки не знайдеться необхідне твердження. Зразок мав ту ж структуру, що і твердження. У зразку вказувалися тільки ключі для пошуку твердження, інші компоненти могли залишатися невизначеними змінними. При вдалому накладанні зразку його невизначені змінні одержували значення із твердження. Таким чином процедура пошуку за зразком була набагато менше за об'ємом, ніж процедура, що аналізує комбінації ключів.

У задачі планування дій робота під час пожежі твердження містять: місце знаходження робота, наявність інструмента у робота, стан об'єкта з пожежею, дії робота в цих умовах. Зразок для пошуку твердження буде містити описи станів: місце знаходження робота, наявність інструмента і стан об'єкта з пожежею, а невизначеними будуть дії робота в цих умовах.

Програмний механізм зіставлення зразку із твердженням був універсальним, тобто міг застосовуватися до будь-яких зразків та для різних типів тверджень. Твердження записувалися у вигляді списків. Списки можна було вкладати один в один.

Над твердженнями можна було виконувати наступні операції: пошук твердження у базі даних за зразком, викреслювання

твердження (відмітка твердження як не активного на певний момент), відновлювання твердження (активізація твердження).

На початку роботи програми на мові PLENER база даних була порожня. Після чого вона заповнювалася твердженнями, які описували модель – середовища функціонування об'єкту. Вводився початковий та кінцевий стан об'єкту. Початковий стан задавав зразок для пошуку відповідного твердження. Знайдене твердження визначало дію об'єкту. Після виконання дії змінювалися умови функціонування об'єкту. Твердження бази даних, що не відповідають поточному стану об'єкту, відмічалися викресленими. Твердження, що відповідають поточному стану відмічалися, як активні.

Дії повторювалися до тих пір, поки ціль не була осягнена або виявлялось, що її неможливо досягнути. База даних (БД) існувала під час виконання програми, а наприкінці роботи програми твердження вилучалася. Тобто PLENER розглядав твердження, як дані.

Механізми мови PLENER: зіставлення із зразком, пошуку вглиб та вшир, механізм звороту були принципово новими. Методи мови PLENER мали велике значення для створення мов логічного програмування і були реалізовані як механізми у мові Пролог.

Резюме

Створення мови Пролог стало можливим завдяки переліченим вище досягненням інших дослідників.

Діалогова система Фостера і Елкока є прообразом функціонування сучасних систем логічного програмування. Системи стали застосовувати загальний план розробки діалогової системи на основі логіки, який інтерпретувався, як процес обчислення. *Таке функціонування природне для розв'язування задач різних типів.*

Створена Фреге гілка математичної логіки – числення предикатів дозволила описувати формалізованими твердженнями зміст задачі, описувати твердженнями простір на якому функціонує задача.

Ербран розробив алгоритм формального логічного виводу, у якому він застосовував процедуру уніфікації для пов'язування кількох логічних гілок.

Алгоритм Ербрана, поліпшений Робінсоном введенням в алгоритм правила резолюції, дозволяв обійти комбінаторний вибух для малих задач. Надалі алгоритм був модифікований у лінійну

резолюцію на базі хорновських диз'юнктів, що повністю усунуло комбінаторний вибух.

Для спрощення пошуку складних даних будь-якої структури у мові PLENER був розроблений новий *універсальний метод зіставлення із зразком*. А для перебору варіантів у мові PLENER вперше створили функцію, яка реалізовувала *механізм звороту*. Функція спрощувала реалізацію універсальних *механізмів пошуку вглиб та вшир*.

Поява числення предикатів та алгоритму формального логічного виводу, який працював поза змістом тверджень, дозволила розв'язувати мовою Пролог *задачі різних типів*, а реалізація механізмів *спростити пошук і перебір даних*.

Контрольні питання

1. Як реалізується сьогодні ідея – програмне знаходження алгоритму розв'язування задачі на базі її тексту? Чи є інші шляхи реалізації ідеї?
2. На яких поняттях базуються всі мови програмування? Пояснити на яких поняттях базуються відомі вам мови.
3. Пояснити поняття декларативна мова. Чому Пролог декларативна мова?
4. Що навело Фреге на думку ввести змінну у предикат?
5. Що вам відомо про універсальну мову подавання форм раціонального мислення Фреге?
6. Які є квантори у численні предикатів і як вони застосовуються до пропозиціональної формули? Поясніть, що таке пропозиціональна формула, атомарна формула.
7. Що означає повнота універсальної системи позначень Фреге?
8. Де і для чого застосовується процедура уніфікації?
9. Що таке правило резолюції? Де воно застосовується і для чого? Визначити поняття пропозиціональної змінної, резольвенти.
10. Пояснити поняття лінійної резолюції, хорновських диз'юнктів.
11. Чому організація діалогової системи Фостера і Елкока мала вплив на створення мови Пролог?
12. Пояснити зміст механізмів пошуку вшир та вглиб.

13.Пояснити роботу механізму звороту у мові Пленер. Чи відрізняється робота цього механізму від роботи механізму звороту у мові Пролог?

14. В чому суть механізму зіставлення даних із зразком?

1.2 Мова Пролог і логічне програмування

У 1971 році в Марселі дослідницька група Марсельського університету на чолі з Аланом Колмероу створила мову програмування Prolog на базі перелічених компонентів. Назва мови означає – програмування в термінах логіки. Розроблювачі мови Prolog призначали її для створення програми-перекладача.

Після створення мови зробили кілька реалізацій мови Prolog, що розширювали його можливості та поліпшували засоби мови. Перші системи програмування мови Prolog мали тільки інтерпретатори. Це значно зменшувало швидкість роботи програми.

Інтерпретатор читав твердження програми, аналізував його і виконував дії відповідно твердженню. Виконувалася така програма повільно і тому програмування на Пролозі не застосовувалося для практичного програмування, а тільки для досліджень.

У 1977 році Д. Уоррен и Ф. Перейра з Единбурзького університету (Шотландія) створили мову Prolog для ЕОМ DEC-10. Створена система програмування Прологу мала інтерпретатор і компілятор, що значно наблизило мову до практичного застосування в порівнянні з більш ранніми версіями Прологу. Завдяки появі компілятора програма на Пролозі стала виконуватися швидше.

Але застосування інтерпретатору має також свої переваги. У функціональній мові штучного інтелекту Лісп до сих пір є інтерпретатор та компілятор і цей факт не є недоліком. Лісп розміщує програми та дані у виділеному в оперативній пам'яті середовищі. Інтерпретатор мови Лісп дозволяє легко переглядати проміжні та кінцеві результати роботи програми прямо у середовищі, що значно полегшує налагодження програм.

Діалект мови Пролог створений Д. Уорреном і Ф. Перейрою був найбільш поширеним в той час. Група дослідників Единбурзького університету реалізувала діалект на різних типах комп'ютерів та для різних операційних систем фірми DEC. Розробка діалектів була пов'язана з неможливістю переносити програми Прологу з одного

типу машини на інший тип машини і навіть з однієї операційної системи (ОС) на іншу операційну систему цієї ж машини.

Единбурзький Пролог реалізували на широко відомих ЕОМ фірми DEC VAX та PDP-11 в операційних системах UNIX та LINUX.

Причини неможливості перенесення мови Пролог були наступні. Наявність у мові тільки механізмів Прологу виявилося недостатньою для практичного застосування мови. Необхідно було мати традиційні засоби роботи програм. У мові такі засоби реалізовувалися вбудованими предикатами. До вбудованих предикатів відносять предикати вводу-виводу, роботи з типами даних.

Визначення вбудованих предикатів не можна описати мовою Пролог, їх писали тією мовою, на якій реалізовувалася мова Пролог. Вбудовані предикати мали «побічні ефекти», вони застосовували можливості певної ЕОМ та ОС настільки, наскільки їх застосовувала алгоритмічна мова, на якій створювався Пролог.

Отже можливість перенесення мови Пролог залежала від наявності у ній вбудованих предикатів. Кількість тверджень програми також залежала від реалізації механізму керування динамічною пам'яттю в операційній системі.

Сучасна мова Visual Prolog генерує програми для різних ОС за винятком деяких можливостей ОС, що застосовуються мовою. Щоб перенести програму, із однієї ОС в іншу у системі програмування є конвертор (VIPCONV. exe).

Діалект Д. Уоррена і Ф. Перейри назвали на честь авторів книги «Программирование на Прологе» У. Клоксіна та К. Мелліша [3] – C&M. Ця книга стала першим підручником по логічному програмуванню. У книзі була описаний створений ними діалект. Единбурзький діалект Прологу C&M неофіційно вважали за стандарт мови Пролог.

Перший діалект Прологу для персональних комп'ютерів був створений у 1980 році К. Кларком і Ф. Маккейбом (Великобританія).

У 1981 році японці зробили заяву о своєму намірі створити до 2000 року ЕОМ п'ятого покоління. Метою проекту було створення програмних систем нового типу, що базуються на знаннях. Програмне забезпечення ЕОМ повинно було базуватися на методології логічного програмування, а головним засобом створення операційних систем передбачалося зробити мову Prolog. Проте проект не був реалізований.

Наприкінці 80-років одним з найбільш популярних діалектів мови Пролог стала мова Турбо-Пролог, яку створила компанія Borland International (США).

Популярність мови забезпечила реалізація мови на персональних комп'ютерах, що зробило її доступною для широкого кола користувачів, а також висока швидкість роботи програми.

Система програмування Турбо-Прологу мала зручний та простий у роботі гарний на вигляд віконний інтерфейс. Зручний відладчик дозволяв легко слідкувати за виконанням програми. Компілятор моделював процес виконання програми, за рахунок чого міг виявити більшість помилок її виконання. Турбо-Пролог застосовують і сьогодні. Він має продуманий і невеликий набір вбудованих предикатів, які дозволяють розв'язувати різноманітні задачі.

У 1996 році опублікували офіційний стандарт ISO/IEC 13211-1, який стандартизує основні елементи мови Prolog. Стандарт мови програмування – це опис ядра спільного для діалектів мови Prolog безвідносно до ОС та комп'ютера. Програма, що застосовує засоби, описані в стандарті – переносна. У 2000 році опублікували стандарт ISO/IEC 13211-2, який описує поняття та методи модульного програмування мовою Prolog.

Серед сучасних діалектів мови Prolog можна виділити діалект Strawberry Prolog, розроблений у 1996 році Інститутом математики і інформатики болгарської академії наук. Ядро діалекту близьке до ISO Prolog. Strawberry Мова Prolog має розширення, яке не входять до стандарту. Ядро Strawberry Prolog легке у використанні і тому його часто застосовують при вивченні мові Prolog.

Мови PDC і Visual Prolog датської фірми Prolog Development Center є найбільш розповсюдженими мовами логічного програмування. Мова візуальний Пролог і його система програмування мають найбільшу функціональність серед мов логічного програмування. Можливості мови значно перевищують можливості інших логічних мов.

Популярність мови Пролог привела до появи родини мов програмування подібних Прологу, до таких можна віднести мови T-Prolog, M-Prolog, LQF. Для родини мов можна побудувати генеалогічне дерево, коренем якого є мова Пролог. Одночасно з появою мови виник новий теоретичний напрям «Логічне програмування» і нова парадигма мов програмування – логічне програмування.

Теоретичний напрям «Логічне програмування» розуміють у вузькому та широкому сенсі.

У вузькому сенсі логічне програмування займається пошуком нових методів опису задач логічними формулами на базі хорновських диз'юнктів та пошуком нових методів розв'язування задач шляхом формального логічного виводу з використанням методу резолюції.

У широкому сенсі логічне програмування займається пошуком нових методів описів задач твердженнями в деяких формальних логічних мовах та пошуком ефективних методів розв'язування задач виводом в деякій формальній дедуктивній системі.

Логічне програмування досліджує зв'язки логічного програмування з іншими стилями програмування, зокрема з методами паралельних обчислень; з функціональним програмуванням. В результаті з'явилися нові мови паралельного програмування Parlog, Concurrent Prolog; мови , що комбінують логічне та функціональне програмування LogLisp, LEAF.

Мови логічного програмування відносяться до однієї з головних сучасних парадигм програмування. Під парадигмою програмування розуміють спосіб мислення і програмування, що не пов'язані з конкретною мовою програмування. Тобто, під парадигмою програмування розуміють ті ідеї на яких заснована група мов програмування.

Парадигмою логічного програмування є *формальний опис простору, на якому функціонує задача та опис задачі у вигляді тверджень поданих логічними формулами. Розв'язок задачі знаходиться формальним логічним виводом.*

Резюме

Мова Пролог є родоначальником родини мов логічного програмування подібних їй. Найбільш поширеними діалектами мов цієї родини є Turbo-Prolog, PDC – Prolog, Visual Prolog. Одночасно з виникненням мови Пролог народився новий теоретичний напрям – логічне програмування та парадигма логічного програмування.

Контрольні питання

1. В чому недоліки інтерпретаторів?
2. В чому переваги інтерпретаторів перед компіляторами?
3. Пояснити, чому перші діалекти мови Пролог були непереносними з машини на машину та з однієї ОС на іншу?

4. Якою мірою є переносним Visual Prolog? Що зроблено у мові для можливості перенесення програм?

5. Що таке стандарт діалектів мови програмування?

6. Чи є стандарт у діалектів мови Пролог?

7. Як ви розумієте поняття родини мов програмування?

8. Які ви знаєте мови програмування родини Пролог?

9. Чим займається теоретичний напрям «Логічне програмування» у широкому та вузькому сенсі?

10. Яки ви знаєте напрями розвитку мов логічного програмування?

11. Сформулюйте визначення парадигми програмування та парадигми логічного програмування.

1.3 Можливості мови і системи програмування Visual Prolog

Мова Visual Prolog має всі достоїнства перших мов родини і до того ж не має їх недоліків [4].

Реалізація сучасної мови Visual Prolog нерозривно пов'язана з ОС. А саме, мова застосовує ідеї, концепції, програмні засоби операційної системи, з якими вона взаємодіє через інтерфейс програмування додатків (API). До таких концепцій відноситься поточна організація обміну даними, робота з оброблювачами подій.

Графічний програмний інтерфейс мови Visual Prolog (GUI API) – інтерфейс високого рівня. Він узагальнює в собі можливості інтерфейсів сучасних віконних ОС. За рахунок цього досягається по-більшості незалежність мови від операційної платформи.

Мова забезпечує програмування у операційних системах: Dos, Windows, Linux, компанії SCO Unix. Мова Visual Prolog дозволяє створювати програми для різних платформ: Dos, розширеного DOS, Windows16, Windows32, ОС/2 16 біт, ОС/2 32 біт, Linux. Візуальні засоби не підтримуються тільки у Linux та DOS.

До непереносних предикатів відносять: вбудовані предикати віконного інтерфейсу, предикати, які стискують графічну інформацію, предикати обробки деяких винятків синтаксису. Інший код програми за винятком графічного інтерфейсу повністю переносний.

Графічний інтерфейс Visual Prolog дозволяє легко реалізовувати оболонки програм і підтримувати складні інтерфейси між ОС і програмою користувача.

Користувач може змінювати вигляд інтерфейсу графічними редакторами (інтегровані редактори для підготовки ресурсів). Ресурси можуть бути імпортовані з інших застосувань користувачів, динамічних бібліотек, файлів ресурсів.

Програма на Visual Prolog працює зі швидкістю порівнянної із швидкістю програми на C++. Текст програми на Visual Prolog складає 1/10 частину коду аналогічної програми на C++. Але, в процесі розв'язування певної задачі може знадобиться реалізувати такі функції системи, для яких неможна застосувати ні механізми Visual Prolog, ні його вбудовані предикати. До таких задач відносяться задачі керування програмно-апаратними комплексами та технічними системами. Тому програму на візуальному Пролозі можна компонувати з програмами на мовах Асемблер, Паскаль та *C*. Кажуть, що для мови Visual Prolog характерна відкрита платформа.

Програма на мові Visual Prolog може викликати *функції* мови MSVC++32 біт, якщо об'явити їх як глобальні предикати з опцією language C++ або з опцією стандартного виклику STDCALL. В свою чергу предикат Visual Prolog можна викликати із програми на MSVC++32 біт, якщо об'явити його, як глобальний предикат з опцією language MSVC++.

Візуальний Пролог має між програмний інтерфейс спільний для всіх компіляторів, що генерують об'єктний код програми. Тому умовою компоновки програми на Visual Prolog з програмами на інших мовах є наявність у цих мов компіляторів, що генерують об'єктний код.

Система програмування Visual Prolog надає можливість гнучко слідкувати за роботою програми під час її налагодження. Система має розвинений відладчик, оптимізований компілятор для готових програм, дозволяє працювати над одним проектом кільком програмістам.

Утворення програми користувача виконується за допомогою візуального середовища розробки програм – VDE (Visual Development Environment). Середовище містить текстовий і графічні редактори, компілятор, експерт коду, експерт застосування, компонувальник, відладчик, утиліту тестування цілі – TestGoal, браузери різних типів. Користувач застосовує певні типи програм середовища залежно від вибору параметрів свого проекту.

Експерт додатків значно полегшує створення програми користувача. Експерт додатків генерує і конфігурує проект згідно

обраних користувачем платформою, стратегією вводу-виводу, інструментальними засобами, пакетами програм. Експерт додатків сам задовольняє вимоги компонувальника програм.

Відладчик дозволяє виконувати вашу програму по крокам, переглядаючи значення змінних для кожного предикату, або встановлювати в програмі точки зупинки, щоб переглянути значення змінних предикатів вибірково.

Відладчик також дозволяє переглядати вміст фактів і вилучати їх, переглядати події візуального інтерфейсу, вміст динамічної пам'яті, стеку, одержати дамп пам'яті. Програміст може переглянути програму, як на рівні Прологу, так і на рівні асемблеру. Відладчик Visual Prolog – окрема програма, яку можна викликати з середовища VDE або з ОС.

Візуальний Пролог відноситься до багатофункціональних мов. Він дозволяє розроблювати *клієнт-серверні застосування* для Windows NT, Linux, Unix. Підтримує основні протоколи TCP/IP, FTP, HTTP мережевого обміну.

Мова має конвертори перетворення HTML, RTF, IPF форматів сторінок в структури Прологу і навпаки.

Засобами Visual Prolog можна: реалізовувати SQL запити до зовнішніх БД за допомогою інтерфейсу, який базується на бібліотеках ODBC та OCI баз даних Oracle і DB2; створювати сервери БД або логічні сервери.

Візуальний Пролог може працювати з *внутрішньою і зовнішньою базами даних. Внутрішня база даних* – це статичні або динамічні факти на Пролозі. Статичні факти розміщуються у програмі, їх не можна завантажувати, змінювати, вилучати. Динамічні факти можна розміщувати у файлах і завантажувати у програму.

Зовнішня БД не застосовує механізми Прологу, а застосовує свої засоби доступу до БД. Вона дозволяє розміщувати БД великих об'ємів на диску або в оперативній пам'яті, одержувати швидкий доступ до даних за ключами; зберігати їх на диску або завантажувати дані в оперативну пам'ять у двійковому вигляді. Для роботи із зовнішньою БД існують спеціальні предикати.

При інсталяції Visual Prolog 5.2 можна включити в систему програмування *оболонку експертної системи ESTA*. Користувач може настроювати і включати її в свою програму. В Україні на візуальному

Пролозі розроблено систему НАУ (Нормативні акти України). Експертна система може підключатися до WEB-сторінок.

Об'єктна-орієнтований стиль програмування мови Visual Prolog разом із стилем логічного програмування являє собою могутній інструмент моделювання ПДО.

Мова Visual Prolog еволюціонує, що дозволяє користувачам розширювати свої знання, а не вивчати нові мови.

Резюме

Мова Visual Prolog відноситься до розвинених сучасних мов родини Пролог. Вона поєднує у собі можливості мов логічного та об'єктна-орієнтованого програмування із застосуванням сучасних WEB технологій та БД. Програма на мові Visual Prolog працює із швидкістю порівняній із швидкістю програми на C++. Об'єм програми майже в 10 разів менше об'єму програми на C++. Програму можна компонувати з програмами на інших мовах. Умовою компоновки програми на Visual Prolog з програмами на інших мовах є наявність у цих мов компіляторів, що генерують об'єктний код програми.

Контрольні питання

1. У яких ОС можна застосовувати мову Visual Prolog?
2. Якою мірою код програми мовою Visual Prolog переносний із однієї ОС у іншу?
3. Перелічити програми, що входять у візуальне середовище розробки програм. Вказати їх призначення.
4. Які є можливості у відладчика системи програмування Visual Prolog?
5. Як полегшує роботу програміста експерт додатків?
6. Перелічити засоби Visual Prolog, що дозволяють працювати у мережі.
7. Чи можна компонувати програми написані мовою Visual Prolog з програмами написаними іншим мовами? Перелічити ці мови.
8. З якими БД може працювати Пролог? Дати характеристики цих БД.
9. Які є можливості зовнішньої бази даних Visual Prolog?
10. Які формати сторінок підтримує Visual Prolog?

РОЗДІЛ ІІ. ТРАДІЦІЙНЕ ПРОГРАМУВАННЯ ЗАСОБАМИ ПРОЛОГУ

Ідея автоматичного створення алгоритмів розв'язування задач на базі опису задач спочатку не була пов'язана ні з якими типами задач. Проте в основу мови Пролог були покладені методи розв'язування задач штучного інтелекту. Надалі виявилося, що ці методи розв'язування задач цілком підходять і для реалізації алгоритмів імперативного програмування (програмування в командах, операторах, процедурах).

Ще в 1976 році Р. Ковальський і М. Ван Емден запропонували два підходи *до читання* логічних програм – декларативний і процедурний.

Інтерпретація логічної програми, як програми, що написана за імперативним підходом, наближає мову Пролог до мов традиційного програмування. Після знайомства з процедурним підходом читання програм більшість користувачів мови, міркуючи над створенням програми на Пролозі, починає інтуїтивно застосовувати процедурний підхід.

2.1 Процедурний підхід до реалізації програм мовою Visual Prolog

Програма, що написана алгоритмічною мовою програмування, має основними елементами: послідовне виконання дій, розгалуження, цикли. Процедурний підхід дозволяє розглядати цільове твердження програми, як виклик однієї процедури, або як послідовність викликів кількох процедур користувача та стандартних. Послідовність викликів процедур задається порядком запису цілей у кон'юнкції. Аргументи цільових тверджень трактуються як фактичні параметри процедур.

Умовні твердження програми та факти, що записані одним предикатом так і називають процедурою. Щоб полегшити роботу механізмів Прологу існує правило «У тексті програми твердження записані одним предикатом записуються підряд.».

Запис умовного твердження разом з його умовами (правило) трактують як запис процедури, де умовне твердження подає ім'я процедури та її формальні параметри, а умови (поточні цілі) подають тіло процедури. Тіло може містить виклики інших процедур, які виконують необхідні для основної процедури дії, одержують дані з фактів тощо.

Якщо умовне твердження має кілька гілок умов, то процедура розгалужується. У кожній гілці є одна або кілька поточних цілей, які задають умови виконання гілки. Такі цілі розташовують на початку гілки, інші поточні цілі гілки виконують необхідні за умовами дії або викликають інші процедури, що виконуватимуть необхідні дії.

Серед поточних цілей гілок можуть заходиться виклики процедур себе. Тобто процедура може виконувати дії, які повторюються рекурсивно. Факти, записані тим же предикатом що і умовне твердження, застосовують як обмеження роботи рекурсивної процедури. Застосування у гілці процедури механізму звороту також заставляє процедуру виконувати дії, що повторюються.

Розглянемо приклад простої задачі, для розв'язування якої природна послідовність дій, і програму мовою Visual Prolog, що розв'язує задачу.

Задача №1. Виявити, чи є парним введене ціле число.

Програма виконує наступну послідовність дій: вводить ціле число, перевіряє число на парність та виводить результат залежно від типу числа.

```
Predicates
Nondeterm перевірити_парність_числа (integer)

Clauses
перевірити_парність_числа(N):-
        N mod 2 = 0, write («Число», N, «парне»), nl, !;
        write («Число», N, «непарне»), nl.

Goal
write («Введіть ціле число»), readint (N),
перевірити_парність_числа (N).
```

Ціль програми задана кон'юнкцією цілей. Кожна ціль кон'юнкції послідовно зліва направо стає поточною ціллю і виконується. Остання поточна ціль є викликом процедури «перевірити_парність_числа» з фактичним параметром – ціле число.

Процедура одержує число в свій формальний параметр *N*. За першою гілкою вона перевіряє число на парність стандартною операцією *mod*. Гілки умовного твердження працюють аналогічно

оператору IF. Якщо число парне (Then – форма), то продовжує працювати перша гілка і виводиться повідомлення про парність числа, інакше працює друга гілка (Else – форма), виводиться повідомлення про непарність числа.

В результаті послідовно виконаються дії.

1. Виводиться прохання ввести число.
2. Вводиться число.
3. Перевіряється парність числа.
4. Якщо число парне, то виводиться повідомлення про парність числа, інакше виводиться повідомлення про непарність числа.

Розглянемо задачу типову для імперативних мов, розв'язок якої вимагає дій, що повторюються. Розв'язування задач такого типу імперативною мовою вимагає застосування оператору циклу. Наприклад, оператор циклу у мові Паскаль *for*. Для розв'язування задачі мовою Visual Prolog будемо виконувати дії, що повторюються, механізмом звороту та висхідною рекурсією.

Задача№2. Завантажити квадратну матрицю a_{ij} порядку 3, транспонувати її та вивести транспоновану матрицю на екран.

Операція транспонування матриці полягає у заміні *i*-рядків матриці на її *i*-стовпці.

Програма вводить матрицю a_{ij} узагальненим списком (списком, елементи якого теж є списки). Кожен елемент узагальненого списку подає рядок матриці.

Матриця перетворюється у множину фактів, де кожен факт містить індекси певного елементу і сам елемент матриці. Таке подавання матриці значно полегшує роботу з її елементами. Транспонування матриці поданої фактами виконується знаходженням кожних двох фактів виду a_{ij} та a_{ji}, у яких треба поміняти місцями елементи матриці. Елементи матриці, які розташовані на діагоналі, не змінюються.

Зберігання матриці у фактах має свій недолік. Після цієї операції порядок фактів перестає відповідати розташуванню елементів у матриці і необхідно відсортувати факти. Сортування фактів програма виконує за набором фактів індексів, розташованих у необхідному порядку.

Перетворення матриці. поданою фактами, до узагальненого списку програма виконує стандартним предикатом FINDALL.

Предикат FINDALL автоматично створює список з аргументів фактів за вказаною змінною. У програмі створюються три списки для трьох рядків. Рядки об'єднуються в узагальнений список.

Для тестування програми ми будемо застосовувати квадратну матрицю порядку 3 у вигляді узагальненого списку:

[[2.5, 3.2, 2.7], [1.5, 4.1, 1,1], [2.3, 4.5, 6.1]].

```
Global domains
list = real*
lst = list*

Global facts
Nondeterm a(integer, integer, real)
Nondeterm a1(integer, integer, real)
Nondeterm  індекс (integer, integer)

Predicates
Nondeterm створити_базу (lst, integer, integer)
Nondeterm транс_матр ()
Nondeterm сорт ()
вивід()

Goal
write («Введіть матрицю>«), readterm(lst, L),
створити_базу(L, 1, 0), транс_матр(), сорт(), вивід(),!.

Clauses
створити_базу([],_,_).

створити_базу([H|T], C1,_):-H=[], C11=C1 + 1, C22=0,
                            створити_базу(T, C11, C22).

створити_базу([H|T], C1, C2):-H=[H1|T1], C22=C2 + 1,
                            assert (a (C1, C22, H1)),
                            assert (індекс (C1, C22)),
                            створити_базу ([T1|T], C1, C22).

транс_матр():-a(I, J, R1), I<>J, a(J, I, R2),
                            retract (a (I, J, R1)),
                            retract (a (J, I, R2)),
                            assert (a1 (I, J, R2)),
```

```
                                assert(a1(J, I, R1)), fail.
транс_матр ():- a(I, J, R), I = J, retract(a(I, J, R)),
                                assert (a1 (I, J, R)), fail.
транс_матр ().

сорт():- індекс(I, J), a1(I, J, R1), assert (a(I, J, R1)),
                                retract (a1(I, J, R1)), fail.
сорт().

вивід():- findall(R1, a(1, _, R1), L1), write(L1), nl,
          findall(R2, a(2, _, R2), L2), write(L2), nl,
          findall(R3, a(3, _, R3), L3), write(L3), nl,
          L = [L1, L2, L3], write (L).
```

Процедура «створити _базу» одночасно створює факти матриці a_{ij} і факти індексів для сортування матриці. Процедуру реалізовано висхідним методом рекурсії. Вона має дві граничні умови за кінцем узагальненого списку і за кінцем елементів узагальненого списку. Зверніть увагу на те, що застосування стандартного предикату findall вимагає об'яви списку у секції Domains навіть в тому випадку, якщо в інших предикатах програми списки не застосовуються.

Вхідні дані та результат роботи процедури подано нижче.

Факти матриці a_{ij}:

```
a(1, 1, 2.5).
a(1, 2, 3.2).
a(1, 3, 2.7).
a(2, 1, 1.5).
a(2, 2, 4 1).
a(2, 3, 1.1).
a(3, 1, 2.3).
a(3, 2, 4.5).
a(3, 3, 6.1).
```

Факти індексів для сортування матриці a_{ij}:

```
індекс(1, 1, 2.5).
індекс(1, 2, 3.2).
```



```
індекс(1, 3, 2.7).
індекс(2, 1, 1.5).
індекс(2, 2, 4 1).
індекс(2, 3, 1.1).
індекс(3, 1, 2.3).
індекс(3, 2, 4.5).
індекс(3, 3, 6.1).
```

Процедура транспонування матриці та процедура сортування матриці виконують операції, що повторюються механізмом звороту. Процедура транспонування матриці одержує факти матриці $a1_{ij}$ у наступному порядку.

```
a1(1, 2, 1.5).
a1(2, 1, 3.2).
a1(1, 3, 2.3).
a1(3, 1, 2.7).
a1(2, 3, 4.5).
a1(3, 2, 1.1).
a1(1, 1, 2.5).
a1(2, 2, 4.1).
a1(3, 3, 6.1).
```

Для вірного групування фактів матриці $a1_{ij}$ застосовується процедура sort. Після сортування матриці $a1_{ij}$ формується матриця a_{ij} у вірному порядку*: a11, a12, a13*, … . Стандартний предикат FINDALL формує з фактів узагальнений список.

Транспонована матриця a_{ij} у вигляді узагальненого списку подана нижче.

```
[[2.5,1.5,2.3], [3.2,4.1,4.5], [2.7,1.1,6.1] ]
```

Наступна задача також типова для імперативних мов. Розв'язування задачі імперативною мовою нагадує застосування оператору циклу *while*, але виконання дій припиняється *за виконанням умови* або кількох умов.

Задача №3. Дана матриця з цілих чисел, де a_{ij} елементи матриці

(i = 1, 2, 3, 4; j = 1, 2, 3). Обчислити :

$$b_j = \sum_{i=1}^{4} a_{ij}$$

У програмі, яка розв'язує задачу №3, дії, що повторюються, реалізуються низхідною рекурсією. Реалізація програми подібна реалізаціям програм імперативними мовами, які застосовують вкладені цикли. Матрицю a_{ij} подамо узагальненим списком.

Програма виконує наступні дії.

1. Вводить матрицю цілих чисел a_{ij} у вигляді узагальненого списку.
2. Процедура сума_*j* формує суму елементів поточного стовпця матриці a_{ij} та вилучає з матриці a_{ij} стовпець, суму елементів якого знайдено.
3. Процедура b_j поповнює матрицю b_j сумою елементів стовпця, одержаною процедурою сума_*j*.
4. Процедура b_j повторює дії 2,3 до вичерпання стовпців матриці a_{ij}.
5. Виводить матрицю b_j на екран.

Для тестування програми застосовується матриця a_{ij}:

[[5, 7, 2,1], [3, 4, 6, 9], [8, 6, 2, 3]]

```
Global Domains
рядок = integer*
матриця = рядок*

predicates
Nondeterm сума_j(матриця, integer, матриця)
Nondeterm bj(матриця, рядок)

Goal
write(«Введіть матрицю»), readterm(матриця, M),
bj(M, BJ), write(BJ).

Clauses
bj([], []):-!.
bj(M, BJ):- сума_j (M, Sum, M1), write(M1, «», SUM),
                nl, !, bj(M1, BJ1), BJ = [Sum|BJ1].
```



```
сума_j([], 0, []):-!.
сума_j([H|T], S1, M1 ):-H = [H1|[]],!, сума_j (T, S, M1),
                                S1=S+H1.

сума_j([H | T], S1, M1 ):-H=[H1| T1],!, сума_j(T, S, M),
                                S1=S + H1, M1=[T1|M].
```

Процедура suma_*j* має дві граничні умови по закінченню стовпців матриці a_{ij} і у випадку, коли матриця має один стовпець. Друга гранична умова необхідна для знаходження суми елементів останнього стовпця і щоб не додати у матрицю a_{ij} порожні списки.

Результат: [16, 17, 10, 13] .

Резюме

Мова Visual Prolog дозволяє розв'язувати задачі імперативного програмування. Мовою можна програмувати послідовні, умовні та циклічні елементи програми. В той же час застосування фактів для подавання даних програми робить програму простою та прозорою для розуміння.

Контрольні питання

1. Якими засобами Visual Prolog можна реалізувати послідовність виконання дій та розгалуження?
2. Якими засобами Visual Prolog можна реалізувати циклічні операції?
3. Як на мові Visual Prolog можна ввести та подати матрицю?
4. Для чого в програмі об'являються списки?
5. Для чого у задачі №2 застосовується стандартний предикат FINDALL?

Вправи

1. Дана матриця з дійсних чисел, де a_{ij} елементи матриці (i= 1, 2, 3, 4 ; j = 1, 2, 3, 4,5). Обчислити:

$$b = \sum_{i=1}^{4} \sum_{j=1}^{5} a_{ij}$$

2. Знайти суму парних чисел, що належать відрізку цілих чисел, які вводяться користувачем.

3. Обчислити cos (15x) за формулою

cos (n*X) = cos ((n–1) X) *cos(X)–sin((n–1) X)* sin(X),

якщо cos (X) = 0,15, 0 < X < π/2.

2.2 Сталість роботи програми мовою Visual Prolog

Всі сучасні мови імперативного програмування забезпечують контроль роботи програми спеціальними засобами обробки збійних ситуацій. Промислові програмні системи працюють з великими обсягами даних. При збою програми користувач загублює час і втрачає результати роботи. Проблема стійкості програми актуальна для всіх програмних систем незалежно від того, якими мовами реалізуються система.

Наявність у мови Visual Prolog високоефективного компілятору дозволяє максимально виявляти помилки програми. Компілятор моделює роботу програми і знаходить можливі помилки виконання програми. Компілятор перевіряє: конкретизацію змінних, вплив детермінізму предикатів на роботу програми, контроль відповідності типів аргументів предикатів тощо. Проте помилки виконання програми залишаються.

Для забезпечення стійкості програми під час її роботи Visual Prolog має спеціальні стандартні предикати контролю та обробки збою.

Взагалі всі стандартні предикати мови Visual Prolog та предикати користувача повертають істину або неправду. Крім того, кожен предикат має побічний ефект – виконувати необхідну дію в програмі. Саме при реалізації побічних дій можливі збійні ситуації.

Виконуюча система програмування мови Visual Prolog контролює наступні помилки виконання:

- виконання програми;
- вводу-виводу даних;
- роботи із зовнішньою БД;
- обміну програмних додатків даними в ОС Windows, ОС2 за механізмом Dynamic Data Exchange (DDE);
- роботи візуального інтерфейсу BGI;
- роботи візуального інтерфейсу VPI;
- звертання до програм ОС Windows, UNIX, DOS, OS2;

- застосування ODBCBIND пакету для доступу до БД через програмний інтерфейс ODBC;
- застосування SOCKBIND пакету для взаємодії застосувань через єдиний програмний інтерфейс у комп'ютерних мережах;
- застосування програмного інтерфейсу програмних додатків до протоколу HTTP.

Ми розглянемо тільки обробку помилок виконання програми, вводу-виводу та ОС. Інші типи помилок лежать поза змістом цієї книги.

Приклади повідомлень про помилки виконання програми
RUNTIME ERRORS

1001 Переповнення глобального стеку. Не вистачає пам'яті або нескінченний цикл.

1002 Переповнення купи. Не вистачає пам'яті або нескінченний цикл.

1007 Терм занадто великий.

1010 Переповнення стеку.

1024 Об'єкту не існує або некоректне перетворення його типу.

1026 Прямий виклик предикату з абстрактного класу.

1031 Арифметичне переповнення.

1032 Ділення на нуль.

1041 Спроба стверджувати другий примірник факту, оголошеного як детермінований.

1042 Спроба відмовитися від факту, оголошеного як єдиний.

1045 Індекс занадто великий.

Приклади повідомлень про помилки вводу-виводу
IO & OS errors

1101 Спроба відкрити вже відкритий файл.

1104 Файл не відкритий.

1107 Предикат EOF може застосовуватися тільки під час читання.

1110 Файл занадто великий або не вистачає пам'яті файлу.

1113 Невірний аргумент предикату date.

1114 Невірний аргумент предикату time.

1133 Невірна версія операційної системи.

1134 Невірний шлях.

1140 Занадто великий writef або формат.

Приклади повідомлень про помилки операційної системи
Operating system errors

7002 Файл не знайдено.
7003 Шлях не знайдено.
7019 Диск захищено від запису.
7080 Файл вже існує.
7029 Помилка запису.
7030 Помилка читання.

Основним предикатом контролю збійних ситуацій є предикат trap.

trap (предикат_виконання, код_помилки, предикат_обробки)

(*i, o, i*)

Предикат контролює виконання стандартних предикатів або предикатів користувача, у роботі яких можуть виникнути помилки всіх вище вказаних типів. Предикат trap виявляє збійну ситуацію у роботі предикату за яким слідкує, повертає код помилки (ціле число) і передає керування предикату, що оброблює збійну ситуацію. При виявленні збійної ситуації trap повертає неправду. Предикат trap повертає істину, якщо предикат, за яким він слідкує відпрацював без помилки, після чого виконується наступний предикат програми. Процедури, аналогічні предикату trap, є у різних мовах, наприклад у імперативній мові Фортран.

Предикат обробки збою пише програміст. Предикат виводить тип помилки, виконує необхідні дії і може передати керування предикату для повторення дій, під час яких виявилася помилка. Якщо виконання програми надалі неможливо, то предикат обробки може закінчити роботу програми. Доступ до повідомлення за кодом помилки забезпечує стандартний предикат errormsg.

errormsg (ім'я_файлу_помилок, код_помилки, повідомлення, допомога)

(i, i, o, o)

(string, integer, string, string)

Файл помилок – це стандартний файл Prolog. err або файл користувача з кодами та повідомленнями про можливі помилки виконання. Для виконання програми під керуванням утиліти TestGoal

файл помилок треба розмістити в каталозі проекту *obj*. Для роботи програми за функцією Run файл помилок розміщується в каталозі *exe*.

Код помилки – це ідентифікатор помилки у файлі помилок. Повідомлення – це повідомлення про помилку, що вибирається з файлу помилок за кодом помилки. Рядки допомоги показують, в яких випадках може з'являтися помилка або що треба зробити при помилці. Вони розміщуються у наступних рядках файлу помилок після рядка з кодом і повідомленням. Вся група, що характеризує помилку закінчується керуючим символом «#» в окремому рядку. Рядок допомоги може бути відсутнім.

Наприклад,
7002 File not found
#

Користувач може зробити свій файл повідомлень про помилки, вказавши будь-яке ім'я та розширення *err*. Наприклад, ім'ям файлу помилок може бути: *користувач*. *err*. Вміст файлу має бути оформлений, як показано у прикладі нижче і відповідати проекту.

7002 Файл БД не знайдений
БД не існує.
#

Текст повідомлення та допомоги можна писати українською мовою, а коди повинні відповідати кодам файлу Prolog. err.

При застосуванні файлу помилок користувача стандартний файл помилок Prolog. err треба вилучити. Якщо застосовувати свій файл помилок без обробки збою (без предикату trap), то виконавча система буде видавати повідомлення, що файл Prolog. err відсутній. Після того виконавча система знайде файл повідомлень користувача і виведе його повідомлення за кодом помилки та допомогу.

Наприклад:

1032 No error messages Native OS error
Ділення на нуль
Одержане або введене невірне значення дільника.

Краще застосовувати файл Prolog. err попередньо коригуючи його під програми проекту. При коректуванні файлу Prolog. err

повідомлення «1032 No error messages Native OS error» не виводиться. Коригування файлу Prolog. err не зашкодить роботі інших проектів, бо кожний проект має свої файли Prolog. err у каталогах *obj* або e*xe*.

Розглянемо програму із застосуванням предикату trap і обробкою збою стандартного предикату. Робота програми продовжується після виправлення помилки.

Задача №1. Прочитати вміст файлу, що містить прізвища письменників і вивести їх на екран. Ім'я файлу ввести з клавіатури.

Файл з прізвищами письменників знаходиться у каталогу проекту *exe*.

Вміст файлу:

Володимир_Винниченко
Леся_Українка
Маруся_Чурай
Олександр_Олесь
Олесь_Гончар
Остап_Вишня

Обробка помилок залежить від кодів помилок. Застосуємо предикат trap для контролю предикату openread. Для обробки вказаної помилки зроблено припущення, що файл має інше ім'я. Тому предикат обробки помилки запропонує ввести інше ім'я. Для цього у файлі Prolog. err має бути попередньо відкориговане українською мовою повідомлення з кодом 7002.

7002 Файл не знайдено.
Файл з вказаним ім'ям не існує у поточному каталогу або шлях до каталогу невірний
#

Назвемо проект з програмою Trap1. Для роботи програми додамо до секції Global Domains файлу Trap1. inc символічне ім'я файлу *g*. Вміст файлу Trap1. inc додається в програму за директивою компілятора *include*.

```
include «Trap1. inc»
Global predicates
Nondeterm читати()
```



```
Nondeterm обробити(integer)
Nondeterm вивести()

clauses
читати():-eof(g),!.
читати():-readln(Письменник), write (Письменник),!, nl, читати().

обробити(Cod):-errormsg(«Prolog. err», Cod, Msg, Hlp),
        write (Cod, '\n', Msg, '\n', Hlp, '\n'), !, вивести.

вивести:-write(«Введіть назву файлу>«), readln(Назва),
        trap(openread(g, Назва), Cod, обробити(Cod)),
        readdevice(g), читати(), closefile(g).

goal
вивести.
```

Нижче показаний приклад роботи програми при введені ім'я файлу, що не існує в каталозі.

Введіть назву файлу > *a. txt*

7002 Файл не знайдено. Файл з вказаним ім'ям не існує у поточному каталогу або шлях до каталогу невірний

Введіть назву файлу > *c. txt*

Володимир_Винниченко
Леся_Українка
Маруся_Чурай
Олександр_Олесь
Олесь_Гончар
Остап_Вишня

Перервати роботу програми можна за клавішами Ctrl-break. Предикатом trap можна контролювати роботу предикату користувача, якщо в ньому застосовано кілька стандартних предикатів. Але краще застосовувати предикат trap для кожного стандартного предикату. Такий підхід дає гарантію, що кожна оброблена збійна ситуація не перерве роботу програми.

Розглянемо приклад задачі, при розв'язуванні якої контролюється кілька стандартних предикатів.

Задача №2. Створити файл, що містить прізвища авторів та назви їх книжок. Ім'я файлу повинно мати 5 символів.

Програма запитує ім'я файлу і контролює його на кількість символів. Якщо файл з таким іменем вже існує у проекті, то пропонується змінити ім'я файлу, інакше програма закінчує свою роботу. При відсутності файлу з вказаним ім'ям програма вводить прізвища авторів та їх книги і зберігає їх у файлі. Для файлу встановлюється атрибут «тільки читати».

```
% Include trap2. inc
Global Facts
single файл(string)

predicates
Nondeterm створити()
Nondeterm читати()
          писати(string)
Nondeterm обробка(integer)

goal
створити(),!; true.

clauses
файл(«»).

Створити():-write(«Введіть назву файлу»),
        readln(Назва), nl, assert(файл(Назва)),
        trap(subchar(Назва, 5, _), Cod, обробка(Cod)),
        trap(openwrite(g, Назва), Cod1, обробка(Cod1)),
        читати(), closefile(g).

читати():-write(«Введіть прізвище автора (ESC закінчити увід)»),
        readln(Прізвище), писати(Прізвище),!, читати().

читати():- файл(Назва),
          fileattrib(Назва, 65).

писати (Прізвище):-write(«Введіть назву книги>»),
       readln(Книга), writedevice(g),
       write(Прізвище, «», Книга), nl, writedevice(screen).
```



```
обробка(7005):- файл(Назва),
       write («7005 Файл», Назва,
       «захищений від запису»),
       write(«Створити файл з іншою назвою?»),
       write(«(Enter-так, ESC – ні)»),
       readchar(C), nl, C=13, nl,!, створити(); true.

обробка(1601):- errormsg(«Prolog. err», 1601, Msg, Hlp),
       write(Msg,  '\n', Hlp, '\n'), nl, створити().
```

Розглянемо способи контролю стандартних предикатів поданих у програмі. Програма контролює кількість символів у імені файлу предикатом subchar. Ім'я файлу повинно мати п'ять символів – одна буква у імені файлу, точка та розширення з трьох букв. Предикат subchar одержує з рядку символ за індексом. Якщо індекс більше ніж довжина імені файлу, то предикат закінчує роботу за помилкою «1601 Помилка індексу рядку».

Предикат trap контролює роботу предикату subchar. При виявленні помилки предикат повертає код помилки і передає керування предикату користувача – обробка (1601). Предикат обробка(1601) застосовує стандартний предикат errormsg, який обирає за кодом помилки в файлі Prolog. err повідомлення та рядок допомоги. Після виводу повідомлень керування передається на повторний увід назви файлу.

Другий спосіб контролю збійної ситуації показано при контролю відкриття файлу. При відкритті файлу для запису з атрибутом «тільки читати» видається повідомлення «7005 Доступ заперечений». Предикат обробки помилки обробка (7005) не застосовує предикат errormsg і файл Prolog. err. Предикат виводить конкретне повідомлення про збій з вказівкою коду помилки та імені файлу. Програма може ввести інше ім'я файлу або закінчити роботу за бажанням користувача.

Для забезпечення збереження даних створеного файлу застосовується стандартний предикат fileattrib.

fileattrib (Назва_файлу, Код_атрибуту)
(i, i)
(i, o)
(string, integer)

Предикат може встановлювати або повертати атрибути файлу. Предикат не працює з папками.

Атрибути:

- тільки для читання – 65;
- архівний – 96;
- системний – 68;
- прихований – 66.

За замовченням код файлу 96.

Треба пам'ятати, що для контролю арифметичного виразу його застосовують повністю: застосовують весь вираз S1=S/N.

```
Predicates
do()
obr(integer)
Clauses
do:-S=20000, readint(N),
trap(S1=S/N, Cod, obr(Cod)),
write(S1).
obr(Cod):-write(Cod, «Ділення на 0»).
Goal
do.
```

Для полегшення контролю деяких предикатів вводу застосовуються спеціальні предикати доступу до інформації про розташування помилки.

Універсальний стандартний предикат вводу readterm може застосовувати допоміжний стандартний предикат readtermerror, що полегшує знаходження помилки при читанні із файлу структур або даних простих типів. При обробці збою предикат readtermerror виводить рядок файлу, у якому виявлена помилка та позицію помилкового символу.

Розглянемо програму із застосуванням предикатів trap, readterm та readtermerror.

Задача №3. Даний файл із структур, де кожна структура містить відомості про деталі, що виробляє цех. Вивести на екран відомості про деталі.

Вміст файлу detal. txt:

```
деталь(11003, «Болт»)
деталь(11004, «Гайка»)
деталь(11005, «Гвинт»)% помилка – пропущені лапки
```

```
% include «detal. Inc»
Global domains
file = f
d = деталь(шифр, назва)
шифр = integer
назва = string
Predicates
збій(integer)
Nondeterm вивід()
Clauses
Вивід():-eof(f).
Вивід():-trap(readterm(d, L), Cod, збій(Cod)),
write(L), nl, !, вивід.
Вивід().
збій(Cod):- nl, readtermerror(Line, Pos), write(Cod, «Помилка у рядку»,
Line, «позиція», Pos).
Goal
openread(f, «detal. txt»), readdevice(f), nl, вивід, closefile(f).
```

Програма контролює читання структур з файлу і виводить їх вміст на екран. При виявленні помилки у структурі видається повідомлення і програма закінчує роботу.

Результат роботи:

```
деталь(11003, «Болт»)
деталь(11004, «Гайка»)

1409 Помилка у рядку «деталь(11005, «Гвинт»)» позиція 20
```

Розглянемо ще приклад обробки збійної ситуації при вводі фактів з файлу. Звичайно програмні системи застосовують такі файли великого обсягу, тому при створенні файлів користувач робить

помилки. Знаходження фактів із помилками важка робота, особливо при наявності у фактах великої кількості аргументів.

Предикат trap може виявити факти з помилками при завантаженні фактів предикатом consult. При обробці збійної ситуації для локалізації помилки застосуємо предикат consulterror. Предикат consulterror виводить рядок з помилкою, номер невірного символу у рядку та номер невірного символу від початку файлу.

consulterror (Рядок, Позиція_в_рядку, Позиція_в_файлі)
(*o, o, o*)
(string, integer, ulong)

Розглянемо приклад програми з контролем вводу фактів із файлу.

Задача №4. Даний файл з набором фактів, що містять адреси студентів групи. Вивести на екран прізвища студентів та їх адреси.

Нехай файл *adres. txt* має наступний вміст.

адреса(«Петренко», «Запоріжжя» «Леніна»,24.3)
адреса(«Сидоров», «Запоріжжя», «Театральна»,12.9)
адрес («Іваненко», «Запоріжжя», «Шкільна»,45,58)
адреса(«Кім», «Запоріжжя», «Гоголя»,1.2)

У третьому факту функтор із помилкою. Програма вводить поточний факт і контролює його. Якщо факт вірний, то програма продовжує вводити і контролювати факти. При помилці у факті видається повідомлення з кодом помилки і місцем її розташування. Після чого запитується, чи бажає користувач коригувати факт. Якщо користувач збирається коригувати помилку, то викликається текстовий редактор «Блокнот». Після коригування факту з помилкою факти завантажуються і їх вміст виводиться на екран.

```
% include «consult_err. inc»

Global Domains
прізвище, місто, вулиця = string
дім, квартира = integer

Global Facts - adr
адреса(прізвище, місто, вулиця, дім, квартира)

predicates
```



```
Nondeterm завантажити()
Nondeterm вивід()
Nondeterm обробка(integer)
Nondeterm перезапис()

Goal
завантажити(),!; true.

Clauses
завантажити():-trap(consult(«adres. Txt», adr), Cod,
обробка(Cod)), вивід().

обробка(Cod):-consulterror(Рядок, Символ, _),
write(Cod, «Помилка в факті:», Рядок, «Символ №», Символ, '\n', «Для
роботи з файлом натисніть Enter»),
write(«(ESC - вихід)\n»), readchar(C), C=13, перезапис(),!,
завантажити(); true.

перезапис():- system(«C:\\Windows\\System32\\notepad. exe, adres. txt»).

вивід():- адреса(Прізвище, Місто, Вулиця, Дім, Квартира),
write(Прізвище,' ', Місто, ' ', Вулиця, ' ', Дім, ' ', Квартира, '\n'), fail.
вивід().
```

З програми мовою Visual Prolog можна викликати будь-яку зовнішню програму. Для цього застосовується стандартний предикат system.

system (Шлях_до_зовнішньої_програми)
(i)
(string)

Для виклику програми потрібно знати шлях до зовнішньої програми. Можна також встановлювати змінну оточування для додавання шляху до системних каталогів, в яких файл шукається за замовченням.

Виклик текстового редактору notepad. exe подається разом з іменем файлу, що дозволяє відразу відкрити файл. Для створення нового файлу виклик подають без імені файлу. Після коригування факту користувач зберігає файл і виходить з редактору. Керування передається програмі користувача.

При виклику зовнішньої програми можливі наступні помилки.

1116 Невдача в «системі» виклику.
1117 Помилка при виконанні зовнішньої програми.

Виклик предикату system з порожнім рядком передає керування командному інтерпретатору ОС. Після чого можна застосовувати команду.

Наприклад:

```
Goal
system(«»).
```

Одержати склад певного каталогу можна за командою DIR. Повернутися до програми за командою EXIT. Вміст вікна *cmd*. exe подано нижче.

```
D:\Primer Visual Prolog> dir     G:\Progr
Том у пристрої G має мітку
Серійний № тому

Вміст папки G:\Progr

28.03.2012      16:07              423      Trap1. pro
28.03.2012      16:27              866      Trap2. pro
                2 файли            1289 байтів

D:\Primer Visual Prolog>exit
```

Для переривання роботи програми у виключних ситуаціях можна застосовувати предикат exit. До виключних ситуацій відносять ситуації, у яких неможливо продовжувати роботу програми. До такої ситуації можна віднести увід невірного паролю.

Задача №5. Написати процедуру, що контролює паролі користувачів системи «Тестування».

```
Global Facts
single код(integer)
Nondeterm пароль (string, string).

Predicates
Nondeterm контроль()
```



```
Nondeterm увід(string, string)

Goal
контроль (); True.

Clauses
пароль («w2r15», «Коваленко»).
пароль («a463y», «Сімонов»).
пароль («h344d», «Євтушенко»).

код (0).

контроль ():-write(«Пароль?»), assert(код (1)),
увід («», S), пароль (S, Прізвище), assert(код (0)),
write(«\n*Тестування*», Прізвище), nl,!.

контроль ():- код (C), write(«\nКод =», C,
«\n Невірний пароль \n»), exit(C),!.
увід (Str, S):- Str_len(Str, N), N<5,
readchar(C1), write('*'),
str_char(S1, C1),
concat(Str, S1, Str2), увід (Str2, S).

увід (S, S).
```

Програма аналізує вміст паролю. За невірним паролем видається код помилки 1 і повідомлення «Невірний пароль». Предикат exit(1) повертає код виконуючій системі Visual Prolog. Код помилки можна повертати іншій програмі.

Предикат exit() можна викликати без аргументів. Такий запис тотожній запису exit(0).

Резюме

Мова Visual Prolog має засоби, що забезпечують сталість роботи програми. Пролог дозволяє оброблювати як помилки роботи програми користувача, так і помилкові звертання до програм ОС, допоміжних пакетів доступу до сучасних технологій БД та мережі.

Основним предикатом, контролюючим роботу стандартних предикатів і предикатів користувача, є предикат Trap. Предикат виявляє збійні ситуації і дозволяє обробити збійну ситуацію, щоб продовжити роботу програми. Для виявлення помилок можна застосовувати стандартний файл помилок Prolog. err або файл помилок користувача.

В Пролозі існують стандартні предикати, які дозволяють користувачу вивести повідомлення і локалізувати місце помилки. До таких предикатів відносять errormsg, readtermerror, consulterror.

Особливо важливо локалізувати помилки у файлах з фактів та складних структур. Звичайно такі файли великих обсягів.

Продовжити роботу програми можна коригуючи файли з помилками без виходу з програми. Для цього застосовують зовнішні текстові редактори, які можна викликати стандартним предикатом *system*.

Предикат fileattrib також дозволяє стабілізувати роботу програми. Предикат захищає дані файлу від їх знищення.

Завершити програму при неможливості продовжити роботу програми можна за предикатом exit, який повертає код помилки.

Контрольні питання

1. Як застосовують предикат контролю помилок trap?
2. У якому файлі зберігаються повідомлення про помилки?
3. Де розташовується файл помилок і чи можна його коригувати українською мовою?
4. Описати структуру запису для кожного коду помилки в файлі помилок.
5. Як створити свій файл помилок?
6. Як прочитати атрибути певного файлу, або змінити їх?
7. Для чого застосовується стандартний предикат readtermerror?
8. Як використовується стандартний предикат CONSULTERROR?
9. Як можна звернутися до текстового редактору із програми користувача під час її роботи?

Вправи

1. Написати програму, що знаходить продуктивність підприємства за цехами і загалом. Контролювати арифметичне переповнення. Закінчуйте програму зберіганням фактів і предикатом exit.
2. Написати програму, яка читає файл за вказаним шляхом до нього. При відсутності файлу у вказаному каталогу створити його не виходячи з програми.
3. Написати програму, що контролює увід фактів з файлу, що містить такі відомості про робітників заводу: прізвище, адресу проживання, табельний номер, місце роботи.

2.3 Засоби систем традиційного програмування у системі програмування Visual Prolog

Виправлення помилок в програмі займає багато часу. Щоб знайти помилку її намагаються локалізувати, тобто виявити відрізок програми, на якому може бути помилка. Помилки можуть бути явними, які просто знаходяться. В той же час можуть зустрітися кілька помилок, які вкупі дають такий результат, що здогадатися де знаходяться помилки важко.

На відміну від компіляторів мов традиційного програмування компілятор мови Visual Prolog моделює процес виконання програми і тим самим виявляє більше помилок виконання. Крім того, можна впливати на генерацію коду програми компілятором для різного рівня слідкування за помилками виконання: minimal, medium, maximal. Компілятор вставляє в програму користувача коди процедур, що слідкують за помилками.

Для налагодження програм Visual Prolog має програму відладчик. Чим зручніше інструмент налагодження і чим більше у нього можливостей, тим швидше можна налагодити програму. Відладчик Visual Prolog є потужнім інструментом налагодження програми і ні в чому не поступається відладчикам інших мов програмування. Він може налагоджувати застосування 32-розрядного Windows (Win32) в залежності від установок опції Target: VPI, EasyWin і Textmode.

Відладчик Visual Prolog – можна застосовувати як з ОС, так і з VDE (на відміну від старого відладчика PDC Prolog, який був частиною середовища розробки і тому його можна було запускати тільки з середовища).

2.3.1 Опції компілятору для налагодження програми

Щоб відладчик міг одержувати поточні результати роботи програми та слідкувати за її виконанням, перед компіляцією треба в головному меню середовища Visual Prolog послідовно обрати опції Options/Project/ Compile Options і на вкладці компілятора Output обрати опції Generate Browser Information і Generate Debug Information, (рис. 2.1).

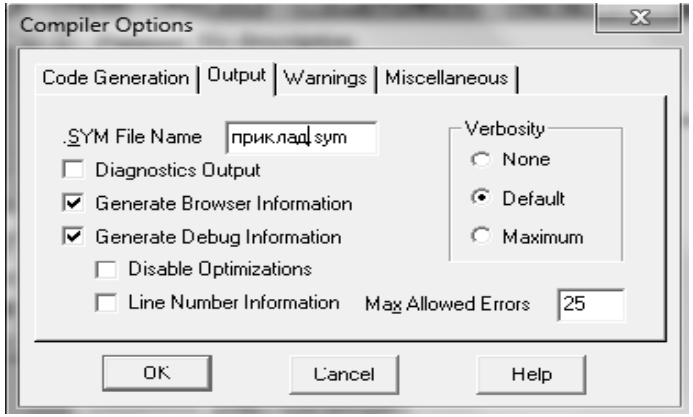


Рисунок 2.1 – Генерація налагоджувальної інформації

Включення прапорця Generate Debug Information змусить компілятор генерувати файли з розширенням DEB в каталозі OBJ. Ці файли використовуються відладчиком для позиціонування вихідного коду, змінних, фактів і т. д. під час виконання програми.

Включення прапорця Disable Optimizations блокує оптимізацію хвостової рекурсії, тобто рекурсія не буде заміщатися на ітерацію. Програма, яка відкомпільована з оптимізацією хвостової рекурсії, не буде показувати всі пункти у вікні відладчика Call Stack. Однак при включенні прапорця Disable Optimizations програма не буде заміщати хвостову рекурсію на ітерацію, і значно збільшиться ризик переповнення стека виконуваної програми.

Для активізації процесу налагодження з VDE просто виберіть команду Project | Debug. Також можна використовувати комбінацію клавіш <Ctrl> + <Shift> + <F9> або кнопку Debug панелі інструментів. Спочатку буде побудований (якщо необхідно) проект, потім буде викликано цільове застосування і активізований відладчик Visual Prolog на це застосування. Також можна запустити відладчик з операційної системи. Відладчик Visual Prolog – це виконуваний файл, шлях до якого – BIN \ WIN \ 32 \ VIPDEBUG. EXE. Відладчик може запускатися під 32-розрядною Windows (Windows 95/98/ME, NT/2000 і Windows) для налагодження програм, для користувача інтерфейсу 32-розрядної Windows і консольних застосувань.

2.3.2 Завантаження програми

Після виклику відладчика Debugger з операційної системи автоматично з'являється вікно завантаження програми у відладчик (рис. 2.2).

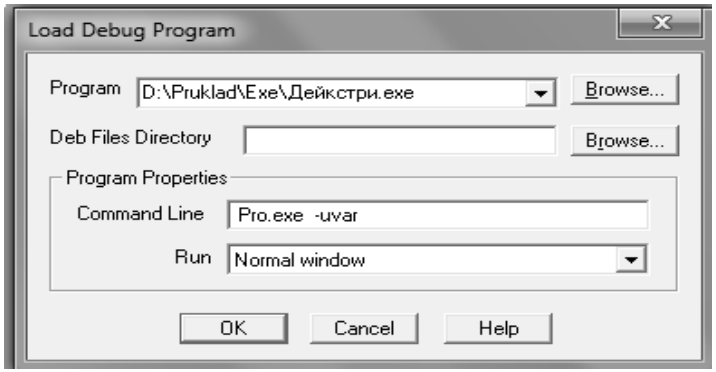


Рисунок 2.2 – Вікно завантаження програми у відладчик

У полі Program автоматично з'являється шлях до останньої виконуваної програми, з якою працював користувач.

Список опцій у полі Run дозволяє обрати режим запуску програми: в повному, мінімізованому або максимізованому вікні.

У полі Command Line потрібно задати командний рядок для компілятора. Він вміщує опції для компіляції програми та імена файлів.

Командний рядок задається у формі:

PRO. EXE [–option[{+|–|value[{+|–}] }]] file…,

де програма PRO. EXE розширює можливості DOS – дозволяє передавати опції компілятору та імена файлів для компіляції у командному рядку.

У квадратних дужках вказують опції компілятора, у фігурних дужках через риску вказують, що треба обрати одне з вказаних значень опції. Знак мінус виключає опцію або значення опції, знак + повертає виключену опцію або її значення.

Наприклад, опція -uvar означає: не видавати попередження про змінні, що не можуть бути конкретизовані в процесі роботи програми.

Якщо командний рядок використовується для компіляції і компоновки, то необхідно враховувати наступні аспекти:

• файли з розширеннями *sym, map, obj* і *deb* повинні бути розташовані в одному каталозі перед компонуванням. Спочатку ці файли можуть розміщуватися в різних каталогах;

• можуть використовуватися компонувальники Microsoft і Borland, але в цьому випадку не гарантується, що всі імена будуть бачитися з відладчика Visual Prolog.

Якщо переміщуються файли з розширенням *deb*, то необхідно визначити їх нове місце розташування у полі Deb Files Directory. За кнопками Browser можна змінити програму поточного проекту для налагодження або знайти каталог для файлів з розширенням *deb*.

Для завантаження виконуваної програми у відладчик можна також обрати команду Load в меню Files або обрати піктограму для завантаження програми, з'явиться теж саме вікно.

Коли програма завантажена у відладчик, то стартова точка та інша інформація (які модулі завантажені, які з них мають налагоджувальну інформацію і т. д.) можуть бути переглянуті у вікні Messages в нижній частині екрана.

Оберіть кирилицю за опцією Font в меню Edit. Програму можна запустити за допомогою меню Run або кнопок на панелі інструментів.

2.3.3 Меню Run

Меню Run використовується для запуску та керування виконанням програми у відладчика.

Меню містить такі команди:

• Run. Команда (або клавіша <F9>) запускає програму. Вона зазвичай використовується у поєднанні з установкою точок зупину.

• Run Until Fail. Ця команда виконує код від поточного предиката до першої відмови. Курсор буде розміщений на предикаті, який викликається після цієї відмови.

• Break Program. Якщо програма запущена у відладчику, то можна її перервати, щоб увійти знову у відладчик за іншим режимом. Наприклад, можна перевірити стек викликів для перегляду того, що програма робить; або встановити точку зупину у тому місці, де потрібно зупинити процес виконання.

• Wait for VPI Events. Відладчик чекає першу подію VPI, потім розміщує і відображує курсор на першому твердженні предикату обробника подій, який отримав подію VPI.

• Trace Into. Команда (або клавіша <F7>) використовується для покрокового руху по вихідному коду із заходом в усі предикати, що викликаються.

• Step Over (або клавіша <F8>). Покроковий рух по вихідному коду без заходу в предикати, які викликаються. По досяганні виклику предиката він виконується як проста команда.

• Run to Cursor. Команда (або клавіша <F4>) працює як при встановлення точки зупину в поточній позиції вихідного коду, виконуючи команду Run. Для виконання команди треба розмістити курсор у потрібне місце у вихідному коді і виконати цю команду.

• Invoke Fail. Команда негайно викликає відмову (подібно виклику предиката fail). Цю команду можна використати у випадку, якщо потрібно виконати наступні гілки процедури після виконання поточної або виконати іншу процедуру.

• Invoke Exit. Команда імітує виклик предиката EXIT(0). Нагадаємо, що стандартний предикат EXIT(0) завершує виконання програми з кодом нормальне завершення 0.

• Restart. Команда перезапускає програму.

Меню Run надано на рис. 2.3. Команди меню згруповані за призначенням: ініціювання, трасування, завершення.

Рисунок 2.3 – Меню Run

2.3.4 Вікна перегляду(View)

У відладчику можна відкрити кілька вікон. Вони відкриваються з меню перегляду View (рис. 2.4). Розглянемо докладніше вікна меню перегляду і роботу з ними.

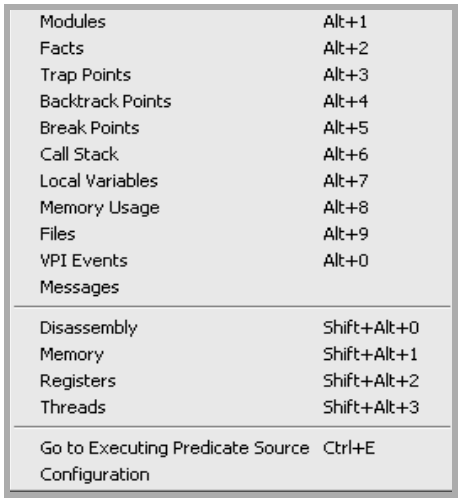


Рисунок 2.4 – Меню перегляду

Вікно модулів(Modules)

Якщо відладчик запущений і програма завантажена, то після запуску на виконання вона відображається у вікні модулів, а структура файлів може бути переглянута за допомогою дерева, яке зображене на рис. 2.5.

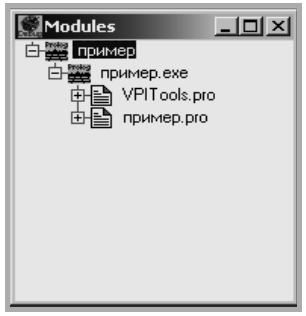


Рисунок 2.5 – Вікно модулів

У цьому вікні можна двічі клацнути на файлах, з яких складається проект, і двічі клацнути на окремих предикатах. В результаті відкриється вікно вихідного коду з курсором, встановленим в обрану позицію.

Щоб побачити вихідний код, програма має бути написана на Visual Prolog. Всі інші програми можна переглядати на асемблері. Не можна редагувати код у відладчику PDC, тому що він автономний. Найбільш зручний спосіб виправляти помилки – це знайти відповідні предикати у відладчику і у вихідному коді з використанням VDE.

Вікна вихідного коду(Go to Executing Predicate Source)

У вікні вихідного коду можна переглядати крок за кроком, як виконується програма. Команда меню View|Go to Executing Predicate Source (або комбінація клавіш <Ctrl> + <E>) встановлює курсор на рядок, що містить предикат, який виконується. Код буде виділятися білим кольором в тих місцях, де можливо встановити точки зупину. Вікно вихідного коду може бути відкрито майже з будь-якого пункту меню у відладчику (рис. 2.6).

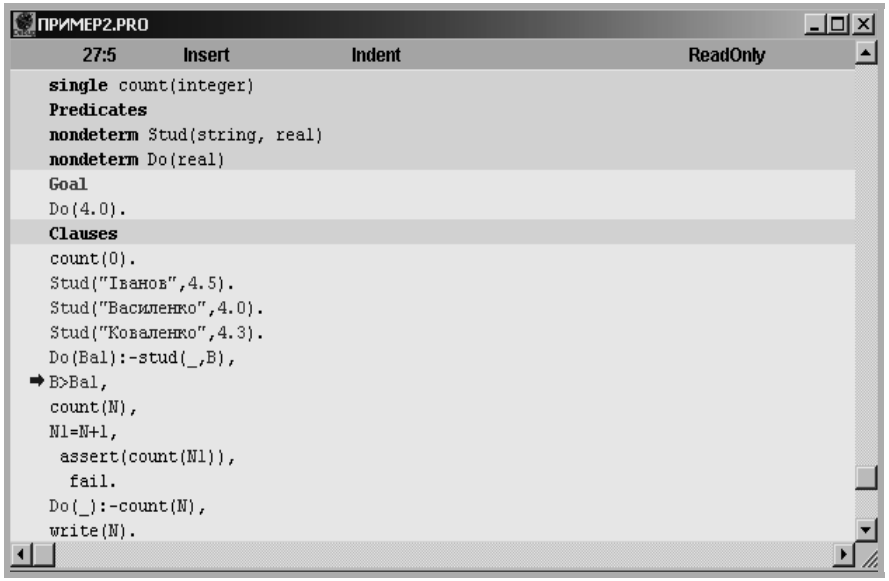


Рисунок 2.6 – Вікно вихідного коду

У діалоговому вікні View|Debugger Configuration можна вимкнути опцію «відтінити вихідний код» (прапорець Gray Source Lines not Containing Executable Code).

У вікнах вихідного коду можна використовувати кілька команд, які активізуються з меню Edit або при натисканні правої кнопки миші. Існують можливості пошуку тексту (команда Search), переміщення на певну позицію або рядок (команди Go to Line Number і Go to Position відповідно), переходу до оголошення або до речення зазначеного предиката (команди Go to Declaration і Go to Clause відповідно). У діалоговому вікні View|Debugger Configuration можна вимкнути опцію «відтінити вихідний код» (прапорець Gray Source Lines not Containing Executable Code).

У вікнах вихідного коду можна використовувати кілька команд, які активізуються з меню Edit або при натисканні правої кнопки миші. Існують можливості пошуку тексту (команда Search), переміщення на певну позицію або рядок (команди Go to Line Number і Go to Position відповідно), переходу до оголошення або до речення зазначеного предиката (команди Go to Declaration і Go to Clause відповідно).

Можна копіювати виділений вихідний код в буфер обміну: вибрати необхідний фрагмент, переміщаючи мишу з натиснутою лівою клавішею, і використати команду Copy.

Команда Show Position відображає абсолютну позицію, зазначену мишею в коді. З будь-якої позиції можна перейти до виконуваного предикату (команда Go to Executing Predicate). За допомогою команди Put Position on Clipboard for in VDE можна записати позицію, зазначену мишею, в буфер обміну. Надалі в VDE можна використовувати комбінацію клавіш *<Alt> + <F2>*, щоб перейти до цієї позиції.

Установка точок зупину(Break Points)

Установка точок зупину (Break Points) дуже важлива для ізоляції помилок або перевірки коректності виконання фрагмента програми. За допомогою вікна Break Points можна набагато зручніше і ефективніше управляти точками зупину (рис. 2.7). Вікно викликається командою Break Points з меню View.

Рисунок 2.7 – Вікно точок зупину

При відкритому вікні вихідного коду, на рядку можна клацнути правою кнопкою миші, після чого відкриється меню, звідки може бути встановлена точка зупину. Якщо вибрана команда Run (гаряча клавіша <F9>), то програма виконається тільки до цієї точки.

У списку будуть перелічені всі точки зупину, їх стан (включена або виключена). Точка зупину може бути видалена за допомогою команд Remove та Remove All. Стан точки зупину перемикається командою Toggle. Всі команди можуть бути виконані із спливаючого меню або з меню Edit. За допомогою команди Go to Code можна перейти до файлу вихідного коду з встановленою точкою зупину. Вибором команди Properties, активізується діалогове вікно Breakpoint State, яке показано на рис. 2.8.

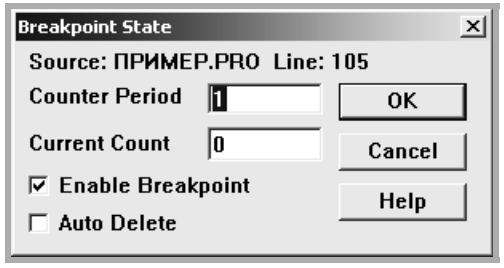


Рисунок 2.8 – Властивості точки зупину

Лічильник періоду (Counter Period). Якщо точка зупину встановлена в циклі (наприклад, в рекурсивному предикаті), то програмний додаток буде повертатися до цієї точки тисячі разів.

Якщо не треба зупинятися на кожному циклі, то можна визначити періодичність (інтервал), з якою відладчик буде зупинятися

в цій точці. Для цього визначте необхідне значення P_Value періоду для цієї точки в полі Counter Period. Відладчик пов'язує внутрішній лічильник з кожною заданою точкою зупину. При кожному проході через цю точку відладчик додає 1 до відповідного лічильнику. Поточне значення лічильника відображено в полі Current Count.

За замовчуванням значення Counter Period дорівнює 1; це означає, що відладчик повинен зупинятися при кожному попаданні на цю точку. Якщо для лічильника періоду встановлено різне значення P_Value, то відладчик зупиниться тільки на значенні лічильника, кратному вказаною P_Value.

Значення лічильника Current Count відображає поточне значення лічильника. Це значення дорівнює числу повторюваних попадань на точку зупину під час виконання додатка.

Опція Enable Breakpoint означає дозволити точку зупину. Скинувши цей прапорець, ви можете тимчасово відключити точку зупину без її видалення. Пізніше точку зупину знову можна включити.

Якщо прапорець автоматично видаляється (Auto Delete), то точка зупину автоматично буде видалена після першого попадання в неї.

Вікно змінних (Variables For Current Clause)

Під час виконання можна переглядати значення змінних (рис. 2.9). Для цього потрібно виконати команду View|Local Variables. Коли курсор миші вказує на деяке твердження у вихідному коді, то через коротку паузу можна побачити значення змінних у твердженні. Це дуже зручний спосіб для перевірки значення змінної без відкривання вікна змінних.

Рисунок 2.9 – Вікно змінних

Вікно фактів (Facts)

При виправленні помилок програм дуже важливо гарантувати, щоб були додані коректні факти. Факти програми можуть бути переглянуті у вікнах фактів (рис. 2.10). Вікно вибирається командою Facts в меню View.

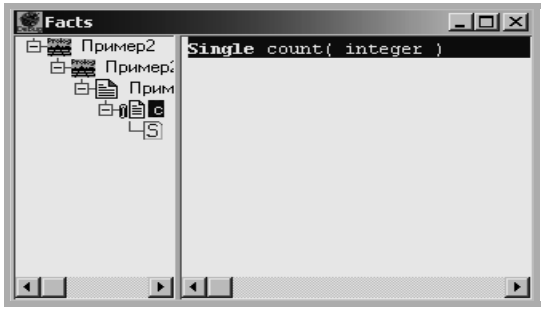


Рисунок 2.10 – Вікно фактів

Стек викликів

Вікно стека викликів (рис. 2.11) показує список предикатів, які вже були викликані при налагоджені програми. Подвійне клацання на предикаті показує код, де стався виклик обраного предиката. Якщо були зроблені зміни, вікна можна оновити командою Refresh із спливаючого меню.

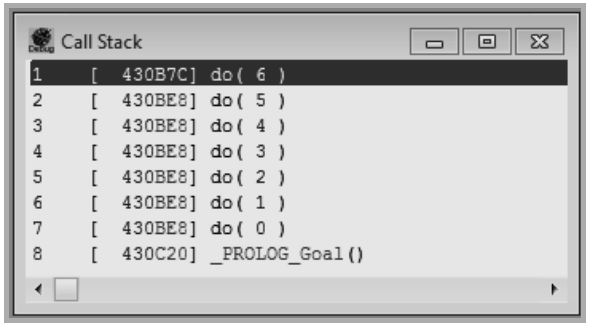


Рисунок 2.11 – Представлення стеку викликів

У вікні 2.11 показано приклад вмісту стеку для рекурсивної процедури, що немає граничної умови:

```
Predicates
роби(integer)

Clauses
роби(N):-N1=N+1, write(N1), nl, роби(N1).

Goal
роби(0).
```

Вікно точок відкату

Вікно точок відкату показує список точок відкату налагоджуваної програми. Подвійне клацання на виділеній точці відкату встановлює курсор на рядок коду, де знаходиться точка відкату.

На рис. 2.12 наведено список точок відкату для програми поданої нижче. Під час конкретизації змінних предикату stud є ще одна точка відкату для предикату stud і одна точка відкату для предикату do.

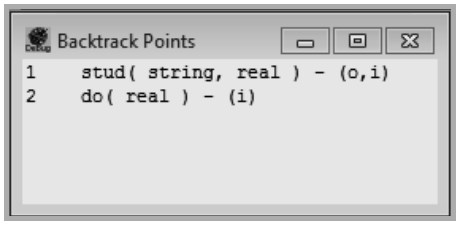


Рисунок 2.12 – Вікно Backtrack points

```
Predicates
Nondeterm do(real)
Nondeterm stud(string, real)

Clauses
stud(«Коваль», 4.5).
stud(«Швець», 3.6).
stud(«Рибалко», 4.5).

do(Bal):-stud(Pr, Bal), write(Pr, «», Bal), nl, fail.
do(_).
Goal
do(4.5).
```


Подання фрагментів програми на машинному рівні

У вікнах меню View можна переглядати коди програми асемблером (Disassembly), вміст регістрів (Registers), дамп оперативної пам’яті зайняті фрагментами програми (Memory) і потоки даних (Threads). Основною метою цих дій є перевірка коду, написаного не на мові Пролог. В той же час дамп часто застосовують щоб глибше зрозуміти внутрішню роботу тієї чи іншої програми, внутрішню структуру даних.

Вікно перегляду кодів програми асемблером показано на рис. 2.13.

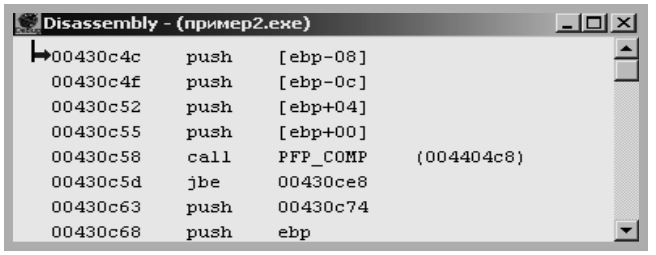


Рисунок 2.13 – Вікно перегляду кодів асемблера

Вікно перегляду дампу пам’яті показано на рис. 2.14

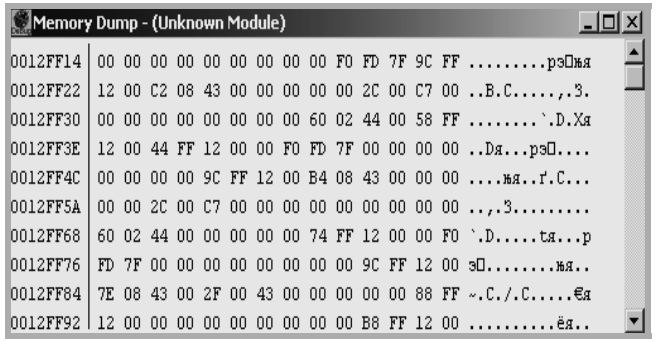


Рисунок 2.14 – Вікно дампу пам’яті

Найбільш зручний спосіб переміщатися у вікні пам’яті – явне визначення адрес у функціях Goto спливаючого меню (або підменю Edit). Команда Goto Address приймає адреси трьох форматів:

- пряме подання шістнадцятирічної адреси (наприклад, 012FEF4);
- імена регістрів. Наприклад, *esp*. Якщо ім’я регістра визначено, то область перегляду прокручується на адресу, що відповідає значенню, збереженому у вказаному регістрі;

• глобальні об’єктні імена (об’єктні імена глобальних предикатів). Область перегляду прокручується на адресу, що відповідає цьому імені.

• предикат Goto Dword Prt включається, коли встановлений прапорець Show as Dword (показувати як подвійне слово), і курсор вказує на номер, який може бути прийнятий як адреса формату домену dword. У цьому випадку Goto dword Prt змінюється на Goto «address». Ця функція прокручує область перегляду на address.

Регістри

Вікно перегляду вмісту машинних регістрів показано на рис. 2.15.

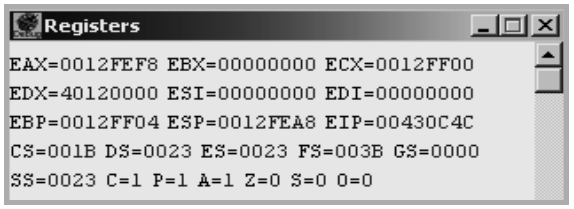


Рисунок 2.15 – Вікно машинних регістрів

2.3.5 Опції налаштування відладчику

Вікно опцій налаштування Debugger Configuration можна відкрити за допомогою команди меню View | Configuration.

Опції налаштування відладчику є глобальними (Global) і зберігаються у файлі *vipdebug. ini*.

На рис. 2.16 подано вікно вибору опцій відладчику.

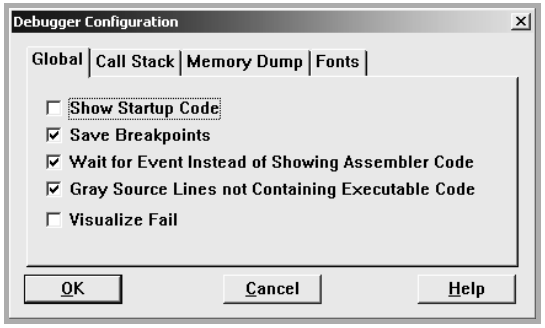


Рисунок 2.16 – Опції налаштування відладчику

Глобальні опції налаштування відладчику

Show Startup Code – дозволяє налагоджувати код ініціалізації програми (на рівні асемблера), який виконується перед викликом goal-програми.

Save Breakpoints – дозволяє зберігати точки зупину для поточного проекту в ініціалізованому файлі відладчику *vipdebug. ini*.

Після зміни або перекомпіляції модуля деякі, або всі точки зупину, задані в цьому модулі, можуть бути втрачені.

Коли відладчик відкриває новий проект, він видаляє всю збережену в файлі ініціалізації інформацію про раніше налагоджені проекти. Тому інформація про точки зупину віддаляється, коли відладчик відкриває інший проект.

Wait for Event Instead of Showing Assembler Code – забороняє посилання на код асемблера і відображення самого коду асемблера. Замість цього відладчик чекає першу подію, отриману програмою, що налагоджується, і переходить до обробника цієї події.

Gray Source Lines not Containing Executable Code – зафарбовує рядки коду програми сірим кольором, якщо рядки не містять виконуваного коду. Точки зупину не можуть бути встановлені на таких рядках.

Visualize Fail – викликає явне відображення неуспішного виклику виконуваного предиката. Перед таким викликом відображується маркер F–>.

Опції стеку викликів

Можна визначити параметри вікна Call Stack.

Update – група перемикачів, яка містить наступні елементи:

• *Manual* – вікно Call Stack оновлюється тільки при натисканні клавіші <F5>;

• *Smart* – вікно Call Stack оновлюється кожного разу, коли змінюється регістр;

• *Always* – вікно Call Stack оновлюється при кожному кроці відладчика.

Оновлення вікна Call Stack може вимагати значних обчислень, які сповільнюють виконання відладчику. Наприклад, якщо параметри предикатів – великі списки значень і т. п.

Scanning Call Stack from Goal – вікно Call Stack буде показувати виклики лише тих предикатів, які викликаються після виклику мети goal програми.

Show Module Name – імена предикатів будуть випереджатися іменами відповідних вихідних модулів.

Show Argument Domains – будуть відображатися домени аргументів предикатів.

Опції дампу пам'яті

У вікні Update Window every <NN> Sec можна визначити, що вікно Memory Dump повинно оновлюватися кожні NN секунд.

Fonts

Опція дозволяє змінювати шрифти для вікон відладчика Visual Prolog.

Source Win – обирається шрифт для відображення початкового коду.

Messages – обирається шрифт для вікна повідомлень Messages.

Other Win – обирається шрифт для всіх інших вікон. Рекомендується використовувати шрифти тільки з фіксованою шириною.

Зміна шляхів до вихідних файлів

Діалогове вікно зміни шляхів до вихідних файлів може бути відкрито з меню Files. У цьому вікні є можливість тимчасово визначити різні імена підкаталогів, що містять вихідні файли проекту.

Наприклад, запустити відладчик із програмою, вихідні файли якої знаходяться в мережі. Інформація, визначена в цьому діалоговому вікні, зберігається у файлі ініціалізації відладчику VIPDEBUG. INI.

Коли відладчик відкриває проект, він видаляє всю інформацію про інші проекти, збережені у файлі ініціалізації відладчику. Тому інформація про змінені шляхи до вихідних файлів проекту стирається, як тільки відладчик відкриває інший проект.

2.3.6 Приклад використання відладчику

Розглянемо роботу відладчику на прикладі. Покажемо трасування на прикладі програми, що виконує завдання №1.

Завдання №1. Даний набір фактів, які мають наступні відомості про книги: прізвище автору, назву книги і рік видання. Знайти кількість книг по кожному автору, а також найбільш продуктивного автора. Трасування програми відображує покроковий процес виконання програми. Тому розташуємо кожен предикат програми у окремому рядку.


```
Facts – бд
single лічильник (integer)
Nondeterm книга(string, string, integer)
single макс(string, integer)

predicates
Nondeterm рахувати(string)
Nondeterm знайти_макс(string)

Goal
книга(Автор, _, _),
рахувати(Автор),
assert(лічильник(0), бд),
retractall (книга(Автор, _, _), бд),
fail;

макс(Автор, Значення),
write(«Найпродуктивніший автор–» , Автор, « («, Значення,» книги)»).

clauses
лічильник(0).
макс(«», 0).

книга(«Льюїс Керролл», «Символічна логіка», 1958).
книга(«Льюїс Керролл», «Аліса в країні див», 1960).
книга(«Джон фон Нейман», «Комп'ютер і мозок», 1958).

рахувати(Автор) :-
    книга(Автор, _, _),
    лічильник(К),
    К1=К+2,
    assert (лічильник(К1), бд),
    fail.

рахувати(Автор):-
    лічильник(К),
    write(Автор, «–», К, «книги»), nl,
    знайти_макс (Автор), !.

знайти_макс(Автор) :-
    макс(_, Значення),
    лічильник(К),
```

К>Значення,
assert(макс(Автор, К), бд); True.

Нехай нам необхідно знайти помилку у процедурі рахувати(Автор).

Встановимо точку зупину на рядку з цим предикатом. Для цього встановимо курсор на рядок з предикатом рахувати(Автор) і натиснемо клавішу <Enter>. Ініціювання програми виконаємо за опцією Run|Run. Програма виконається до точки зупину автоматично. Щоб розглянути докладно роботу процедури продовжимо роботу програми покрокова <F7>. Для слідкування за зміною вмісту факту лічильник(К) викличемо вікно View|Facts. Розкриємо вміст БД з фактами програми(рис. 2.17).

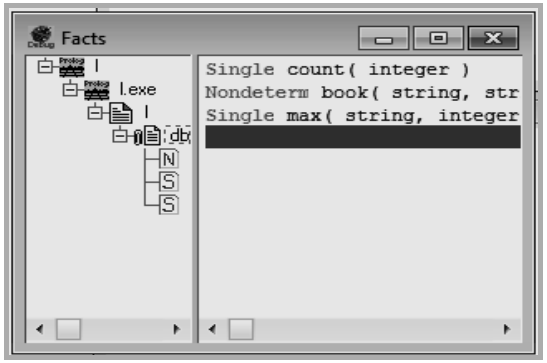


Рисунок 2.17 – Факти програми

Обравши букву *N* (Nondeterm) можна побачити вміст фактів з відношенням книга, обравши *S* (Single) можна побачити поточний вміст динамічних фактів лічильник (К) і макс(Значення).

У нашій процедурі лічильник фактів – лічильник (К), який зберігає кількість фактів за прізвищем автору, замість 1 нарощується на 2. Помилка виявляється під час зміни вмісту факту лічильник (К). Після знаходження помилки можна продовжити роботу програми за гарячими клавішами <F7> або <F8>, можна перервати роботу програми за опцією Run| Break Program, перейти до певного місця в програмі або перезапустити її тощо.

Резюме

Відладчик Visual Prolog є окремою програмою, яку можна застосовувати, як із середовища, так і окремо від середовища. Відладчик дозволяє настроюватися на роботу певного проекту і певної програми проекту. Він також дозволяє динамічно настроювати та застосовувати певні інструменти налагодження.

Відладчик має велику кількість інструментів налагодження програм, як загального характеру (типових для більшості відладчиків різних мов програмування), так і специфічні для Visual Prolog.

До типових інструментів можна віднести: трасування програми по вихідному коду із заходом в предикати, що викликаються, або без заходу; трасування програми на рівні асемблеру; установку точок зупину; включення обмежень для зупину при циклічних діях; перегляд стеку та вмісту змінних; рух за програмою до місця розташування курсору та дампування пам’яті програми.

До специфічних інструментів Visual Prolog можна віднести: роботу з фактами; роботу з точками відкату; можливість виконувати програму до першої невірної цілі; можливість викликати у певному місці програми невірний результат, як у предикаті fail.

Контрольні питання

1. Як локалізувати помилку? трасувати програму?
2. Які опції компілятору необхідно обрати для застосування відладчику?
3. Як ініціювати відладчик: а) з системи? б) з Visual Prolog?
4. Якщо у проекті кілька окремих програм, як завантажити одну з них у відладчик?
5. Як включити трасування програми після її завантаження у відладчик?
6. Чим відрізняється трасування за клавішею <F7> від трасування за клавішею < F8>.
7. Як виконати трасування різними способами до певного місця у програмі?
8. Як перервати трасування програми або перезапустити його?
9. Для чого застосовують вікно Break Points?

10. Як одержати результати трасування тільки на певних рівнях виклику рекурсії?

11. Для чого застосовують вікно змінних?

12. Як можна спостерігати зміну значень фактів?

13. Для чого можна подивитися вміст стеку?

14. Що таке дамп пам'яті?

Вправи

1. За допомогою відладчику знайдіть помилки в програмах, що реалізують нижче подане завдання:

Завдання. Дані відомості про пасажирів літака. Вони містять номер місця пасажира та вагу його речей. Отримати загальну вагу речей пасажирів.

Програма №1.

```
Predicates
пасажир(integer, real)
заг_вага(integer, real, real)

clauses

пасажир(1,5.1).
пасажир(2,4.0).
пасажир(3,12.5).
пасажир(4,9.4).
пасажир(5,7.9).

заг_вага(Номер, Сума_н, Сума_к):-
        пасажир(Номер, Вага),
        Сума = Сума_н + Вага,!,
        заг_вага(Номер, Сума, Сума_к).
заг_вага(_, Сума_к, Сума_к).

goal
заг_вага(1,0.0, Сума_к), write (Сума_к).
```

Програма №2.

```
facts
заг_сума(real)
```



```
Predicates
Nondeterm пасажир(integer, real)
Nondeterm заг_вага

clauses

заг_сума(0.0).

пасажир(1,5.1).
пасажир(2,4.0).
пасажир(3,12.5).
пасажир(4,9.4).
пасажир(5,7.9).

заг_вага:-
        пасажир(_, Вага),
        заг_сума(С),
        Сума = С + Вага,
        assert(заг_сума(Сума)),
        fail.

заг_вага:-заг_сума(Сума_к), write(Сума_к).

goal
заг_вага.
```

2. За допомогою відладчику знайдіть помилки в програмі:

Завдання: Ввести рядок з клавіатури. Одержати з даного рядка новий, слова в якому записані в зворотному порядку, а порядок слів той же.

```
Predicates
замінити(string, string, string)
замінити_порядок_букв(string, string, string)

Clauses
замінити(«», S, Sn):-
        frontchar(S,_, Sn),!.

замінити(S, S1, Sn):-
        fronttoken(S, W, Ost),
        замінити_порядок_букв(W, «», Wn),
```

```
        concat(S1, «», SS),
        concat(SS, Wn, S2),!,
        замінити(SS, S2, Sn).
замінити_порядок_букв(«», S, S):-!.
замінити_порядок_букв(W, S1, Wn):-
        frontchar(W, C, O),
        frontchar(S, C, S1),!,
        замінити_порядок_букв(O, S, Wn).
Goal
readln(S),
замінити(S, «», Sn),
write(Sn).
```


РОЗДІЛ ІІІ. МЕХАНІЗМИ МОВИ ПРОЛОГ ДО РОЗВ'ЯЗУВАННЯ ЗАДАЧ ШТУЧНОГО ІНТЕЛЕКТУ

Інтелектуальні системи призначені для розв'язку задач, що не формалізуються, а саме, задач, що не розв'язуються на основі точних знань, або для яких формалізація невідома. Такі задачі вимагають символічних обчислень, алгоритмічний розв'язок їх невідомий, ціль задачі не може визначатись цільовою функцією.

Неформалізованим задачам властива неповнота знань, неоднозначність розв'язку, суперечність у знаннях. У таких задачах не шукається оптимальне рішення, а береться перше рішення, що підходить користувачу. Вказаний тип задач називають задачами штучного інтелекту.

До задач штучного інтелекту відносять задачі навчання та самонавчання систем; обробки символьної інформації, машинної творчості; планувань поведінки роботів, інтерпретації даних у поняття ПДО, прогнозування подій, класифікації об'єктів, керування складними об'єктами, підтримки прийняття рішень у складних ситуаціях і т. п..

Створення програми мовою Visual Prolog для розв'язку задачі штучного інтелекту значно полегшує роботу програміста. Вказаним властивостям Visual Prolog зобов'язаний процедурі зіставлення із зразком, методам пошуку вглиб та вшир, механізму звороту, які забезпечують перебір варіантів.

3.1 Пошук даних за зразком

Механізми Visual Prolog дозволяють просто і компактно розв'язувати задачі, що працюють з великою кількістю даних, вимагають швидкого пошуку та перебору даних за багатьма ключами. Причому ключі для відбору можна динамічно змінювати, залишаючи саму програму без змін.

Для більшості задач штучного інтелекту пошук даних за зразком є основною операцією на знаннях, бо саме задачі штучного інтелекту характеризуються неповнотою та суперечністю вихідних даних, неоднозначністю розв'язків.

Розглянемо докладніше пошук даних із використанням процедури зіставлення зі зразком. Покажемо перевагу методу перед застосуванням *процедурного аналізу даних*.

Задача №1. Розробити програму «Телефонний довідник». Кожний запис про абонента повинен містити наступну інформацію: прізвище, ім'я, по батькові, вулицю, дім, квартиру, № телефону. Програма повинна знаходити № телефону та дані про абонента за будь-яким ключем або усякими комбінаціями ключів. Ключем може бути будь-який компонент запису, крім номера квартири і телефону. Якщо за ключами знаходиться група користувачів, то виводяться дані для всієї групи.

В задачі об'явлено п'ять ключів. Розв'язування задачі вимагає

$2^5-1 = 31$ варіантів ключів. Реалізація у тексті програми *процедурного аналізу даних* вимагає реалізації 31 умови, тобто всього простору варіантів. Така процедура буде достатньо великою.

Програма мовою Пролог реалізує зіставлення із зразком шляхом створення зразка. Під зразком розуміють набір аргументів цілі, які задаються змінними. Зразок можна застосовувати для будь-яких варіантів ключів. Конкретизуючи ті чи інші змінні ми кожен раз оберемо тільки один варіант.

Ключі для пошуку телефону можуть задаватися прізвищем (неповні вхідні дані) або прізвищем, ім'ям та по батькові і адресою або адресою(повною або неповною). Вихідні дані можуть бути суперечними, наприклад є два Петренко Петра Івановича, а адреси в них різні. Запит за вулицею одержує кілька розв'язків.

Реалізуємо програму, що розв'язує задачу, мовою Visual Prolog. Компоненти довідника розташуємо у файлі *Tel. txt*.

```
абонент(«Петренко», «Петро», «Іванович», «Шкільна»,
«8», «1», «2674534»)
абонент(«Василенко», «Ірина», «Іванівна», «Київська»,
«15», «2», «2634671»)
абонент(«Петренко», «Петро», «Іванович», «Шорника»,
«4», «5», «2956723»)
абонент(«Варягін», «Остап», «Сергійович», «Козацька»,
«1», «3», «2659081»)
абонент(«Петрова», «Ольга», «Василівна», «Київська»,
«3а», «9», «2457825»)
```

Програма завантажує телефонний довідник з фактами, одержує з цілі певну комбінацію ключів і відбирає за ними факт про абонента.

Компоненти фактів форматуються за однаковою довжиною і виводяться на екран. Дії продовжуються до вичерпання фактів.

```
Global Facts
Nondeterm абонент(string, string, string, string, string, string, string)

Predicates
Nondeterm довідка(string, string, string, string, string)
Nondeterm табл(string, string)

Clauses
довідка(Пр, Ім, Бт, Вул, Дім):-
        абонент(Пр, Ім, Бт, Вул, Дім, Кв, Тел),
        табл(Пр, Пр2), табл(Ім, Ім2), табл(Бт, Бт2), табл(Вул, Вул2),
        табл(Дім, Дім2), табл(Кв, Кв2 ), табл(Тел, Тел2 ), write(Пр2, «»,
        Ім2,

        «», Бт2, «», Вул2, «», Дім2, «», Кв2,

        «», Тел2), nl, fail.

табл(Комп, Комп2):- str_len(Комп, Довж),
        Кільк_проп=14-Довж,
        str_len(Проп, Кільк_проп),
        concat(Комп, Проп, Комп2 ),!;
        str_len(Комп, Довж), Довж>14, substring(Комп, 1,14, Комп2);
        Комп2=Комп.


Goal
consult(«Tel. txt»), Пр = «Петренко»,
Вул = «Шорника», довідка(Пр, Ім, Бт, Вул, Дім), nl, !; True.
```

Ключі відбору факту вказані у цілі конкретизованими змінними. За комбінацію ключів, можна одержати один або багато результатів. За вказаною у програмі ціллю можна одержати результат:

Петренко Петро Іванович Шорника 4 5 2956723

За ціллю:

```
Goal
consult(«Tel. txt»), Вул = «Київська», довідка(Пр, Ім, Бт, Вул, Дім), nl, !; True.
```

Можна одержати відомості:

Василенко Ірина Іванівна Київська 15 2 2634671
Петрова Ольга Василівна Київська 3а 9 2457825

Програма форматує компоненти для виводу. Кожний компонент доповнюється такою кількістю пропусків, щоб всі вони були однакової довжини.

Задача №2. Доповнимо умову задачі №1 вимогами:
1. Ключі для відбору компонент вводити з клавіатури.
2. Запис шукати за частиною ключа.

Програма опитує ключі для пошуку, формує за ключами зразок для пошуку. Зчитує поточний факт про абонента і перевіряє чи задовольняють його компоненти зразку, тобто накладає зразок на факт. Якщо компоненти відповідають зразку, то дані виводяться, інакше факт пропускається. Після чого зчитується наступний факт.

```
Domains
str = reference string

Global Facts
Nondeterm абонент(string, string, string, string, string, string, string)

Predicates
Nondeterm опит()
Nondeterm ключі(string, str)
Nondeterm частина(str, string, string)
Nondeterm довідка(str, str, str, str, str)
Nondeterm табл(string, string)

Clauses
опит():-write(«Ключі відбору? (ENTER-пропустити)»),
        nl, write(«Прізвище?»), readln(Пр),
        ключі(Пр, Пр1),
        write(«Ім’я?»), readln(Ім), ключі(Ім, Ім1),
        write(«По батькові?»), readln(Бт), ключі(Бт, Бт1),
        write(«Вулиця?»), readln(Вул), ключі(Вул, Вул1),
        write(«Дім?»), readln(Дім), ключі(Дім,Дім1),
        довідка(Пр1, Ім1, Бт1, Вул1, Дім1).
```



```
ключі(Ключ, Ключ2):- Ключ<> «», Ключ2=Ключ, !; True.

довідка(Пр1, Ім1, Бт1, Вул1, Дім1):-
        абонент(Пр, Ім, Бт, Вул,
        Дім, Кв, Тел),
        частина(Пр1, Пр, Пр11),
        частина(Ім1, Ім, Ім11),
        частина(Бт1, Бт, Бт11),
        частина(Вул1, Вул, Вул11),
        частина(Дім1, Дім, Дім11),
        табл(Пр11, Пр2), табл(Ім11, Ім2),
        табл(Бт11, Бт2), табл(Вул11, Вул2),
        табл(Дім11, Дім2), табл(Кв, Кв2 ),
        табл(Тел, Тел2 ),
        write(Пр2, «», Ім2, «», Бт2, «», Вул2,
        «», Дім2, «», Кв2, «», Тел2), nl, fail.

частина(Ключ, Ключ1, Ключ2):-Bound(Ключ),
        concat(Ключ,_, Ключ1), Ключ2=Ключ1, !;
        not(Bound(Ключ)), Ключ2=Ключ1, !;
        Bound(Ключ), fail.

табл(Комп, Комп2):-str_len (Комп, Довж),
        Кільк_проп=14-Довж,
        str_len (Проп, Кільк_проп),
        concat(Комп, Проп, Комп2 ),!;
        str_len(Комп, Довж), Довж>14,
        substring(Комп, 1,14, Комп2);
        Комп2=Комп.

Goal
consult(«Tel. txt»), опит(), !; True.
```

Предикат опит () послідовно опитує значення ключів. Вводиться повне значення ключа або будь-яка частина значення, яка є його початком. Для пропуску вводу значення ключа треба нажати клавішу

ENTER. Із значень ключів та вільних змінних для ключів формується зразок пошуку для предикату довідка.

У мові Visual Prolog змінна, якій не привласнюється значення автоматично вважається областю, на яку посилаються. Тип string відноситься до основних типів даних. Тим самим основний тип стає типом для посилання. У таких випадках при виклику функцій мовою C++ виникає помилка. Щоб уникнути помилки для предикатів, що працюють з вільними змінними вводиться тип даного *посилання*. Таким типом в програмі є тип *str*. Предикати ключі, довідка, частина застосовують тип *str*.

Предикат довідка зчитує поточний факт з даними про абонента і перевіряє чи відповідає факт зразку. Перевірка виконується предикатом «частина», який застосовують до кожного аргументу факту і відповідного ключа із зразку.

Предикат частина перевіряє чи конкретизований поточний ключ із зразку. Якщо ключ конкретизований то предикат порівнює його значення із значенням аргументу факту. Порівняння виконується стандартним предикатом *concat*. Значенням ключа може бути частина аргументу факту або весь аргумент. Предикат *concat* може виконувати порівняння в обох випадках. При вірному порівнянні керування передається в предикат довідка і перевіряється наступний аргумент факту, при невірному порівнянні керування передається в предикат довідка і зчитується наступний факт. Якщо ключ зразку неконкретизований, то для виводу приймається значення аргументу факту і перевіряється наступний аргумент факту.

По успішному закінченню обробки всіх аргументів факту, він форматується і виводиться на екран. Зчитується наступний факт. Обробка продовжується до вичерпання фактів.

Резюме

Пролог значно спрощує пошук та перебір даних завдяки процедурі зіставлення із зразком. Для роботи з такими даними у тексті програми не треба фіксувати певні ключі пошуку, бо процедура зіставлення із зразком дозволяє застосовувати будь-яку комбінацію даних у якості ключів.

Часто при вводі значень ключів з клавіатури потрібно ввести значення тільки деяких ключів. Таким чином змінні для інших ключів

у зразку залишаються вільними. Щоб Пролог не вважав таку ситуацію за помилкову, для предикатів, що працюють з вільними змінними, вводиться тип даного посилання.

Контрольні питання

1. Як реалізується процедурний метод пошуку даних?
2. Як створюється зразок для пошуку даних?
3. Описати роботу процедури пошуку за зразком.
4. Навіщо вводяться типи даних для посилання?
5. Для чого у другому прикладі вводиться тип даних для посилання?
6. Чому Visual Prolog не дозволяє основним стандартним типам об’являтися типами для посилання?
7. Як об’явити тип для посилання?

Вправи

1. Написати програму, яка вводить і зберігає у файлі факти про студентів: прізвище та ім’я по батькові, спеціальність, № групи та місце проживання. Передбачити, що студенти можуть проживати у обласному місті, районному місці або селі.
2. Написати програму, яка вибирає дані з файлу, створеному у завданні 1 і виводить на екран списки студентів за різними комбінаціями ключів.

3.2 Динамічне створення алгоритмів механізмами Прологу

Однією з основних властивостей системи штучного інтелекту є можливість створювати алгоритми розв’язування задач, узагальнювати їх та модифікувати для настроювання на конкретну задачу.

Програма мовою Пролог – це опис задачі, який керує виконанням механізмів Прологу. Тим самим динамічно створюється алгоритм розв’язування задачі.

Виконання програми мовою Пролог – це доведення істинності цілі програми. Ціллю будь-якої програми мовою Пролог є стандартний предикат Goal, що дозволяє однаково розпочинати роботу всіх програм. Предикат Goal є умовним твердженням, а його умовами є ціль програми, яку вказує користувач. У предиката Goal може бути кілька гілок умов, як у умовних тверджень користувача.

Виявлення істинності цілі програми реалізується через декомпозицію цілі на поточні цілі, які в свою чергу декомпозуються. Процес виконується до тих пір поки для всіх кінцевих цілей не з'ясується їх істинність, а через них виявиться істинність поточних цілей та цілі програми.

По суті, програма на Пролозі є деревом цілей, обхід якого виконується механізмами Прологу «пошук вшир», «пошук вглиб» та механізм звороту. А знаходження зіставимого із зразком твердження, знаходиться механізмом прямого трасування та процедурою уніфікації.

Програміст може керувати обходом дерева, застосовуючи стандартні предикати відсік (!), true, fail; предикати, що вставляють та вилучають динамічні факти (предикати групи assert, предикати retract та retractall). Важливим для керування обходом дерева є поняття детермінізму предикатів (їх характеристики), його вплив на предикати роботи з динамічними фактами та на механізми Прологу.

Завдяки вбудованим механізмам Прологу програма користувача в десять разів менше програми, що створюється на імперативній мові за тим же завданням. В той же час завдяки оптимізованому компілятору швидкість роботи програми мовою Visual Prolog порівнювана із швидкістю роботи мови C^{++}. Задачі, розв'язування яких вимагає перебору великої кількості варіантів рішень, розв'язуються мовою Visual Prolog значно легше, ніж на імперативних мовах. Розглянемо сказане на прикладах програм, що знаходять шлях у лабіринту та знаходять найкоротший шлях у орієнтованому графі.

3.2.1 Пошук в лабіринті

Серед спеціалістів штучного інтелекту популярна задача пошуку шляху у лабіринтах. Спеціалісти застосовують лабіринти, як полігон для навчання роботів-машинок знаходженню шляху в будь-якому лабіринті. Конструктори комп'ютерів вважають, що така робота необхідна для створення машин, які навчаються самі. Щоб з'ясувати, чому саме на лабіринтах навчають роботів, треба зрозуміти, що поняття лабіринту можна трактувати як множину ситуацій у певній ПДО. Робот повинен знайти шлях від ситуації, в якій він знаходиться до ситуації - цілі. Тобто пошук шляху в лабіринті можна розуміти, як універсальний метод розв'язування певної задачі.

Програмна реалізація різних методів пошуку цілі в лабіринті вимагає реалізації механізмів перебору даних з великою кількістю

зворотів. Необхідно також мати великі масиви даних про структуру лабіринту. Реалізовувати такі програми імперативними мовами не зручно, текст програми виходить великий. Механізми Прологу, зберігання даних у динамічних фактах, дозволяють значно зменшити об'єм програми, що моделює процес навчання роботів.

Щоб розглянути докладніше поняття лабіринту звернемося до міфів та історичних фактів. У стародавньому грецькому міфі розповідається про боротьбу Тезея з Мінотавром у критському лабіринті. Аріадна допомогла вийти Тезею з лабіринту. Вона дала йому клубок ниток, щоб знайти зворотний шлях. У XX столітті англійський археолог Артур Еванс дійсно знайшов лабіринт при розкопках на острові Крит. За легендою його побудував афінський майстер Дедал. Насправді лабіринт було побудовано ще у 1400 році до н. е., а його призначенням було, як вважають археологи, захоронення мертвих. Лабіринт критського типу з семи кіл зображено на рис. 3.1. Цифрами зображено шлях входу до центру лабіринту та виходу з лабіринту.

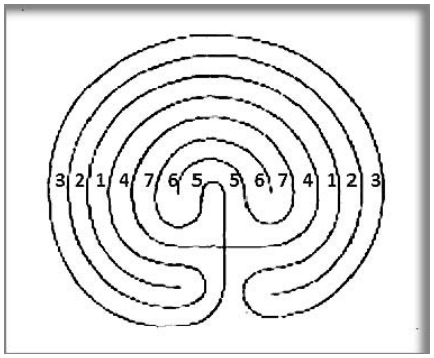


Рисунок 3.1 – Лабіринт критського типу з сьома колами

Як ми бачили на малюнку, лабіринти типу критського не мали гілок і не перетиналися. Лабіринт мав один вхід, а доріжка вела до центру – тупика. Більше тупиків лабіринт не мав. Досягнувши тупика людина поверталася тим же шляхом. У своїй книзі [5] німецький дослідник лабіринтів Герман Керн показав способи створення графічних лабіринтів та різні інтерпретації змісту таких лабіринтів.

За думкою Керна сім кіл критського лабіринту відображує світогляд стародавніх людей. Однією з інтерпретацій є життя, смерть,

існування у підземному світі та відродження людини. Лабіринт відокремлений від зовнішнього простору нагадує життя окремої людини. Вхід у лабіринт можна розглядати як її народження, а досягнення цілі як смерть. Велика кількість поворотів у лабіринті вказує на різні події у житті людини, велику втрату часу та душевну втому. Декілька разів доріжка майже приводить її до центру, але тут же відводить у протилежний бік. Довгий шлях викликає в людини велику психічну напругу, а відсутність можливості вибору шляху посилює напругу. Все ж таки людина неодмінно досягне цілі. Рух за одною доріжкою без альтернатив відображує необхідність слідувати за законами природи. У центрі людині відкривається щось таке важливе, що відкриття вимагає кардинальної зміні напряму руху. Людина повертається назад по своїх слідах, але це вже людина, що перероджується для нового рівня існування.

Застосовуючи різні варіанти правил можна створювати схеми різних типів лабіринтів. Розглянемо правила створення лабіринтів критського типу. Знайомство з правилами побудови лабіринтів важливо для більш глибокого розуміння топології лабіринтів.

Основою лабіринту є центральний хрест. Між сторонами хреста малюють чотири прямих кута. В середині кожного кута ставлять крапку. Після чого треба з'єднати верхівку хреста з вищою точкою вертикальної сторони верхнього лівого кута. З точки в цьому прямому куті проводять в протилежному напрямку криву лінію – дугу, що з'єднує точку з вершиною вертикальної сторони кута вгорі праворуч. З точки в цьому прямому куті проводять сполучну криву до кінця горизонтальної сторони кута угорі ліворуч (рис. 3.2). Повторюючи дії для інших кутів, ви одержите рисунок критського лабіринту з семи кіл.

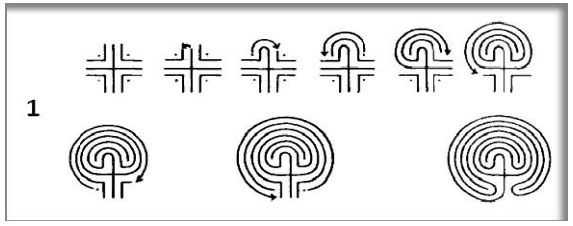


Рисунок 3.2 – Схема побудови лабіринту критського типу із одним прямим кутом

Якщо в кожному з чотирьох секторів, утворених центральним хрестом, розташувати не по одному, а по два прямих кута (рис. 3.3), кількість кіл стане більше на чотири. Кожній додатковій четвірці прямих кутів відповідають чотири додаткові кола.

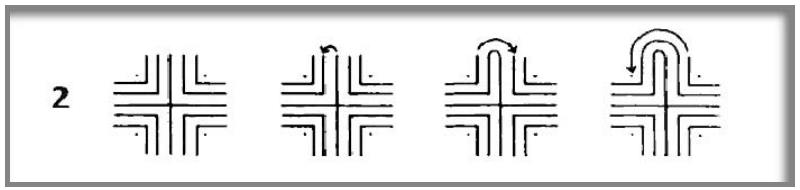


Рисунок 3.3 – Схема побудови лабіринту критського типу із двома прямими кутами

Варіанти лабіринтів виникають, коли точки з'єднуються в іншому порядку, або коли основою застосовують не хрест, а іншу фігуру (наприклад *Y)*. Кути можна додавати тільки зверху або тільки знизу хреста (рис. 3.4). Такі лабіринти зустрічаються в Скандинавії.

Треба розуміти, що форма лабіринту бачиться тільки зверху. Для людини, що йде лабіринтом, форма прямокутного або колоподібного лабіринту неістотна. Важливий лише напрям руху та структура лабіринту.

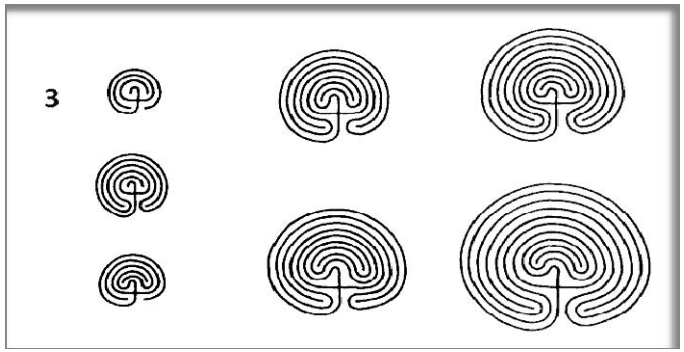


Рисунок 3.4 – Схема побудови лабіринтів із Скандинавії

Два лабіринти вважать однаковими, якщо їх шляхи від входу до цілі є однаковими, незалежно від форми лабіринту. Кажуть, що шлях в лабіринту є топологічним інваріантом.

Згідно з Мартином Гарднером [6], якщо намалювати план лабіринту на резині та розтягувати резину, то незалежно від форми ліній шлях від входу до цілі залишиться тим же. Тому *при створенні програми, що малює лабіринт, можна застосовувати будь-яку форму лабіринту*.

Розглянуті типи лабіринтів відносяться до лабіринтів, що мають один вхід, шлях по одній доріжці до цілі в середині лабіринту, а вихід виконується тим же шляхом.

Серед грецьких орнаментів часто зустрічаються орнаменти, що складаються з повторення фігури подвійний меандр. Цікавий факт, що схема критського лабіринту створюється поворотом схеми подвійного меандру. На рис. 3.5 показано процес перетворення подвійного меандру у критський лабіринт. Цікаво розробити програму, що демонструє процес перетворення подвійного меандру у критський лабіринт.

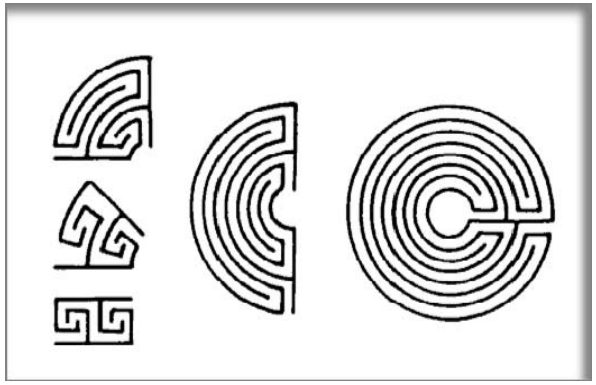


Рисунок 3.5 – Процес перетворення подвійного меандру в критський лабіринт

З XV – століття одержали розповсюдження будівлі, живі огорожі, у формі лабіринтів-плутанок. Лабіринтами-плутанками з мозаїки прикрашали поли залів у палацах.

Лабіринти – плутанки мали багато розгалужень з тупиками. До таких лабіринтів можна віднести лабіринт із живих огорож в Англії у літній резиденції Вільгельма Оранського. Лабіринт розташовано на березі Темзи в 14 кілометрах від Лондона. Сьогодні туристи, що відвідують палац Оранського також шукають шлях до центру лабіринту і назад.

Лабіринт називають однозв’язним, якщо на його схемі всі лінії пов’язані в єдине ціле. Кажуть, що лабіринт багато зв’язний, якщо у його схемі є лінії не пов’язані з іншими. Лабіринт Вільгельма Оранського відноситься до багато зв’язних лабіринтів(рис. 3.6). Незв’язані стіни на рис. 3.6 виділені червоним кольором.

Рисунок 3.6 – План лабіринту із живих огорож при дворі Вільгельму Оранського

Існує метод автоматичного пошуку шляху в лабіринті – «метод руки». Метод працює лише при умові, що людина «не відірве руку від стіни під час всього шляху за лабіринтом». «Метод руки» працює тільки для лабіринтів, стіни яких пов’язані в єдине ціле. Загубитися в таких лабіринтах, застосовуючи «метод руки» неможливо.

Якщо лабіринт має замкнені шляхи (з тупиком), то людина застосовуючи «метод руки» завжди повернеться в те місце звідки увійшла в замкнений шлях. Ціль, що лежить на замкненому шляху, буде осягнена. Поодиноким випадком такого лабіринту є критський.

В той же час, якщо не застосовувати «метод руки», шлях може бути знайдено навіть для лабіринтів, що мають багато зв’язків. Для цього треба, щоб не пов’язані лінії не оточували ціль. Перевірте на рис. 3.6 можливість пройти до центру у багато зв’язному лабіринті.

Існує універсальний метод проходження за лабіринтом – метод Е. Люка-Тремо. Опишемо лабіринт при дворі Вільгельма Оранського множиною коридорів. Будемо вважати, що коридор кінчається там, де з’являється розгалуження шляхів. Для кожного коридору вкажемо його індивідуальний номер, список номерів коридорів, в які можна пройти з фіксованого коридору і наявність цілі в цьому коридорі.

За методом Люка треба йти по поточному коридору до розгалуження шляхів, обрати один з можливих коридорів і дослідити його. Якщо він веде в тупик, то треба повернутися до розгалуження і дослідити інший можливий коридор. По закінченню можливих шляхів в цьому розгалуженні треба повернутися до попереднього розгалуження і повторити дії. Таким чином можна знайти центр в лабіринті зображеному на рис. 3.6.

Метод Люка-Тремо легко реалізується мовою Visual Prolog.

Задача. Написати програму, що знаходить у великому будинку – лабіринту кімнату з телефоном. Пошук треба виконати методом Люка-Тремо (рис. 3.7).

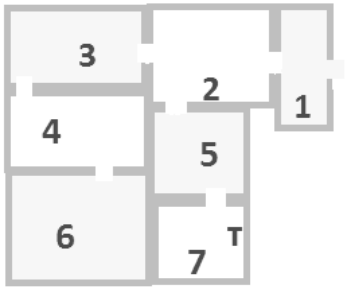


Рисунок 3.7 – План будинку – лабіринту

Подамо план будинку фактами, аргументи яких означають відповідно номер поточної кімнати; номер кімнати до якої є вхід з поточної кімнати; наявність телефону у поточній кімнаті. Кожна кімната може мати декілька входів у інші кімнати. Вхід до будинку – лабіринту розташовано у кімнаті № 1, телефон в кімнаті № 7.

На початку роботи програма обирає факт з номером кімнати, у якій є вхід в будинок-лабіринт. Після чого за фактом перевіряється наявність у цій кімнаті телефону. Якщо телефону у кімнаті немає, то з факту обирається номер будь-якої кімнати суміжної з обраною. Обрана кімната стає поточною і для неї повторяються дії описані вище.

Поточні кімнати обираються до тих пір, поки для суміжної кімнати не виявиться, що факт, який її описує, відсутній. Така ситуація розцінюється як тупик.

Наприклад, нехай прокладено шлях через кімнати 1, 2, 3, 4, 6, а для кімнати 6 факт відсутній, тобто вихід з неї той же, що і вхід. Тоді кімната № 6 є тупиком. При виявленні тупика включається механізм звороту і шукається найближча кімната на пройденому шляху, у якій є альтернативні виходи. Це означає, що для цієї кімнати є ще один факт з іншою суміжною кімнатою. Такий факт є для кімнати № 2, яка є точкою розгалуження. Далі повторюються дії вказані на початку. Нова суміжна кімната стає поточною. У ній перевіряється наявність телефону. Якщо телефону немає, то будь-яка суміжна з поточною кімната стає поточною і. т. д.

Від кімнати № 2 прокладається новий шлях з номерами: 2, 5, 7. У кімнаті № 7 виявляється телефон.

```
Global Facts
стан(integer, integer, integer)
Single поточна_кімната (integer)
Predicates
Nondeterm лабіринт()
Nondeterm тупик(integer)
Goal
лабіринт(), nl, !.
Clauses
поточна_кімната(1).
стан(1, 2, 0).
стан(2, 3, 0).
стан(3, 4, 0).
стан(4, 6, 0).
стан(2, 5, 0).
стан(5, 7, 0).
стан(7, 5, 1).
лабіринт ():-поточна_кімната(Номер),
        стан(Номер, _, 1),
        write («Телефон у кімнаті №», Номер), nl, !.
```

```
лабіринт():-поточна_кімната(Номер), тупик(Номер),
        стан(Номер, Суміжний, 0), write(Номер), nl,
        assert(поточна_кімната(Суміжний)), лабіринт().

        тупик(Номер):- not (стан(Номер, _, 0)), write(«Кімната №»,
        Номер, «тупик!»), nl; True.
```

Можливий інший варіант роботи програми, якщо порядок фактів змінити. Розмістимо факт стан(2, 5, 0). раніше ніж факт стан (2, 3, 0). В цьому випадку шлях до кімнати з телефоном буде знайдений програмою відразу: 2, 5, 7. Порядок фактів у програмі впливає на вибір напряму в лабіринті та довжину шляху аналогічно вибору напряму в лабіринті людиною. В штучному інтелекту завжди обирається перше знайдене рішення, що підходить.

У нижче розташованій програмі вибір напряму руху обирається людиною. Для цього вводиться опит руху від кожної поточної кімнати, яка має альтернативні шляхи. За результатами вводу два факти з обраним поточним номером та наступної кімнати (прямий шлях і зворотний) розташовуються на початку всіх фактів. Якщо альтернативного шляху немає, то програма сама знаходить номер наступної кімнати, а відповідні два факти переміщує в кінець фактів.

```
Global Domains
list = integer*

Global Facts
стан(integer, integer, integer)
Single поточна_кімната (integer)
Single список (list)
Single попередній_номер(integer)
Single кільк (integer)

Predicates
Nondeterm лабіринт()
Nondeterm суміжний (integer, integer)
Nondeterm людина(integer)
Nondeterm вивід(integer)
Nondeterm sp (list)

Goal
лабіринт(), !.
```



```
Clauses
попередній_номер(0).
список([]).
кільк(0).
поточна_кімната(1).

стан(1, 2, 0).
стан(2, 1, 0).
стан(2, 3, 0).
стан(3, 2, 0).
стан(3, 4, 0).
стан(4, 3, 0).
стан(4, 6, 0).
стан(6, 4, 0).
стан(2, 5, 0).
стан(5, 2, 0).
стан(5, 7, 0).
стан(7, 5, 1).

лабіринт ():-поточна_кімната(Номер), стан(Номер, _, 1),
  write («Телефон у кімнаті №», Номер), nl,!.
лабіринт():-поточна_кімната(Номер), людина(Номер),
  стан(Номер, Суміжний, 0),
  write(«Кімната», Суміжний), nl,
  assert(поточна_кімната(Суміжний)),
  assert(попередній_номер(Номер)),
  retract(стан(Номер, Суміжний, 0)),
  assertz(стан(Номер, Суміжний, 0)),
  суміжний(Номер, Суміжний), лабіринт().

суміжний(Номер, Суміжний):- стан(Суміжний, Номер,0),
  retract(стан(Суміжний, Номер, 0)),
  assertz(стан(Суміжний, Номер, 0)),!;
  стан(Суміжний, Номер,1),
  assert(поточна_кімната(Суміжний)).

Людина(Номер):- вивід (Номер),
  write(«Оберіть № кімнати:»), readint (N),
  стан(Номер, N, 0), retract(стан(Номер, N, 0)),
  asserta (стан(Номер, N, 0)),
  retract(стан(N, Номер , 0)),
```

```
  asserta (стан(N, Номер , 0)),!,
  assert(кільк(0)), assert(список([])).

Людина(_):-assert(кільк(0)), assert(список([])).

вивід(Номер):- стан(Номер, Суміжний,0),
  попередній_номер(Суміжний1), Суміжний<>Суміжний1,
  список(Сп), Сп1=[Суміжний|Сп], assert(список(Сп1)),
  кільк(К), К1=К+1, assert(кільк(К1)), fail.

вивід(_):- кільк(К), К>1, список(Сп), sp(Сп),
  assert(список([])); assert(список([])), fail.

sp([]).
sp([Н|Т]):-write(«Наступна кімната №», Н), nl,!, sp(Т).
```

Якщо факт для зворотного шляху з поточної кімнати стоїть попереду факту для наступної кімнати, то людина повернеться до попередньої кімнати. Щоб уникнути цього, факт для зворотного шляху з поточної кімнати переміщується разом з фактом для прямого шляху.

В наш час лабіринтами займаються психологи, конструктори комп'ютерів, спеціалісти штучного інтелекту. Одним з перших роботів, що вмів знаходити шлях в лабіринту, була миша-робот, яку створив Клод Шеннон з массачусетського технологічного інституту. Шеннон застосував один з варіантів проходження лабіринту запропонований Тремо за наступними правилами.

1. Виходячи з будь-якої точки лабіринту треба зробити відмітку на стіні (хрестом) і рухатися до тупика або розгалуження.
2. Якщо рух закінчився тупиком, то повернутися назад і зробити другу відмітку – шлях пройдено двічі. Після чого йти у напрямку, у якому не був або був один раз.
3. Якщо точкою звороту було розгалуження, то йти у будь-якому напрямку, відмічаючи кожне розгалуження на вході і на виході хрестом. Якщо на розгалуженні вже є один хрест, то треба йти новим шляхом, якщо хреста нема, то пройденим шляхом, відмітив його другим хрестом.

Сьогодні, у кількох країнах світу проходять змагання роботів у лабіринті [1]. Роботи-машинки повинні за найкоротший час знайти шлях у лабіринті від старту до фінішу. Робот повинен контролювати свій рух за чорними відмітками на полу. Змагання відносяться до технічного спорту роботів.

3.2.2 Пошук найкоротшого шляху між двома вершинами орієнтованого графу

Задача знаходження найкоротшого шляху між будь-якими двома вершинами орієнтованого графу має велику кількість практичних застосувань. Для цього треба приписувати дугам орієнтованого графу не відстані, а інші числові характеристики, які більше або дорівнюють нулю. Дуга може означати час між подіями, вартість проїзду між станціями, відстань між населеними пунктами, кількість подій між певними моментами життя людини, пропускну спроможність каналу між двома суміжними маршрутизаторами тощо.

У штучному інтелекті задача знаходження найкоротшого шляху виникає при пошуку у просторі станів (*SS* – проблема). Серед станів простору треба знайти шлях, що веде від початкового стану до цільового. Опис задачі в просторі станів подається множиною станів, множиною операторів та їх дії на зміну станів, початковим та цільовим станом. Простір станів можна задавати графом, у якому стан вершина, а дуга оператор. Розв'язати задачу можна повним перебором шляхів, але такий метод для реальних задач працює довго. Швидше працює алгоритм Дейкстри, який буде розглянуто нижче. Алгоритм широко застосовується в різних технологіях та програмних системах в тих випадках, коли характеристики дуг всі невід'ємні.

Наприклад, алгоритм застосовують в мережевому протоколі динамічної маршрутизації *Open Shortest Path First* (OSPF), який застосовують у якості внутрішнього протоколу маршрутизації.

Автономна система може складатися з кількох областей. Кожна область може складатися з окремих комп'ютерів або мереж. Для мереж внутрішні маршрутизатори можуть не мати інформації про топологію інших областей автономної системи. Але кожна мережа має один маршрутизатор, який дає інформацію про маршрути мережі іншим маршрутизаторам автономної системи. Всі маршрутизатори однієї мережі самостійно приймають рішення про пересилку пакетів, застосовуючи відомості про топологію мережі. Перед цим маршрутизатор самостійно розв'язує задачу оптимізації маршрутів на *орієнтованому графі мережі.*

Маршрутизатор одержує адреси мереж одержувачів з пакетів даних і визначає за цими адресами у таблиці маршрутизації шляхи передачі даних. Таблиця маршрутизації містить записи. У кожному

запису є: адреса мережі одержувача, адреса наступного вузла, якому треба передати пакет даних, вага маршруту. Вага маршруту залежить від коефіцієнту якості обслуговування. Чим менше вага маршруту, тим кращий маршрут. Записи маршрутів в таблиці маршрутизації обновляються автоматично.

Маршрутизатор обчислює коефіцієнт якості обслуговування за критерієм пропускної спроможності каналів. Стан каналу він одержує від суміжного з ним маршрутизатору, записує його в свою БД і посилає копію іншим суміжним з ним маршрутизаторам. Таким чином будується БД стану всіх каналів мережі. Після цього кожен маршрутизатор використовує алгоритм Дейкстри для оптимізації маршрутів. Кожен маршрутизатор автоматично створює таблицю оптимізованих маршрутів із своєї БД. Такий підхід дозволяє обчислювати оптимальні шляхи маршрутів відповідно поточній топології мережі.

Щоб порівняти методи, спочатку розглянемо метод повного перебору шляхів з поверненням для знаходження найменшого шляху у орієнтованому графі. Нехай є орієнтований граф $G(V, E)$, де V – множина вершин графу, E множина його дуг. Кожній дузі приписана певна числова невід'ємна характеристика. Будемо називати її відстанню. Приклад графу наведено на рис. 3.8.

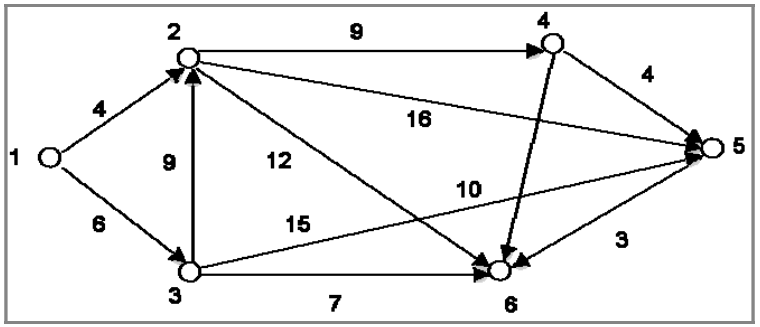


Рисунок 3.8 – Приклад орієнтованого графу

Програма повного перебору шляхів з поверненням знаходить спочатку всі шляхи, після чого знаходить найменший шлях.

```
Global Domains
lst = integer*
```



```
Global Facts
Nondeterm дуга(integer, integer, integer)
Single список (lst)
Single сума (integer)
Nondeterm шлях (lst, integer)
Single m (integer)

Predicates
Nondeterm перебрати(integer)
Nondeterm print()
Nondeterm min()

Goal
Write («Шляхи \n»), перебрати (1),
список(S), сума(Sum), assert (шлях (S, Sum)),
Assert (список ([])), assert (сума (0)), fail;
Print (), min ().

Clauses
список([]).
сума(0).
m(95).

дуга(1, 2, 4).
дуга(1, 3, 6).
дуга(4, 5, 4).
дуга(4, 6, 10).
дуга(2, 4, 9).
дуга(2, 5, 16).
дуга(2, 6, 12).
дуга(5, 6, 3).
дуга(3, 2, 9).
дуга(3, 5, 15).
дуга(3, 6, 7).

перебрати(VP):- VP<>6, дуга(VP, VS, L1),
        перебрати(VS), список(S1),
        S = [VP|S1], assert(список(S)),
        сума(Sum1), Sum=Sum1+L1, assert(сума(Sum)).

перебрати(6):- список(S), S1=[6|S],
```

```
        assert(список(S1)).
Print ():- шлях(S, Sum),
          write(S, «Довжина шляху», Sum), nl, fail.
Print ().
min():- шлях(_, Sum), m(M1), Sum < M1,
          assert(m(Sum)), fail.
min():-m(M1), write(«Найкоротший шлях», M1).
```

Шляхи, одержані в результаті роботи програми:

[1,2,4,5,6] Довжина шляху20
[1,2,4,6] Довжина шляху23
[1,2,5,6] Довжина шляху23
[1,2,6] Довжина шляху16
[1,3,2,4,5,6] Довжина шляху31
[1,3,2,4,6] Довжина шляху34
[1,3,2,5,6] Довжина шляху34
[1,3,2,6] Довжина шляху27
[1,3,5,6] Довжина шляху24
[1,3,6] Довжина шляху13

Найкоротший шлях 13

На кожному рівні низхідної рекурсії включаються неповні списки вершин шляху, але характеристика предиката список single, тому при підйомі на верхньому рівні всі неповні списки замінюються повним списком. Аналогічно рахується довжина повного шляху.

Оцінимо часову складність алгоритму. Із зростанням кількості вершин графу час роботи алгоритму зростає за експонентою. Дійсно, якщо всі множини вершин V_i доступні для вибору вершини в поточний момент обмежити константою C, то оцінку алгоритму буде подано показовою залежністю C^N вузлів, де N кількість вершин.

Розробимо тепер програму, що знаходить найкоротший шлях за алгоритмом Дейкстри. Для цього розглянемо суть алгоритму Дейкстри на тому ж прикладі [7].

Задамо матрицю суміжності вершин графу фактами, де перший аргумент номер поточної вершини, другий аргумент номер наступної суміжної з нею вершини, третій аргумент вказує на відстань між вказаними вершинами.

Кожна поточна вершина описується кількома фактами:

```
дуга(1, 2, 4).
дуга(1, 3, 6).
дуга(2, 4, 9).
дуга(2, 5, 16).
дуга(2, 6, 12).
дуга(3, 2, 9).
дуга(3, 5, 15).
дуга(3, 6, 7).
дуга(4, 5, 4).
дуга(4, 6, 10).
дуга(5, 6, 3).
```

Ми будемо розглядати тільки шляхи, що проходять через кожну дугу графу не більше одного разу.

Нехай початковою вершиною буде вершина 1, а кінцевою вершина 6.

Будемо позначати поточну вершину графу змінною Y, що одержує значення i=1, 2, …,6. Позначимо відстань від вершини 1 до вершини Y змінною L_y.

На початку покладемо Y=1, тоді відстань від вершини 1 до себе L_1=0, а початкову відстань від вершини 1 до інших вершин графу L_i (i=2,..,6) покладемо більше ніж суму всіх відстаней у графі. Це дозволить знаходити мінімальні відстані від вершини 1 до інших вершин графу. Для нашого графу покладемо початкову відстань більше 95, а саме 96.

Найменша відстань від вершини Y до кожної L_i знаходиться наступним чином:

$$L_i = min\ (95, L_y + D_{yi}),$$

де 95 – попередньо прийнята відстань від вершини 1до вершини i;

L_y – попередньо обчислена мінімальна відстань від вершини 1 до вершини Y;

D_{yi} – відстань від поточної вершини Y до вершини i.

Обчислимо множину найкоротших відстаней для Y = 1,

i=2, *i*= 3:

L2 = *min* {95, 0 + 4} = 4;

L3= *min* {95, 0 + 6} = 6.

Серед всіх знайдених мінімальних відстаней до вершин суміжних з вершиною 1 знаходиться мінімальна. В нашому прикладі – це відстань 4, що веде до вершини 2. Тому поточною вершиною стає вершина *Y*=2, попереднє мінімальне значення відстані L_y = 4. Відстань від вершини 1 до вершин 4, 5, 6 як і раніше дорівнює 95.

Підрахуємо множину мінімальних відстаней з вершини 2:

L4 = *min* {95, 4 + 9} = 13;

L5 = *min* {95, 4 + 16} = 20;

L6 = min {95, 4 + 12} = 16.

Відстань *L3*= 6 як раніше;

Серед мінімальних відстаней найменша *L3*= 6, тому поточним значенням *Y* стає *Y*=3. Вершина 2 вже не розглядається, бо до вершини відстань вже обчислена. Попереднє мінімальне значення відстані L_y = 6. Але відстань від вершини 1 до вершин *i* змінилася: Відстань до вершини 5 стала 20, до вершини 6 стала 16, до вершини 4 стала 13.

Підрахуємо можливі мінімальні відстані з вершини 3:

L5 = *min* {20, 6 + 15} = 20;

L6 = *min* {16, 6 + 7} = 13

Звідки одержимо:

L4= 13; *L5*=20; *L6*=13.

Серед мінімальних відстаней найменша *L6*= 13, тому поточним значенням *Y* стає *Y*=6, а мінімальна відстань від вершини 1 до вершини 6 дорівнює 13.

Розглянемо програму, яка реалізує механізм Дейкстри для нашого прикладу. Подамо орієнтований граф фактами дуга; масив відстаней від початкової дуги 1 до кінцевої дуги 6 множиною фактів мін_відст. Список вершин мін_шлях фіксує мінімальний шлях від дуги 1 до дуги 6.


```
Global Domains
list = integer*
Global Facts
Nondeterm дуга (integer, integer, integer)
Single поточна_вершина (integer, integer, integer)
Single мін (integer, integer)
Nondeterm мін_відст(integer, integer, integer)
Single мін_шлях (list)
Predicates
Nondeterm дейкстри ()
Nondeterm сформувати_масив_відстаней (integer, integer, integer)
Nondeterm змінити_масив_відстаней ()
Nondeterm порівняти (integer, integer, integer)
Nondeterm знайти_мін_відстань ()
Nondeterm знайти_мін_шлях(integer, integer)
Nondeterm роби ()
Nondeterm обернути(list, list, list)
Goal
сформувати_масив_відстаней (1, 1, 6), дейкстри ().
Clauses
поточна_вершина (1, 1, 0).
мін (1,96).
мін_шлях ([1]).
дуга(1, 2, 4).
дуга(1, 3, 6).
дуга(4, 5, 4).
дуга(4, 6, 10).
дуга(2, 4, 9).
дуга(2, 5, 16).
дуга(2, 6, 12).
дуга(5, 6, 3).
дуга(3, 2, 9).
дуга(3, 5, 15).
дуга(3, 6, 7).
сформувати_масив_відстаней(_, N2, N2):-nl,!.
```

```
сформувати_масив_відстаней(N1, N, N2):- N11 = N+1,
        write («Масив відстаней», N1, «», N11, «»,96),
        assert (мін_відст (N1, N11, 96)),!,
        сформувати_масив_відстаней(N1, N11, N2).
дейкстри ():- змінити_масив_відстаней(),
        знайти_мін_відстань(),
        роби (), !, дейкстри ().
дейкстри ():- мін_відст (1, 6, S), nl,
        write(«Мінімальна відстань від 1 до 6:», S), nl, мін_шлях
        (Шлях), обернути ([6|Шлях],[], Шлях2),
        write («Шлях», Шлях2), nl.
змінити_масив_відстаней () :-
        дуга (Вершина, Вершина_суміж, Відстань),
        поточна_вершина (Вершина,_, Мін_відстань),
        порівняти (Вершина_суміж, Відстань, Мін_відстань), fail.
змінити_масив_відстаней ().
порівняти(Вершина_суміж, Відстань, Мін_відстань):-
        мін_відст(1, Вершина_суміж, Попередня_відстань),
        Сума = Мін_відстань + Відстань,
        Попередня_відстань > Сума,
        retract(мін_відст(1, Вершина_суміж, Попередня_відстань)),
        assert(мін_відст(1, Вершина_суміж, Сума)),
        write («Мін. відст. від 1 до», Вершина_суміж,
        «Сума», Сума), nl, !;
        мін_відст (1, Вершина_суміж, Попередня_відстань),
        write («Мін. відст. від 1 до», Вершина_суміж,
        «Попередня_відстань», Попередня_відстань), nl,!.
знайти_мін_відстань():- мін_відст (1,_, Відстань),
        мін(1, Відстань2),
        Відстань2 >= Відстань,
        assert (мін(1, Відстань)), fail.
знайти_мін_відстань():- мін (Вершина, Відстань),
        дуга (Вершина, Вершина_суміж, Відстань),
        Retract (мін_відст(1, Вершина_суміж,_)),
        assert(поточна_вершина (Вершина_суміж, Вершина_суміж,
        Відстань)), знайти_мін_шлях (Вершина, Вершина_суміж),
        assert (мін (1, 96)),
```



```
        write(«Поточна вершина», Вершина, «», Вершина_суміж, «»,
        Відстань), nl, nl.
знайти_мін_шлях(Вершина, Вершина_суміж):-
        мін_шлях (Шлях),
        Шлях = [Вершина| _], Шлях2=[Вершина_суміж|Шлях],
        assert(мін_шлях (Шлях2));
        мін_шлях (Шлях), Шлях = [_| [Вершина| Хвіст]],
        Шлях2=[Вершина_суміж|[Вершина| Хвіст]], assert(мін_шлях
        (Шлях2)).
роби ():- мін_відст (N1, V, S),
        Write («Мин_відст від», N1,
        «до», V, «Сума», S), nl , fail.
        роби ():-nl.
обернути([], Шлях2, Шлях2).
обернути([Вершина|Хвіст], Шлях, Шлях2):-
        обернути (Хвіст, [Вершина| Шлях], Шлях2).
```

На початку роботи програми процедура сформувати_масив_відстаней створює масив відстаней від вершини 1 до всіх інших вершин графу і покладає відстані між ними максимально можливими – 96. Поточною вершиною спочатку вважається вершина 1, а відстань від неї до себе 0.

Процедура *дейкстри* є головною процедурою програми, яка керує послідовністю виконання процедур і виводить результати роботи програми.

Робота процедури змінити_масив_відстаней залежить від значень аргументів факту поточна вершина. За аргументом Вершина перебираються всі вершини суміжні з поточною.

Процедура порівняти знаходить мінімальну відстань від початкової вершини до кожної вершини суміжної з поточною наступними чином:

1. Знаходить суму відстаней від початкової вершини до поточної вершини і від поточної до суміжної з поточною.

2. Кожну суму порівнює з попередньою відстанню для цих же вершин у масиві відстаней. У масиві зберігається та відстань, що менше.

Процедура знайти_мін_відстань знаходить у масиві відстаней вершину, відстань до якої від початкової вершини мінімальна. Одержана вершина стає поточною вершиною. Факт, що містить

найменшу відстань до поточної вилучається з масиву відстаней. Одночасно формується список вершин до поточної вершини процедурою знайти_мін_відстань.

Дії повторюються, до тих пір поки у масиві відстаней не залишиться тільки факт від першої до кінцевої вершини.

Нижче поданий результат роботи програми.

```
Масив відстаней 1 2 96
Масив відстаней 1 3 96
Масив відстаней 1 4 96
Масив відстаней 1 5 96
Масив відстаней 1 6 96

Мін відст від 1 до 2 Сума 4
Мін відст від 1 до 3 Сума 6
Поточна вершина 1 2 4

Мін відст від 1 до 4 Сума 96
Мін відст від 1 до 5 Сума 96
Мін відст від 1 до 6 Сума 96
Мін відст від 1 до 3 Сума 6

Мін відст від 1 до 4 Сума 13
Мін відст від 1 до 5 Сума 20
Мін відст від 1 до 6 Сума 16
Поточна вершина 1 3 6

Мін відст від 1 до 4 Сума 13
Мін відст від 1 до 5 Сума 20
Мін відст від 1 до 6 Сума 16

Мін відст від 1 до 5 Попередня_відстань 20
Мін відст від 1 до 6 Сума 13

Знайдена мінімальна відстань від 1 до 6: 13

Шлях[1,3,6]
```

Оцінка часу роботи алгоритму Дейкстри $O(N^2)$, де N кількість вершин графу. Тобто із збільшенням кількості вершин графу час роботи алгоритму зростає не більше ніж N^2.

Резюме

Існує велика кількість типів задач, розв'язування яких вимагає застосування механізмів пошуку вшир та вглиб, механізму звороту. Пролог базується на вказаних механізмах, що значно зменшують об'єми програм у порівнянні з іншими мовами програмування. До задач таких типів відносять задачі пошуку шляху в лабіринтах і задачі, що пов'язані із графами.

Лабіринти завжди цікавили людство. Існують різні тлумачення походження лабіринтів: релігійні, мистецькі, астрономічні. Сьогодні лабіринти різних типів застосовують спеціалісти штучного інтелекту для навчання роботів. Задача пошуку найкоротшого шляху у орієнтованому графу має велику кількість практичних застосувань. Існують різні методи розв'язування такої задачі. Найбільш поширеним є метод Дейкстри.

Контрольні питання

1. Пояснити механізм пошуку вшир на прикладі програми мовою Visual Prolog.
2. Пояснити механізм пошуку вглиб на прикладі програми мовою Visual Prolog.
3. Якими засобами Visual Prolog можна керувати обходом дерева цілі?
4. Чому задачі, що вимагають перебору варіантів рішень легше писати мовою Visual Prolog?
5. Які типи лабіринтів вам відомі?
6. Намалювати за правилами побудови лабіринту критський лабіринт з семи кіл.
7. Яка характеристика лабіринту є топологічним варіантом?
8. Для яких типів лабіринтів працює «метод руки»?
9. Описати метод проходження лабіринту Люка – Тремо.
10. Описати метод проходження лабіринту Тремо, який застосував Клод Шеннон.
11. Пояснити суть алгоритму Дейкстри.
12. У яких областях застосовують алгоритм Дейкстри?

Вправи

1. Знайти програмно всі тупикові гілки у лабіринті (рис. 3.9).

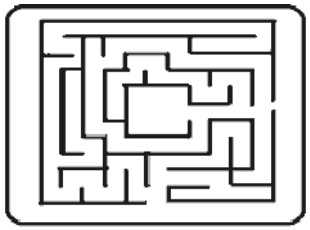


Рисунок 3.9 – Однозв’язний лабіринт

2. Реалізувати програму, що шукає шлях у лабіринті мовою C++. Оцінити обидві програми за часом виконання, об’ємом коду, простотою написання.

3. Розробити програму, що знаходить мінімальну відстань між двома вершинами неорієнтованого графу методом Дейкстри.

3.3 Символічні обчислення

3.3.1 Фізична символьна система

Німецький математик Д. Гілберт зробив припущення, що математичні конструкції можна розглядати незалежно від змістовного сенсу математичного поняття. Дослідження Гілберта мали великий вплив на розвиток математичної логіки, теорію доведення. При вивченні певної математичної теорії, що розглядається із змістової точки зору, розглядають також відповідну їй формальну систему (формалізм) незалежно від змісту.

Під формальною системою розуміють множину вихідних символів, правил побудови та перетворення символьних формул до кінцевих символьних формул. Виконання дій у певній *формальній системі* відносять до символічних обчислень.

Прикладом символічних обчислень можуть бути приведення подібних у символьних виразах, численне інтегрування та диференціювання символьних виразів, ведення символьних баз знань та символьний висновок на них.

Мовою AutoLisp створена діалогова система REDUCE, яка дозволяє виконувати різноманітні дії над символьними виразами. Система призначена для математиків, інженерів, вчених. Перетворення математичних викладок – громіздка робота, що займає багато часу. Система REDUCE бере на себе велику частину цих робіт.

Система може виконувати спрощення символьних виразів, виконувати дії над символічними матрицями, розв’язувати алгебраїчні рівняння, виконувати аналітичне диференціювання та інтегрування.

Можливість інтерпретувати формальні системи поняттями змістової теорії відкриває великі можливості для застосування формалізмів у штучному інтелекту. Це дає можливість інтелектуальним системам застосовувати один формалізм для ПДО різних наук.

Під інтерпретацією формалізму у штучному інтелекту розуміють когнітивну процедуру виявлення змісту понять та значень символів формалізму шляхом відображення їх на конкретну ПДО. Слово «когнітивний» означає з латинської розуміти, бачити свідомо.

Відомий спеціаліст по когнітивній психології та штучному інтелекту Алан Ньюелл та професор психолог Герберт А. Саймон з університету Карнегі-Меллон сформулювали гіпотезу *про формальну систему*, яку назвали фізичною системою символів:

«Фізична *символічна* система має необхідні та достатні засоби для загальної інтелектуальної дії». Під фізичною системою вони розуміли систему, що відображує реальний світ, а під загальною інтелектуальною дією – моделювання мислення людини символічними обчисленнями.

Ньюелл вважав, що людське мислення являє собою систему, що оперує символами, які відображують дійсність. В його роботах було сформульовано основні принципи моделювання мислення людини інтелектуальною системою.

Гіпотезу піддали критиці, бо процеси мислення людини мають складну багаторівневу структуру і існують різні підходи до моделювання мислених процесів (граматичний, концептуальний, семантичний, логічний). Але корисність гіпотези була явною для міркування вищого рівня – логічного мислення на рівні понять, яке застосовує логічний формалізм.

Фізична символьна система повинна містити наступні компоненти.

1. *Пам’ять*. Набір символів та символічних структур, що зберігаються в певній області. Склад символічних структур може поповнюватися.

2. *Інтерфейс*. Засоби спілкування системи з середовищем: увід та вивід, рецептори та ефектори. Рецептори виявляють подразник, а ефектори реагують на нього. Сумісна їх робота нагадує правило «якщо-то».

3. *Інтерпретатор.* Засоби інтерпретації природних для користувача форм представлення понять символічними структурами або засоби інтерпретації символічних структур природними для користувача формами представлення понять.

4. *Процесор*. Множина процедур, для виконання необхідних дій над символами та структурами.

Мова Visual Prolog базується на логічному формалізмі так, як і фізична символічна система. Тому засоби мови пристосовані до символічних обчислень. В цьому розділі ми розглянемо застосування засобів Visual Prolog для виконання одного з видів символічних обчислень – диференціювання виразів у символьному виді. На цьому прикладі ми покажемо, наявність компонент *фізичної символічної системи* у програмі, що робить символічні обчислення.

3.3.2 Знаходження похідної символьного виразу

Математичні символьні вирази, що розглядаються, матимуть інфіксні арифметичні операції: +, –, * та постфіксну операцію піднести до степеня (^), а їх операндами будуть цілі числа, букви або дійсні числа, що записуються у формі з фіксованою точкою.

У компоненту *Пам'ять* повинні зберігатися символи операцій, що може мати вираз, формули диференціювання елементарних виразів, з яких складатиметься будь-який вираз вказаного типу.

Символи, що позначають коефіцієнти та змінні, користувач може застосовувати довільно, вони не фіксуються.

Компонент *Інтерфейс* вводить математичний символьний вираз у природному для користувача вигляді рядком і передає його інтерпретатору.

Компонент *Інтерпретатор* перетворює символьний вираз, що поданий рядком у структуру.

Обчислення значення виразу виконується згідно пріоритету операцій. Порядок диференціювання операцій виразу відповідає порядку їх виконання. Якщо у програмі подати *вираз у вигляді структури за*

пріоритетами операцій, то структура визначить порядок виконання операцій виразу. У структурі всього виразу операції більшого пріоритету, будуть вкладені в операції нижчого пріоритету.

Наприклад, вираз 4**x*–56 буде представлений як структура –(*(4, *x*), 56), де функторами структур є операції, а аргументами структур є операнди операцій.

Порядок виконання операцій одного пріоритету визначається за їх асоціативністю. Інфіксні операції +, –, * відносяться до ліво асоціативних операцій. Якщо у виразі є кілька операцій одного пріоритету, то вони виконуються зліва направо. Постфіксна операція *піднести до степеня* відноситься до право асоціативних операцій. Такі операції виконуються справа наліво.

Нехай у арифметичному виразі є кілька операцій одного пріоритету. Тоді в структурі всього виразу операція, що розташована лівіше у виразі, буде вкладена в операції, що розташовані праворуч.

Наприклад, вираз 16*2*2 у вигляді структури буде поданий mult(mult(16,2),2).

Компонент *Процесор* диференціює складну функцію. Структуру виразу можна розглядати, як складну функцію.

Позначимо складну функцію *F(x)*. Диференціювання складної функції будемо виконувати рекурсивно за наступними правилами.

$$\frac{dc}{dx} \to 0$$

$$\frac{dx}{dx} \to 1$$

$$\frac{d(U+V)}{dx} \to \frac{dU}{dx} + \frac{dV}{dx}$$

$$\frac{d(U-V)}{dx} \to \frac{dU}{dx} - \frac{dV}{dx}$$

$$\frac{d(cU)}{dx} \to \frac{cdU}{dx}$$

$$\frac{d(U*V)}{dx} \to U*\frac{dV}{dx} + V*\frac{dU}{dx}$$

$$\frac{d(U^c)}{dx} \to c * U^{c-1} * \left(\frac{dU}{dx}\right)$$

Результат диференціювання передається у компоненту *Інтерпретатор*, який перетворює його до виразу у вигляді рядку. Компонент *Інтерфейс* виводить результат для користувача. У програмі, що диференціює вираз, ми зробимо деякі спрощення, які розглянемо пізніше.

```
Global Domains
S = reference string
list = s*
c = рядок (s);
        plus (c, c);
        minus (c, c);
        mult(c , c);
        exponent (c, c)

Global Facts
Nondeterm c(string)

Single змінна (string)

Predicates
роби()
Nondeterm диф(c, c, c)
Nondeterm append(list, list, list)
Nondeterm роби_структуру(string, c)
Nondeterm вираз_список(string, list)
Nondeterm список_структура(list, c)
Nondeterm структура_список(c, list)
Nondeterm список_вираз(list, string)
Nondeterm роби_вираз(c, string)
Nondeterm належить(string, list)
Nondeterm спростити(string, string)
Nondeterm спростити1(string, string)
Nondeterm спростити2(string, string)
Nondeterm спростити3(string, string)
```



```
Clauses

c(«+0»).
c(«*1»).
c(«-0»).
c(«0»).
c(«^1»).

змінна («»).

append ([], L, L).
append ([H | L1], L2, [H | L3]):-append (L1, L2, L3).

належить(Операція, [Операція | _]).
належить(Операція,[_|Хвіст]):-
        належить(Операція, Хвіст).

роби_структуру(Вираз, Структура):-
        вираз_список(Вираз, Список),
        список_структура(Список, Структура).

вираз_список(«», []).

вираз_список(Рядок, Список1):-
        fronttoken (Рядок, Слово, Залишок),
        Вираз_список(Залишок, Список2),
        Список1=[Слово|Список2].

        список_структура (L, рядок(V)):-
        not (L = [_|[]]), змінна (R),
        not (належить (R, L)),
        список_вираз(L, V).

список_структура(L, plus(Структура1, Структура2)):-
        append (L1, L2, L), L2 = [«+» | Залишок],
        not (належить («+», Залишок)),
        not (належить («–», Залишок)),
        список_структура(L1, Структура1),
        список_структура(Залишок, Структура2).

список_структура(L, minus(Структура1, Структура2)):-

        append (L1, L2, L), L2 = [«–» | Залишок],
```

```
        not (належить («–», Залишок)),
        not (належить («+», Залишок)),
        список_структура(L1, Структура1),
        список_структура(Залишок, Структура2).

список_структура(L, mult(Структура1, Структура2)):-
        append (L1, L2, L), L2 = [«*» | Залишок],
        not (належить («*», Залишок)),
        Not (належить («/», Залишок)),
        список_структура(L1, Структура1),
        список_структура(Залишок, Структура2),!.

список_структура(L, exponent(Структура1, Структура2)):-
        append (L1, L2, L), L2 = [«^» | Залишок],
        список_структура(L1, Структура1),
        список_структура(Залишок, Структура2).

список_структура([N], рядок(N)).

диф(рядок(C), рядок(X), рядок(«0»)):- C<>X,!.

диф(рядок(X), рядок(X), рядок(«1»)):-!.

диф(plus(U, V), рядок(X), plus(A, B)):-
        диф(U, рядок(X), A), диф(V, рядок(X), B).

диф(minus(U, V), рядок(X), minus(A, B)):-
        диф(U, рядок(X), A), диф(V, рядок(X), B).

диф(mult(C, U), рядок(X), mult(C, A)):-
        диф(U, рядок(X), A),!.

диф(mult(U, V), рядок(X), plus(mult(B, U), mult(A, V))):-
        диф(U, рядок(X), A), диф(V, рядок(X), B).

диф(exponent(U, C), рядок(X), exponent(mult(C, U),
        mult(minus(C, рядок(«1»)), W))):-

        диф(U, рядок(X), W).

роби_вираз(Диф_структура, Результат):-
        структура_список (Диф_структура, Список),
```



```
        список_вираз(Список, Вираз),
        спростити(Вираз, Результат).

структура_список(plus(Структура1, Структура2), Список):-
        структура_список(Структура1, Список1),
        структура_список(Структура2, Список2),
        Список22=[«+»|Список2],
        append (Список1, Список22, Список).

структура_список(minus(Структура1, Структура2), Список):-
        структура_список(Структура1, Список1),
        структура_список(Структура2, Список2),
        Список22=[«–» | Список2],
        append (Список1, Список22, Список).

структура_список(mult(Структура1, Структура2), Список):-
        структура_список(Структура1, Список1),
        структура_список(Структура2, Список2),
        Список22=[«*»|Список2],
        append (Список1, Список22, Список).

структура_список(exponent(Структура1, Структура2),

        Список):-
        структура_список(Структура1, Список1),
        структура_список(Структура2, Список2),
        Список22=[«^»| Список2],
        append (Список1, Список22, Список).

структура_список(рядок(S), [S]).

список_вираз([], «»).

        список_вираз([Слово|Залишок], Рядок):-
        список_вираз(Залишок, Рядок2),
        fronttoken(Рядок, Слово, Рядок2).

спростити(Вираз, Результат):-
        спростити1(Вираз, Результат1),
        спростити2(Результат1, Результат2),
        спростити3(Результат2, Результат3),
```

```
        спростити1(Результат3, Результат).

спростити1(«», «»).

спростити1(Вираз, Результат):-
        с(Маска), concat (Маска, Вираз2, Вираз),
        спростити1(Вираз2, Результат);
        frontchar (Вираз, С, Залишок), str_char (С1, С),
        спростити1(Залишок, Результат2),
        concat (С1, Результат2, Результат).

спростити2(«», «»).

спростити2(Вираз, Результат):-
        fronttoken(Вираз, «^», Залишок),
        fronttoken(Залишок, Число1, Залишок1),
        fronttoken(Залишок1, _, Залишок2),
        fronttoken(Залишок2, Число2, Залишок3),
        str_int(Число1, Ціле1), str_int(Число2, Ціле2),
        Ціле=Ціле1-Ціле2, str_int(Степінь, Ціле),
        concat(«^», Степінь, Степінь1),
        спростити2(Залишок3, Результат1),
        concat(Степінь1, Результат1, Результат);

        fronttoken(Вираз, Слово, Залишок),
        спростити2(Залишок, Результат1),
        concat(Слово, Результат1, Результат).

спростити3(«», «»).

спростити3(Вираз, Результат):-
        fronttoken(Вираз, Число1, Залишок1),
        str_real(Число1, Дійсне1),
        fronttoken(Залишок1, «*», Залишок2),
        fronttoken(Залишок2, Число2, Залишок3),
        str_real(Число2, Дійсне2),
        Дійсне=Дійсне1 * Дійсне2,
        str_real(Дійсне_с, Дійсне),
        спростити3(Залишок3, Результат1),
        concat(Дійсне_с, Результат1, Результат);
```



```
        fronttoken(Вираз, Слово, Залишок),
        спростити3(Залишок, Результат1),
        concat(Слово, Результат1, Результат).

роби():-write(«Уведіть вираз для диференціювання>«),
        readln (Вираз),
        write («За якою змінною диференціювати вираз>»),
        readln(Змінна1),
        assert (змінна(Змінна1)),
        Змінна2=рядок(Змінна1),
        роби_структуру(Вираз, Структура),
        диф(Структура, Змінна2, Диф_структура),
        роби_вираз(Диф_структура, Результат),
        Write («Результат диференціювання>«,
        Результат), nl,!.

Goal
роби().
```

Компонент *Інтерфейс* містить процедуру роби(), яка є головною процедурою програми, що визначає порядок виклику компонентів програми. Процедура є інтерфейсом між користувачем і програмою і між компонентами програми. Процедура опитує та вводить вираз, у вигляді рядка, та змінну, за якою треба його диференціювати. Після диференціювання виразу вона передає результат користувачу.

Процедура роби() взаємодіє з компонентами *Інтерпретатор*, *Процесор*.

Інтерпретатор містить дві основні процедури роби_структуру і роби_вираз.

Процедура роби_структуру одержує вхідну інформацію і застосовує допоміжні процедури перетворення виразу в список – вираз_список і перетворення списку в структуру список-структура.

Побудова структури необхідна для визначення порядку диференціювання членів виразу відповідно пріоритету і асоціативності операцій. Структуру зручно будувати, якщо вираз подати списком, бо розділення списку за операцією на два підсписки (операнди) виконує автоматично процедура append.

Порядок виконання тверджень процедури список_структура() фіксується операцією. Внаслідок цього операції задаються прямо в твердженнях процедури. Процедура реалізується низхідним методом рекурсії. Тому твердження з операціями нижчого пріоритету стоять у процедурі зовні, що дозволяє їх оброблювати пізніше.

Щоб лівоасоціативні операції з однаковим пріоритетом почали виконуватися зліва, необхідно, розмістити першу операцію внутрішньою у структурі. Тобто зовнішньою операцією мусить бути остання операція цього пріоритету. Знаходження останньої операції виконує комбінація предикатів:

not (належить («+», Залишок))

Результат роботи *інтерпретатору* – структура виразу повертається в головну процедуру програми роби() для передачі процесору системи.

Процесор виконує операцію диференціювання функцій. Для диференціювання складної функції *F(x)* застосовується метод низхідної рекурсії.

На етапі редукції знаходиться формула, що відповідає похідній функції *F(x),* поданої через її складові функції *U(x)* та *V(x)*, відповідно виразу. Знаходження похідної функції *F(x)* вимагає знаходження похідних її складових функцій, тому процес повторюється для функцій *U(x)* та *V(x)*. Дії виконуються до тих пір, поки складові не стануть константами або змінною, за якою виконується диференціювання. Відповідні їм формули є граничними умовами рекурсії.

На етапі одержання рішення результати формул з граничних умов підставляються у праву частину формули, з якої був виклик, після чого обчислюються значення з правої частини формули. Одержані значення застосовуються для представлення похідної складної функції. Дії повторюються до тих пір поки не буде одержана похідна складної функції *F(x)*, що подає весь вираз.

Структура, яка подає похідну функції *F(x)* повертається в головну процедуру *роби()* для передачі *інтерпретатору.*

Інтерпретатор повинен перетворити похідну в зручний для користувача вигляд. Для обробки результату диференціювання інтерпретатор має наступні процедури: процедуру через яку він одержує вхідну інформацію роби_вираз, процедуру перетворення структури в список і процедуру перетворення списку в вираз.

Процедура структура_список складає список із аргументів структури та знаку операції , що відповідає функтору структури. Список створюється процедурою append.

Крім цього *інтерпретатор* спрощує одержаний вираз, робить його зручним для читання. Процедура спростити1 виконує наступні дії: вилучає з виразу компоненти: +0; –0; 0, *1; ^1. Процедура спростити2 підраховує числові показники операції підвищення до степеню. Процедура спростити3 перемножує коефіцієнти операції множення. Процедура спростити фіксує порядок виконання процедур спрощення. Спрощений результат видається через інтерфейс.

Приклад роботи:
Уведіть вираз для диференціювання > 41*x^3+24*x-3
За якою змінною диференціювати вираз > x
Результат диференціювання > 123*x^2+24

Резюме

Під формальною системою (формалізмом) розуміють множину вихідних символів, правил побудови та перетворення символьних формул до кінцевих символьних формул. Виконання дій у певній *формальній системі* відносять до символічних обчислень.

До символічних обчислень відносять спрощення символьних виразів, виконання дій над символічними матрицями, розв’язування алгебраїчних рівнянь у символьному виді, виконання аналітичного диференціювання та інтегрування, ведення символьних баз знань та символьний висновок на них тощо.

Існує гіпотеза: «Фізична *символічна* система має необхідні та достатні засоби для загальної інтелектуальної дії», де під фізичною системою розуміють систему, що відображує реальний світ, а під загальною інтелектуальною дією – моделювання мислення людини символічними обчисленнями. Гіпотеза корисна для міркування вищого рівня – логічного мислення на рівні понять, яке застосовує

логічний формалізм. Фізична символьна система складається з пам'яті, інтерфейсу, інтерпретатору та процесору. Мова Visual Prolog базується на логічному формалізмі так, як і фізична символічна система. Компоненти фізичної символьної системи присутні у будь-якій системі, що виконує символічні обчислення.

Контрольні питання

1. Що таке формальна система(формалізм)?
2. Який формалізм вам відомий?
3. Поясніть, що таке символічні обчислення? Наведіть приклади символічних обчислень.
4. Які системи, що реалізують символічні обчислення вам відомі? Які дії виконують ці системи?
5. Поясніть зміст гіпотези про фізичну символічну систему.
6. З яких компонент складається фізична символічна система?
7. Пояснити призначення кожної компоненти фізичної символічної системи на прикладі програми символьного диференціювання.
8. Завдяки чому у прикладі програми для символьного диференціювання виразів зручно подавати вираз структурою?
9. Чому порядок тверджень процедури список_структура важливий?
10. Чому у процедурі список_структура робота розпочинається з останньої у виразі інфіксної лівоасоціативної операції?

Вправи

1. Поповнити програму символьного диференціювання так, щоб вона диференціювала вирази із *sin* та *cos*.
2. Поповнити програму символьного диференціювання так, щоб вона диференціювала вирази із операцією ділення.
3. Поповнити програму символьного диференціювання так, щоб вона приводила подібні.
4. Змінити текст програми символьного диференціювання, щоб вона мала компоненту пам'ять.

РОЗДІЛ IV. ДІАЛОГ ПРИРОДНОЮ МОВОЮ

Наука розвивається поступово за періодами: період диференціації – появи нових наук на стику існуючих, як результат ускладнення та зростання об'ємів знань; період інтеграції наук – синтез знань наук, взаємопроникнення методів наук. В той же час у надрах періоду диференціації зароджується період інтеграції. В. І. Вернадський вважав, що «вперше зливаються в єдине ціле всі, що до сих пір йшли у малій залежності одне від одного, а іноді зовсім незалежно, течії духовної творчості людини». Процес інтеграції наук доводить єдність природи.

У XX столітті стали з'являтися нові науки, що пов'язані із штучним інтелектом та його напрямом «*Natural Language Processing*». До таких наук можна віднести когнітивну психологію(гілку психології, що досліджує пізнавальні процеси людської свідомості), когнітивну лінгвістику(гілку мовознавства, що досліджує проблеми співвідношення мови і свідомості), психолінгвістику(науку про закономірності породження і сприйняття мовного висловлювання), логічну семантику (розділ математичної логіки). Розвиток цих наук може привести до створення єдиної теорії спілкування комп'ютера та людини природною мовою.

Сьогодні людина може спілкуватися з комп'ютерною системою обмеженою природною мовою. Така система має мати інтелектуальний інтерфейс або інтерфейс, що імітує інтелектуальність природно-мовного інтерфейсу. Такі інтерфейси дозволяють спілкуватися з людиною за конкретною темою. Для систем штучного інтелекту, яким потрібно спілкуватися з людиною у будь-якому місці роботи програми, спілкування природною мовою важлива властивість. Розуміння програмною системою речень природною мовою може виконуватися різними методами, від цього залежить тип інтерфейсу. У розділі розглядаються методи розуміння та генерації речень природною мовою, реалізації природно-мовних інтерфейсів, ведення діалогу, їх програмна реалізація мовою Visual Prolog.

4.1 Діалогові системи розв'язування задач

4.1.1 Типи природно-мовних систем та їх призначення

Системи, що спілкуються з людиною природною мовою, називають природно-мовними системами. Сама система може не бути інтелектуальною, але може мати інтелектуальний інтерфейс.

До природно-мовних систем відносять наступні.

1. Системи типу «Питання – відповідь».
2. Системи спілкування з БД та ОС.
3. Діалогові системи розв'язування задач.
4. Системи обробки текстів.

Системи типу «Питання – відповідь» є інтелектуальними системами, їх утворюють для дослідження методів розуміння змісту речень та генерації речень природною мовою. Системи цього типу не мають мети розмови. Сьогодні питаннями розуміння речень природною мовою займається велика кількість дослідників.

Найбільш відомими прикладами систем цього типу є системи: «МИВОС» 1982р., мова спілкування російська, мова програмування РЕФАЛ; «ПОЭТ» 1982р., мова спілкування російська, мова програмування ПЛ/1; «FAS-80» 1983 р., мова спілкування німецька, мова програмування ЛИСП; «MARGIE» 1980р., розроблена у Стенфордській лабораторії штучного інтелекту. Система вводить речення англійською мовою, мова програмування MLISP.

Система «MARGIE» була створена для дослідження методів розуміння та генерації речень природною мовою на основі предметної області «Поведінка людини». Автори системи створили теорію концептуальної залежності на базі, якої розробили програмну систему. Система «MARGIE» перефразує речення природною мовою у концептуальні структури мислення людини, робить умовивід на рівні концептів мислення, після чого перетворює вивід на природну мову. Теорія концептуальної залежності стверджує, що зміст елементарної думки людини залежить від складу понять і типу зв'язків між ними. Зв'язки між поняттями фіксуються концептуальною структурою.

Системи спілкування з БД природною мовою розроблюються для непідготовлених кінцевих користувачів.

Користувачі БД – спеціалісти з різних ПДО і вони не володіють знаннями про логічну структуру БД та мову запитів. Системи

спілкування природною мовою не мають знання про задачу користувача, а тільки полегшують спілкування з БД. Системи спілкування з БД грають пасивну роль у розв'язуванні задачі, а користувач активну.

Відомі такі системи спілкування з БД: «CASPAR», 1981р., мова спілкування англійська; «IRUS» 1984р., мова спілкування англійська, настроюється на БД і тип СУБД; «АИСТ» 1988, спілкування з БД російською мовою, мова програмування АССЕМБЛЕР; «HAM-ANS» багатоцільова діалогова система, мова спілкування німецька, мова програмування UCL-LISP.

Важливою функцією системи «HAM-ANS» є блок настроювання на БД, експертні системи, системи машинного зору. Система «HAM-ANS» створена для дослідження розуміння та генерації речових актів у діалозі. Вона виконує роль портьє готелю.

Системи спілкування з операційною системою. До таких систем відноситься система «MULTIPAR. Система-розуміє команди до ОС англійською мовою (США).

Діалогові системи розв'язку задач завжди мають мету розмови. Метою може бути одержання інформації або розв'язування певної задачі (наприклад обрати мобільний телефон). Існує два підходи до розв'язування задач: знайти оптимальне рішення і знайти рішення, яке підходить користувачеві. Другий підхід застосовують у випадку, коли для рішення задачі не існує методу, за яким можна знайти оптимальне рішення. Такий підхід реалізують експертні системи.

До діалогових систем відносять: «SNUKA» спілкування з експертною системою англійською мовою, 1983р.; «XCALIBUR» – спілкування з експертною системою англійською мовою, 1983р.; «UC» – діалогова система спілкування англійською мовою, США, мова програмування FRANZLISP, 1981; «ADVISOR» – спілкування з експертною системою англійською мовою, 1985р.. Система «ADVISOR» у діалозі застосовує стратегію дискурсу.

Текст, що складається з групи питань і відповідей, пов'язаних змістовим зв'язком, називають дискурсом, якщо для речень характерні однакова стилістика, модальність (відношення до змісту) та час [8]. Діалогова система аналізує дискурс після кожного чергового поповнення дискурсу і виявляє його зміст. Дискурс допомагає давати відповіді на питання. Якщо у реченні природною мовою є помилка,

або у ньому є посилання на попередню інформацію, то система також використовує дискурс.

Системи обробки текстів використовують для дослідження методів розуміння тексту, або використовують для застосування існуючих методів розуміння тексту для практичних цілей.

Наприклад, навчаючі системи утворюють структуру тексту лекцій з певної дисципліни та використовують її для структурування процесу навчання. Системи реферування статей також утворюють структуру тексту статті для швидкого пошуку статті.

До систем обробки текстів відносять системи: «ТАСС» створення рефератів за газетними вирізками, 1983р., спілкування російською мовою; комплекс систем Researcher і Tailor, мова спілкування англійська, система призначена для пошуку рефератів патентів. Система Researcher створює бази знань на основі рефератів – патентів, що описують складні фізичні об'єкти, а система Tailor шукає необхідні реферати в процесі діалогу.

4.1.2 Діалог та його структура

Діалог – процес досягнення учасниками розмови спільної мети шляхом взаємопов'язаного обміну інформацією.

Діалог між програмною системою та людиною має свою специфіку. Людина, що користується діалоговою системою, має метою одержання інформації або розв'язування поставленою нею задачі. Діалогова система має метою сприяти досягненню мети людини.

В процесі спілкування людина або система задають питання, а інший учасник діалогу відповідає на нього. Питання та відповіді, на певний момент утворюють поточний дискурс. Всі речення дискурсу пов'язані між собою логічними, причинно-наслідковими, часовими, лінгвістичними та модальними зв'язками. Модальні зв'язки характеризують відношення людини до сказаного у реченнях дискурсу.

Діалог може завершитися досягненням мети людини або глобальною невдачею. Під час спілкування людини і системи можуть бути локальні тимчасові невдачі. Наприклад, користувач системи та система можуть спілкуватися різною термінологією, мати різні уявлення про проблемну область. Якщо система має засоби роз'яснення термінології, яку вона використовує, може роз'яснювати, як вона бачить проблему користувача, то невдача тимчасова.

Учасник діалогу, що задає питання, подає команду, формулює задачу, грає *активну роль*. Учасник, що відповідає на питання, виконує команду, розв'язує задачу, *грає пасивну роль*.

Ролі користувача та системи можуть бути визначені заздалегідь, і жорстко закріплені між учасниками. Причому один учасник не може грати дві ролі. Кажуть, що в цьому випадку система має *жорстку структуру діалогу*.

Жорстка структура діалогу має місце, якщо активна система, а користувач пасивний, або навпаки. Цікава жорстка структури діалогу з активною системою. Вона має фіксований набір питань, на які одержує відповіді від користувача.

Якщо система задає не всі питання користувачу, що має, а у залежності від попередніх відповідей користувача, то таку структуру діалогу називають альтернативною.

Альтернативна структура діалогу є розвитком жорсткої структури діалогу. При альтернативній структурі діалогу ролі учасників також розподілені заздалегідь і жорстко, а кожне питання системи передбачає множину напрямів діалогу. Форма такого діалогу подається деревом.

Наприклад, система видає запитання і пропонує три відповіді:

Який тип аналізу слова Ви бажаєте виконати?

1. Морфемний
2. Морфологічний
3. Орфографічний

Вибір наступного питання пов'язаний з відповіддю користувача. Питання, що пов'язані з двома іншими типами аналізу не подаються.

Якщо ролі учасників розподіляються при спілкуванні із системою, то структуру діалогу називають гнучкою.

Гнучка структура діалогу дозволяє перехвачувати ініціативу у активного учасника при неясній ситуації. В результаті змінюються їх ролі.

Система може кваліфікувати неясну ситуацію при не збігу змісту питання користувача з питанням, що очікується. Наприклад, користувач задав питання, а система не розуміє його. Тоді система змінює ініціативи учасників і задає питання сама.

Гнучка структура діалогу класифікується за ступенем вільності вибору часу перехвати ініціативи (в певний момент, у будь-який час),

за способом перехвату (розпізнання події). Для інтелектуальних систем характерна гнучка структура діалогу.

Методи представлення текстів для діалогу залежать від структури діалогу. Для жорсткої структури та альтернативної структури тексти заготовлюються заздалегідь. Для гнучкої структури діалогу тексти не складаються. Для такого спілкування необхідні лінгвістичні знання, що дозволяють вільно розуміти природну мову та генерувати речення. Діалогові системи можуть комбінувати структури діалогу. Так при гнучкій структурі діалогу можуть застосовуватися альтернативні питання або жорстко закріпленні питання.

Часто експертні системи застосовують тільки жорстку структуру діалогу. Такі системи зосереджуються на самому процесі розв'язування задачі.

4.1.3 Структура діалогової системи та її функціонування

Діалогові системи застосовують для забезпечення доступу до прикладних систем, що створюються для реального промислового або комерційного застосування. Такі прикладні системи розв'язують задачі різними математичними методами. Але часто зустрічаються задачі, розв'язування яких вимагає розв'язування кількох конкретних підзадач прикладної системи, взятих в певному порядку. В цьому випадку створюють діалогову систему, як інтерфейс прикладної системи.

Діалогова система має лише загальну уяву про задачі прикладної системи. Ця інформація необхідна їй для повідомлення користувачу про можливості прикладної системи, які задачі вона може розв'язувати; для опиту у користувача вихідних даних задач, їх параметрів та обмежень; подавання користувачу проміжних та кінцевих результатів.

Діалогова система складається з двох компонент:

- «Ведення діалогу»;
- «Інтерфейс».

Компонент «Ведення діалогу» є основним компонентом діалогової системи. Компонент реалізує наступні функції.

1. Визначення методу ведення діалогу.
2. Ведення діалогу на протязі роботи всієї системи.
3. Редукцію задачі на підзадачі прикладної системи.

4. Визначення ролі системи та користувача у кожний поточний момент.
5. Визначення процедури, що розв'язує локальну задачу для поточного кроку діалогу та розв'язування задачі.

Компонент «Інтерфейс» виконує такі функції.

1. Перетворення тексту природною мовою на внутрішню мову, яку розуміє система.
2. Генерацію відповіді природною мовою для користувача або пошук відповіді.

Діалоговий компонент повинен забезпечити доцільну поведінку системи для кожного кроку діалогу. Під кроком будемо розуміти пару: «питання-відповідь» або «дія-реакція», які визначаються на локальному рівні діалогу. Дію виконує активний учасник, реагує пасивний учасник.

Інтерфейс повинен забезпечити розуміння відповіді (питання) користувача природною мовою, та генерацію або знаходження питання відповідно попередньо введеної відповіді користувача.

Компонент ведення діалогу організує свою роботу на трьох рівнях:

- глобальному;
- тематичному;
- локальному.

Компонент ведення діалогу на глобальному рівні керує послідовністю виконання етапів роботи діалогової системи.

1. Етапи підготовки до розв'язування задачі.
2. Етап розв'язування задачі.
3. Етапи оцінки розв'язку задачі та його застосування.

Виконання етапів веде до досягнення мети користувача. Глобальна структура діалогу (етапи та їх послідовність) залежить від класу задачі, що розв'язує користувач.

Для добре структурованих математичних задач глобальна структура діалогу містить послідовність завдань, виконання яких веде до розв'язку основної задачі.

Частіше глобальний компонент ведення діалогу простий за структурою та незмінний, тому його розміщують прямо в основну програму ведення діалогу.

На початку роботи програми активним учасником для всіх трьох структур діалогу є система. Надалі для гнучкої структури діалогу активність і пасивність учасників може змінюватися, а для жорсткої і альтернативної структури залишається весь час роботи програми.

Система починає роботу з інструктажу про роботу системи, клас задач, які вона може розв'язати, та ПДО для якої розв'язуються задачі.

Система задає перше питання про ціль роботи, що користувач хоче зробити.

На рис. 4.1 подана схема взаємодії діалогової системи та користувача.

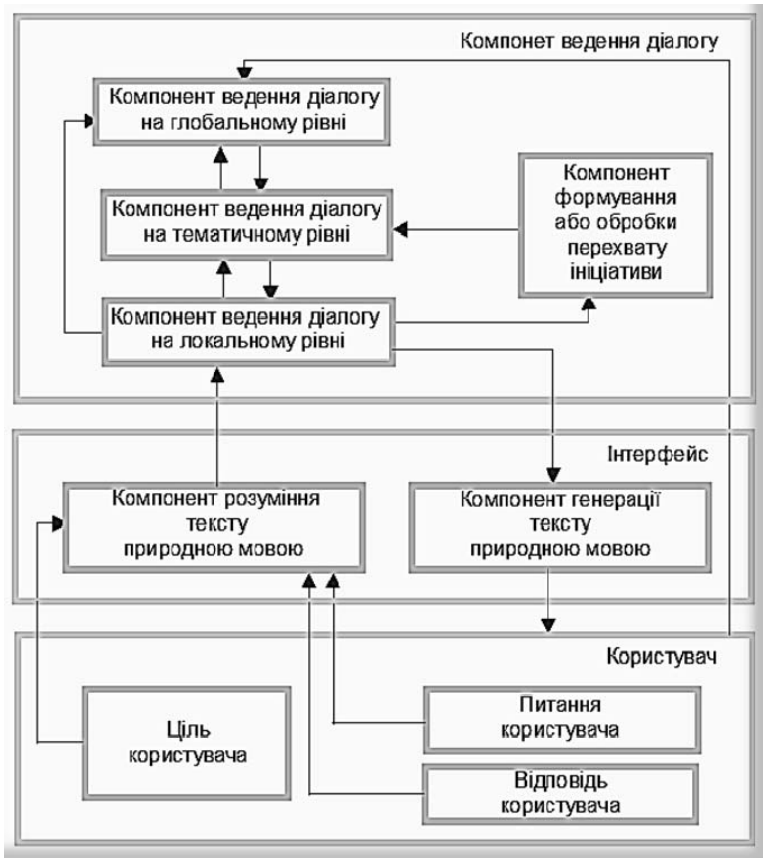


Рисунок 4.1 – Схема взаємодії діалогової системи і користувача

Послідовність і зміст етапів глобального компонента ведення діалогу може задаватися сценаріями. або продукціями.

Сценарії вперше з'явилися у роботах Р. Шенка та Р. Абельсона [9]. Шенк та Абельсон зробили припущення та довели його істинність, що людина, засновуючись на своєму досвіді, для кожної ситуації, що повторюється, створює собі структуру знань – сценарій. Сценарій визначає стандартну послідовності подій, що ведуть до цілі людини. Такі структури знань людина застосовує для прогнозування подій, формування відношення до подій та виробки поведінки.

Наприклад, студенти, які неодноразово здавали іспити, створюють собі сценарій їх проведення. Якщо реальні події не співпадають із подіями, що очікуються, то це викликає в них певну реакцію та поведінку.

У діалогових системах ми будемо розуміти сценарій, як формалізований опис стандартної послідовності подій, пов'язаних часом та каузальними (причинно-наслідковими) зв'язками, що ведуть до цілі. Для глобальної компоненти діалогу сценарій визначає послідовність і зміст етапів, що ведуть до розв'язку задачі.

Для кожного типу задач застосовується свій сценарій. Робота із сценаріями вимагає спеціального програмного забезпечення. Програмісти розглядають сценарій, як абстрактний тип даних. Тип-сценарій має процедури, що забезпечують роботу з компонентами сценарію і дії над кількома сценаріями. Наприклад, сценарії можуть вкладатися один в один.

Покажемо фрагмент програми мовою Visual Prolog, що інтерпретує вміст сценарію, який задає послідовність і зміст етапів глобального компонента ведення діалогу.

У програмі застосовується розповсюджений тип сценаріїв – каузальний сценарій (причинно-наслідковий). Сценами каузального сценарію є: множина посилок, ключова подія, наслідки. У програмі сценарій подається структурою.

```
Global Domains
List = string*
н = назва(string)
з = задача(string)
а = активний_учасник(string)
п = пасивний_учасник(string)
```

```
пос = посилки(list)
ключ = ключова_подія(list)
нас = наслідки(list)
л = лінгвістична_інформація (string)
стр = ксц(н, з, а, п, пос, ключ, нас, л)

Predicates
Nondeterm do (стр)
Nondeterm посилки(list)
Nondeterm ключ(list, string)
Nondeterm наслідки(list, string)
Nondeterm етап(string)
тема(string)

Clauses
do (Стр) :- Стр = ксц(назва(Назва_сц),
        задача(Назва_зад),
        активний_учасник(Акт),
        пасивний_учасник(Пас), посилки(Посилки),
        ключова_подія(Рішення), наслідки(Наслідки),
        лінгвістична_інформація (Файл)),
        write(Назва_сц,'\n', Назва_зад,'\n',
        «Активна», Акт, « », «Пасивний», Пас,'\n', «Лінгвістична
        інформація із файлу», Файл),
        nl, nl, посилки(Посилки),
        ключ(Рішення, Результат),
        наслідки(Наслідки, Результат).
посилки([]).
посилки([H|T]):-write(«Етап», H), nl, посилки(T).
ключ([Рішення], Результат):-write(«Етап», Рішення), nl,
                                тема(Результат).
```



```
наслідки([],_).
наслідки([H|T], Результат):-
        write(«Етап», H), nl, наслідки(T, Результат).

тема(Результат):- Результат = «Задача розв'язана»,!.

етап(«Інструктаж»):-!.
етап(«Видача результату і його_оцінки»):-!.
етап(«Пояснення процесу одержання результату»):-!.

Goal
do(ксц(назва(«Глобальний діалог»), задача(«Експертна система»),
активний_учасник(«Система»),
пасивний_учасник(«Користувач»),
посилки([«Інструктаж»]),
ключова_подія([«Розв'язування задачі з поясненням проміжних
результатів у ході розв'язування»]),
наслідки([«Видача результату і його_оцінки», «Пояснення процесу
одержання результату»]),
лінгвістична_інформація («Лінгв. txt») )).
```

Програма одержує сценарій з цілі, але його можна обирати із файлу за типом задачі. Кожний компонент сценарію задається структурою, а аргумент структури рядком, який визначає етап, назву сценарію, тип задачі, активність учасників.

Посилки, ключова подія та наслідки сценарію вказуються списками рядків, бо кількість їх компонент у різних сценаріях може бути різною. Елементи списку розташовані у порядку виконання відповідних процедур, що реалізують етапи.

Процедура do одержує сценарій і виділяє з нього компоненти. Процедури: посилки, ключ, наслідки по черзі вибирають із своїх списків назви етапів і викликають твердження процедури етап(Тип_етапу) відповідно назві. Кожне твердження процедури

етап(Тип_етапу) реалізує певний етап. Процедура етап(Тип_етапу) містить множину етапів для всіх сценаріїв діалогової системи.

Розв'язування задачі виконується предикатом тема, який належить компоненту ведення діалогу на тематичному рівні, і повертає результат у предикат do, який належить компоненту ведення діалогу на глобальному рівні. Предикат do передає результат у інші етапи обробки результату.

Компонент ведення діалогу тематичного рівня розв'язує конкретну задачу обраного класу. Компонент включається в роботу після виконання етапів підготовки до розв'язку задачі. Після одержання розв'язку задачі компонент повертає керування в компонент глобального рівня діалогу.

Структура компонента ведення діалогу тематичного рівня залежить від типу і особливостей задачі. Особливості задачі визначаються вхідними даними задачі.

Розв'язування задач методом редукції. Задача може розв'язуватися шляхом її декомпозиції (редукцією задачі). На стадії редукції виявляються підзадачі, необхідні для розв'язування задачі. Кожна підзадача може в свою чергу декомпозуватися на менші підзадачі. Процес декомпозиції завершується тоді, коли для розв'язування кожної підзадачі знаходиться конкретна підпрограма або кілька підпрограм прикладної системи, що можуть виконати її розв'язування різними методами. Вибір підпрограми за певним методом виконується у діалозі.

Підзадачі, що розв'язуються підпрограмами прикладної системи називають локальними задачами. З розв'язків локальних задач складається розв'язок підзадач вищого рівня і т. д. Процес завершується одержанням розв'язку основної задачі.

Процес декомпозиції задачі на підзадачі може подаватися по різному: продукціями, сценаріями, універсальним методом Ньюелла тощо. Від вибору методу декомпозиції задачі залежить взаємодія тематичного та локального рівнів діалогу.

Розв'язування стереотипних задач. Діалогові системи орієнтуються на розв'язування стереотипних задач певного класу, а саме, планують дії для розв'язування стереотипних задач. Під стереотипними задачами розуміють добре структуровані задачі, що

мають однакову структуру і відрізняються лише даними. Стереотипні задачі розв'язуються стереотипними діями.

Стереотипні задачі розв'язуються в процесі діалогу з користувачем. Компонент ведення діалогу на локальному рівні одержує ціль, яку треба досягнути. У діалозі виявляються обмеження на розв'язок задачі.

Для розв'язування стереотипних задач компонент тематичного рівня може застосовувати готовий сценарій, що декомпозує задачі цього типу.

Дані одержані при опиті на локальному рівні конкретизують параметри сценарію. У сценарію між подіями встановлюється відношення часу *R1* (бути раніше). В результаті створюється сценарій розв'язування конкретної задачі. Сценарій інтерпретується викликами підпрограм прикладної програми, що розв'язують задачу.

Готові сценарії застосовують для систем, виконання яких вимагає високої швидкості роботи.

Сценарій може формуватися програмно. Для цього у програмі повинен реалізовуватися механізм планування сценаріїв. Методи планування дозволяють встановлювати цілі та будувати плани дій. Динамічне створення сценаріїв застосовують у випадках, коли структура задачі залежить від особистості користувача. До таких систем можна віднести діалогові експертні системи. Задачі обробки текстів, що пов'язані змістом, також вимагають генерації сценаріїв, бо кожен текст унікальний.

Компонент ведення діалогу локального рівня організує роботу кожного кроку діалогу.

Крок утворюється процедурами, що оброблюють взаємно пов'язані речення учасників діалогу (питання-відповідь).

Компонент локального рівня діалогу працює разом з компонентами, що виявляють зміст речень на природній мові або утворюють речення на природній мові. Зміст речення на локальному рівні уточнюється у контексті дискурсу.

Компонент локального рівня виявляє активного діяча кроку (хто розв'язує задачу), вид дії (що робить діяч), спосіб впливу дії (як діє), специфікацію підзадачі (обмеженість на вибір підпрограми, конкретизацію параметрів підпрограми). Всі ці характеристики кроку передаються у компоненту ведення діалогу тематичного рівня.

Компонент формування або обробки перехвату ініціативи. При гнучкій структурі діалогу система у кожний поточний момент може бути активною чи пасивною. В діалоговій системі є два типа компонент: система активна та система пасивна. Кожен учасник діалогу може бути активним чи пасивним.

Активний учасник виконує якусь дію, наприклад, ставить запитання. Пасивний учасник реагує на цю дію, наприклад, відповідає на поставлене запитання чи перехоплює ініціативу і ставить сам запитання. Цим самим пасивний учасник стає тимчасово активним учасником. Одержавши відповідь, він повертає ініціативу іншому учаснику. Тому в системі є компонент перехвату ініціативи – компонент «Обробка перехвату ініціативи».

Перехоплення ініціативи здійснюється також в тому випадку, якщо треба уточнити питання, змінити тему бесіди і т. д.

Залежно від відповіді користувача, обирається підпрограма, яка розв'язує локальну задачу. Таким чином, локальний рівень діалогу забезпечує рішення локальної задачі, що веде до одержання рішення всієї задачі на тематичному рівні.

4.2 Універсальний метод розв'язування задач Ньюелла

Декомпозицію задачі на підзадачі можливо виконати маючи загальні відомості про задачу. До методів, що реалізують декомпозицію, відноситься універсальний метод розв'язування задач Ньюелла [7].

На базі цього методу у 1957 році була створена універсальна програма розв'язування задач GPS (General Problem Solver). Програма могла розв'язувати будь-які символічні формалізовані задачі: логічні, геометричні задачі, доведення теорем, гру в шахи. Програму реалізували на мові програмування низького рівня IPL.

Для функціонування універсальної програми потрібно було мати опис вихідних даних задачі, опис розв'язку задачі та оператори, що поступово перетворюють вихідний опис у розв'язок. Діалогова система одержувала ці описи від користувача і передавала їх в універсальну програму для розв'язку задачі. Описи передавалися у внутрішньому виді.

Програма GPS розв'язувала вихідну задачу методом редукції. Задача розв'язувалася одним оператором. Оператор знаходився за *найменшою відмінністю* між ціллю всієї задачі і вихідними даними. Під відмінністю розуміють будь-яку властивість об'єктів задачі. Найменша відмінність визначала оператор, виконання якого давало розв'язок задачі.

Але кожен оператор міг виконатися тільки при наявності певних умов. Тому наявність кожної умови виконання обраного оператору ставала підзадачею, яку треба розв'язати попередньо. Якщо умов було кілька, то і підзадач теж було кілька. При цьому множина цілей поповнювалася цілями кожної підзадачі.

Для знаходження чергового оператору, що може розв'язати підзадачу, програма кожен раз заново знаходила найменшу відмінність між поточним описом вхідних властивостей об'єктів та їх кінцевих станів для задачі (множини поточних цілей). Дії повторювалися для кожної підзадачі. Останні підзадачі могли розв'язатися операторами безпосередньо унаслідок наявності вхідних даних. Тому першими виконувалися ці оператори, їх виконання забезпечувало виконання операторів, що розв'язували підзадачі вищого рівня і т. д. до розв'язку всієї задачі. Тобто задача розв'язувалася методом редукції.

Таким чином, розв'язок всієї задачі знаходився поступовим зменшенням відмінностей властивостей вхідних об'єктів від їх кінцевих станів, за рахунок чого виконувалося перетворення вхідного опису у кінцевий.

Розглянемо роботу універсального методу розв'язування задач на прикладі задачі запропонованої Дж. Маккарті (творцем мови Lisp) у 1963 році.

Задача про мавпу та банани. У кімнаті знаходиться мавпа, скриня і низка бананів, що підвішена до стелі. Мавпа може рухатися по кімнаті, рухати скриню і забиратися на неї. Мавпа не може дістати банани не забравшись на скриню, що розташована під бананами. Як мавпі дістати банани?

У задачі є один діяч «мавпа», об'єктом дій мавпи є «банани». Для досягнення цілі діяч повинен мати інструмент «скриню» і вміти виконувати наступні дії: «йти» до скрині, «рухати» скриню під банани, «піднятися» на скриню, «хапати» банани. Для опису будемо

застосовувати певні місця: «двері»,«вікно»,«під_банани»,«на_скриня», хоча їх можна і фіксувати просто буквами. Застосуємо також факти, що описують діяча, інструмент та об'єкт дії.

Опишемо задачу та загальні відомості фактами мови Prolog.

```
опис(«місце», «мавпа», «двері»)
опис(«місце», «скриня», «вікно»)
опис(«вміст», «рука мавпа», «порожня»)
ціль(1, «вміст», «рука мавпа», «банани»)
діяч(«мавпа»)
інструмент(«скриня»)
обєкт_дії(«банани»)
```

Програміст, що захоче розв'язати таку задачу програмно, зафіксує порядок дій у програмі. Таку програму написати дуже легко. Реалізація програми універсальним методом дозволяє не фіксувати послідовність дій, а знаходити їх послідовно зменшуючи різницю між ціллю і фактами «опис».

Програмна реалізація стратегії універсального методу розв'язання задач мовою Visual Prolog для задачі про мавпу і банани подана нижче.

```
Global Facts
Nondeterm опис(string, string, string)
Nondeterm ціль(integer, string, string, string)
Nondeterm різниця(string, string, string)
Nondeterm діяч(string)
Nondeterm інструмент(string)
Nondeterm обєкт_дії(string)
Single m(integer)

Predicates
Nondeterm знайти_різницю()
Nondeterm перетворити()
Nondeterm зменшити_різницю(integer)
Nondeterm порівняти(string, string , string)
Nondeterm оператор(integer, string)
Nondeterm цілі()
Nondeterm кінцевий_опис()
```



```
Nondeterm do()

Goal
do ().

Clauses
m(1).
зменшити_різницю(1):- діяч(S), опис(«місце», S, P1),
        інструмент(C), опис(«місце», C, P2),
        порівняти(«місце», S, P2),
        m(M), M1=M+1, опис(«місце», C, P2),
        assert (ціль (M1,»місце», C, P2)),
        write («Розв'язати підзадачу», M1, «  », «місце»,
         C, « », P2), nl,
        M2=M1+1, assert(m(M2)),
        assert (ціль (M2, «місце», S, P1)),
        write («Розв'язати підзадачу», M2, «  »
        «місце», S, «  », P1), nl,!.
зменшити_різницю(2):- діяч(S), інструмент(C),
        обєкт_дії(B), concat(«під_», B, B1),
        порівняти(«місце», C, B1),
        порівняти(«місце», S, B1),
        m(M), M1=M+1, assert(m(M1)),
        опис(«місце», C, P2),
        assert (ціль (M1, «місце», S, P2)),
        write («Розв'язати під задачу», M1,
        «  », «місце», S, «  », P2), nl.
зменшити_різницю(3):- діяч(S), інструмент(C),
        опис(«місце», C,_), concat(«на_», C, C1),
        порівняти(«місце», S, C1),
        обєкт_дії(B), concat(«під_», B, B1),
        m(M), M1=M+1, assert(m(M1)),
        assert (ціль (M1, «місце», S, B1)),
        write («Розв'язати підзадачу», M1, «», «місце»,
         S, «», B1), nl,!.
зменшити_різницю(4):- опис(«вміст», R, _), обєкт_дії(B),
        порівняти(«вміст», R, B),
```

```
        діяч(S), інструмент(C),
        concat(«під_», B, B1),!,
        concat(«на_», C, C1), m(M), M1=M+1,
        assert (ціль (M1, «!місце», C, B1)),
        write («Розв'язати підзадачу», M1, «», «місце»,
         C, «» , B1), nl,
        M2=M1+1, assert(m(M2)),
        assert (ціль (M2, «місце», S, C1)),
        write («Розв'язати підзадачу», M2, «»,
        «місце», S, «», C1), nl,!.

порівняти(V1, Ob11, Ob21):- різниця(V, Ob1, Ob2),
        ціль(_, V1, Ob11, Ob21), V=V1,
        Ob1=Ob11, Ob2=«_»,!;
        різниця(V, Ob1, Ob2),
        ціль(_, V1, Ob11, Ob21), V=V1,
        Ob1=«_», Ob2=«_»,!;
        різниця(V, Ob1, Ob2),
        ціль(_, V1, Ob11, Ob21), V=«_»,
        Ob1=«_», Ob2=«_»,!.

оператор(1, «йти»):- діяч(S), опис(«місце», S, P1),
        інструмент(C), опис(«місце», C, P2),
        P1<>P2, retract(опис(«місце», S, P1)),
        assertz (опис («місце», S, P2)),!.

оператор(2, «рухати»):- діяч(S), опис(«місце», S, P1),
        інструмент(C), опис(«місце», C, P2),
        P1=P2, обєкт_дії(B),
        concat(«під_», B, B1), P1<>B1,
        retract (опис («місце», S, _)),
        assertz (опис («місце», S, B1)),
        retract (опис («місце», C, _)),
        assertz (опис («місце», C, B1)), !.

оператор(3, « піднятися»):- діяч(S), опис(«місце», S, P1),
        інструмент(C), опис(«місце», C, P2),
        P1=P2, обєкт_дії(B),
        concat(«під_», B, B1), P1=B1,
        concat(«на_», C, C1),
```



```
        retract (опис («місце», S, _)),
        assertz (опис («місце», S, C1)), !.

оператор(4, «хапати»):- діяч(S), опис(«місце», S, P1),
        інструмент(C), опис(«місце», C, P2),
        concat(«на_», C, C1), P1=C1,
        ціль(_, «вміст», R, B),
        опис(«вміст», R, St), St <> B,
        concat(«під_», B, B1), P2=B1,
        retract (опис («вміст», R, St)),
        assertz(опис(«вміст», R, B)), !.

перетворити():- ціль(_, V, Ob1, Ob2), опис(V, Ob1, Ob2),!.
перетворити():-  зменшити_різницю(N), !, перетворити,
        оператор(N,_).
перетворити():-write(«Неможливо розв'язати задачу»),
        exit ().

знайти_різницю():-write(«відмінності описів і цілей:»),
        nl, nl, ціль(_, V, Ob1, Ob2),
        опис(V, Ob1, Ob21), Ob2<>Ob21,
        assert (різниця (V, Ob1, «_»)),
        write («різниця», «(«, V» «, Ob1,
        « ,», _ «,»)»), nl, fail;
        ціль(_, V, _, _), опис(V1, _, _),
        V<>V1,
        assert (різниця («_», «_», «_»)),
        write («різниця», «(«, «_», «,», «_», «,», «_»,»)»), nl.

знайти_різницю().

цілі():- nl, write(«Створюються цілі :»), nl, nl,
        ціль(M, V, O1, O2),
        write (M, «», V, «», O1, «», O2), nl, fail.
цілі():- nl, кінцевий_опис().

кінцевий_опис ():- write(«Кінцевий опис :»), nl, nl,
                опис(V, O1, O2),
                write (V, «», O1, «», O2), nl, fail.
кінцевий_опис().
```

```
Do ():-consult («meta. txt»), знайти_різницю (),
        ціль(M, V, Ob1, Ob2), nl,
        write («Декомпозиція задачі :»), nl, nl, write («Розв’язати задачу»),
        M,
        «», V, «», Ob1, «», Ob2), nl, перетворити(),
        цілі(), !, nl.
```

Програма реалізує метод редукції низхідною рекурсією. На початку роботи програма завантажує опис задачі. Процедура знайти_різницю знаходить відмінності між ціллю задачі і вихідними даними задачі.

```
опис(«місце», «мавпа», «двері»)
опис («місце», «скриня», «вікно»)
опис («вміст», «рука мавпа», «порожня»)
ціль(1, «вміст», «рука мавпа», «банани»)
```

Ціль задачі може описуватися кількома фактами. Тому процедура знайти_різницю не фіксує одну ціль. Процедура розглядає різні варіанти відмінностей цілей від вихідних даних і зберігає їх для подальшого застосування.

Спочатку роботи програма аналізує відмінності початкового опису від цільового:

```
різниця(вміст, рука мавпа,_)
різниця(_,_,_)
```

Найменша відмінність цілі у третьому факті з описом. Інші факти не зрівнянні з ціллю. Якщо різниці не має, то програма закінчує роботу.

Декомпозиція задачі починається з виводу цілі задачі: «Розв’язати задачу 1 вміст рука мавпа банани.».

Процедура перетворити реалізує метод редукції низхідним методом рекурсії. Вона застосовує для розбивання задачі на підзадачі процедуру зменшити_різницю і для розв’язування задачі процедуру оператор. Процедура перетворити завершує свою роботу по досягненні цілі задачі або при неможливості досягнути ціль задачі. Досягнення цілі задачі відбудеться, коли виконається дія, що додасть факт опис відповідний цілі задачі.

Процедура оператор містить кілька правил Прологу. Правила процедури оператор ідентифікуються назвою дії та номером дії. Кожне правило описує умови виконання дії за назвою і при виконанні умов виконує цю дію. Дія правила виконується заміною факту, що описував певний попередній стан об’єкту на факт, що описує стан об’єкту після виконання дії.

Процедура зменшити_різницю також містить кілька правил Прологу. Правила процедури зменшити_різницю і правила процедури оператор взаємопов’язані за змістом і номером оператору.

Кожне правило процедури зменшити_різницю перевіряє, чи може відповідне йому правило процедури оператор після виконання своїх дій зменшити різницю між поточними цілями і поточним станом задачі. Перевірка здійснюється процедурою порівняти. Якщо зменшити різницю не вдається, то розглядається наступне правило процедури зменшити різницю. При виявленні можливості зменшити різницю умови відповідного правила процедури оператор розглядаються як поточні цілі. Поточні цілі поповнюють існуючі цілі. Одночасно видається повідомлення про створення підзадач.

По досягненню правила 1 процедури знайти_різницю виявляється, що дві підзадачі розв’язуються тривіально – наявністю фактів опис, аргументи яких відповідають цілям:

ціль((«місце», S, P1) опис(«місце», S, P1),

ціль((«місце», C, P2) опис(«місце», C, P2).

Тому процес розбивання закінчується і починається процес розв’язування задачі процедурою оператор. Послідовність виконання правил процедури оператор відповідає знайденим найменшим відмінностям.

Результати роботи програми містять: відмінності; декомпозицію задачі; цілі, одержані в результаті розв’язування задачі; кінцевий опис.

Декомпозиція задачі :

Розв’язати задачу 1 вміст рука мавпа банани
Розв’язати підзадачу 2 місце скриня під_банани
Розв’язати підзадачу 3 місце мавпа на_скриня

Розв’язати підзадачу 4 місце мавпа під_банани
Розв’язати підзадачу 5 місце мавпа вікно
Розв’язати підзадачу 6 місце скриня вікно
Розв’язати підзадачу 7 місце мавпа двері
Програма створює наступні цілі :
1 вміст рука мавпа банани
2 місце скриня під_банани
3 місце мавпа на_скриня
4 місце мавпа під_банани
5 місце мавпа вікно
6 місце скриня вікно
7 місце мавпа двері
Кінцевий опис:
місце скриня під_банани
місце мавпа на_скриня
вміст рука мавпа банани

Резюме

Існують наступні типи природно-мовних систем: «питання-відповідь» для дослідження методів спілкування природною мовою; діалогові для розв’язування задач; спілкування з БД і з ОС, щоб зробити комфортним людино-машинне спілкування; системи обробки текстів для машинного розуміння та генерації текстів.

Всі типи систем переважно присвячені одній проблемі – розумінню змісту текстів природною мовою. Проблема генерації речень природною мовою розглядається у роботах дослідників штучного інтелекту значно менше.

Основною ціллю спілкування людини і системи в діалозі є ціль людини, ціль системи допомогти людині осягнути ціль. У роботі системи може бути невдача – глобальна або локальна. Щоб обійти локальну невдачу система має спеціальні засоби.

Діалог може мати жорстку, альтернативну та гнучку структури. При гнучкій структурі діалогу активність людини та системи може змінюватися, при жорсткій та альтернативній структурі діалогу ролі постійні.

Діалогова система складається з компонент: ведення діалогу та інтерфейсу. Компонент ведення діалогу розглядається на трьох

рівнях: глобальному, тематичному, локальному. Глобальний рівень фіксує етапи діалогу. На тематичному рівні розв’язується задача. Локальний рівень вводить та оброблює поточні питання та відповідь. Локальний рівень пов’язаний з інтерфейсом, що виявляє або генерує зміст речення.

Компонент формування та перехвату ініціативи формує активність системи і користувача, а при локальній невдачі змінює їх активність.

Програма GPS застосовувала універсальний метод розв’язування задач. Вона могла розв’язувати будь-які символічні формалізовані задачі. Універсальний метод розв’язування задач Ньюелла дозволяє розв’язувати задачу поступовим зменшенням відмінностей вхідних об’єктів від їх кінцевих станів, за рахунок чого виконувалося перетворення вхідного опису у кінцевий.

Контрольні питання

1. Чим відрізняється природно-мовна система «питання-відповідь» від діалогової системи?
2. Дайте визначення поняття діалог.
3. Який варіант розподілення ролей при жорсткій структурі діалогу не розглядається і чому?
4. В чому переваги гнучкої структури діалогу від жорсткої та альтернативної структури?
5. При яких обставинах під час діалогу система вважає ситуацію неясною?
6. З яких компонент складається діалогова система?
7. Які функції виконує компонент ведення діалогу?
8. Які функції виконує компонент ведення діалогу на глобальному рівні?
9. Які функції виконує компонент ведення діалогу на тематичному рівні? Якими способами можна розв’язати задачу?
10. Яку задачу називають локальною? Чи може локальну задачу розв’язувати користувач? Придумайте приклад.
11. Які функції виконує компонент ведення діалогу на локальному рівні?
12. Чому метод Алана Ньюелла називають універсальним?
13. Для чого в програмі, що розв’язує задачу про мавпу та банани вводяться діяч, інструмент та об’єкт дії?

Вправи

1. Дослідити програму, що розв'язує задачу про мавпу і банани. Для цього задавайте інші описи задач.

2. Додайте в програму, що розв'язує задачу про мавпу і банани один оператор. Сформулюйте відповідну ціль і розв'яжіть задачу.

4.3 Інтерфейси природною мовою

4.3.1 Основні підходи до створення інтерфейсів

Інтерфейс діалогової системи повинен мати компоненти, що забезпечують спілкування з користувачем природною мовою. Ця вимога пов'язана не тільки з комфортністю спілкування.

При перехваті ініціативи у системи користувач може задавати питання природною мовою. Але, передбачити які питання буде задавати користувач та якими словами, неможливо. Додаткове спілкування може знадобиться в будь-якому місці діалогу.

Інтерфейс може бути інтелектуальним або імітувати інтелектуальний інтерфейс. Інтелектуальність інтерфейсу залежить від методів, які застосовує система для спілкування з користувачем.

Інтерфейси, що імітують інтелектуальність. У таких інтерфейсах система певним способом інтерпретує зміст речення і відповідно одержаній інтерпретації знаходить відповідь на питання користувача або задає питання. Прикладами таких методів спілкування є «метод ключових слів» і «метод шаблонів».

Метод ключових слів базується на виявленні ключових слів фраз, що поступають у систему, і знаходженні фраз за ключовими словами у системі для відповіді.

Метод шаблонів є узагальненням методу ключових слів. Метод був розроблений фірмою Borland і застосований у системі GEOBASE. Система GEOBASE є БД про географію США. База даних та запити до неї реалізовані природною мовою.

Система перетворює питання користувача у внутрішню форму запиту до БД. Перетворення виконується поступово.

Спочатку відкидаються слова неістотні для запиту. Цей процес реалізується за допомогою шаблонів, які накладаються по черзі на питання. Кожен шаблон – це правило Прологу, яке залишає лише ключові для бази даних слова у певній формі. Серед шаблонів відбирається той шаблон, якій найбільше відповідає питанню.

Сформований шаблоном результат – це первинна нормалізована форма запиту, яка передається синтаксичному аналізатору.

Синтаксичний аналізатор має зразки для внутрішнього подавання запитів. Запит може подаватися комбінацією зразків. Синтаксичний аналізатор перетворює нормалізовану форму запиту у внутрішню форму запиту, застосовуючи комбінацію зразків.

За запитом система збирає необхідні дані з різних фактів БД. Факти, що видаються користувачу, відрізняються від змісту фактів, що зберігають ці дані. Система формує з даних нові факти, що виводяться для користувача.

Інтелектуальний інтерфейс. Інтелектуальний інтерфейс вводить речення природною мовою, виявляє його зміст, формалізує зміст і подає внутрішньою мовою. Людина оперує завжди неповними реченнями, тому одержаний зміст інтерпретується на базу знань для поповнення. При інтерпретації на базу знань ураховується дискурс як контекст речення. Одержаний зміст передається локальній компоненті ведення діалогу.

Для передачі інформації користувачу інтелектуальний інтерфейс одержує від локального компонента формалізований зміст майбутнього речення, генерує речення природною мовою, збагачує речення природними словами.

Для реалізації процесу розуміння змісту речення існують такі основні підходи:

- граматичний;
- концептуальний;
- семантичний.

Граматичний підхід. Існує різниця між значенням слова, яке подається у тлумачному словнику, і змістом слова. Наприклад, значенням слова «вікно» у тлумачному словнику є «Рама із склом.». Свій зміст слово набуває від інших слів речення, в якому воно застосовується. У реченні «Зараз хлопець спокійно стояв біля вікна.» слово вікно застосовується як місце, де знаходиться хлопець, а значення слова другорядне для змісту речення. Воно тільки уточнює місце знаходження хлопця.

Суть граматичного підходу можна подати кількома реченнями. Всі слова речення природною мовою пов'язані у змістові групи.

Головною змістовою групою є група, що містить дію. Інші змістові групи характеризують головну групу, їх називають семантичними відмінками. Змістовим групам речення відповідають його граматичні групи. Виявлення змістових груп базується на виявленні граматичних груп: членів речення та частин мови. Звідси назва граматичний підхід.

У реченні «Зараз хлопець спокійно стояв біля вікна.» головною змістовою групою є дія «спокійно стояв». Змістова група з одного слова «хлопець» вказує на семантичний відмінок «діяч». Змістова група «біля вікна» вказує на «локутивний» семантичний відмінок (відмінок місця дії), а змістова група «зараз» вказує на відмінок часу.

У реченні дії відповідає присудок речення; діячу відповідає підмет речення; часу та місцю дії відповідають обставина часу та додаток. Виявлення членів речення на базі частин мови складна задача.

Розглянемо, як при граматичному підході компонент розуміння змісту речення виконує основні функції інтелектуального інтерфейсу.

Виділити зміст речення означає – виявити граматичні групи; визначити, яку змістову роль грає кожна граматична група слів у реченні.

Під формалізацію представлення змісту речення розуміють таке подавання дії та семантичних відмінків, що для будь-якого речення дія та кожний тип семантичного відмінку мають фіксоване місце. Якщо у реченні певний семантичний відмінок відсутній, то його місце фіксується порожнім рядком. Таким чином всі змістові структури речень однакові. Формалізація змісту дозволяє кожній процедурі, що оброблює конкретний семантичний відмінок, легко обирати свої дані із структури.

Вказані функції реалізуються на морфологічному, синтаксичному, семантичному етапах.

На морфологічному етапі для кожного слова речення виявляються його морфологічні характеристики. Для цього застосовують словникові методи, безсловникові методи, їх варіації.

Найпростіший словниковий метод – це декларативний метод. В ньому застосовується словник готових форм, в якому біля кожного слова (іншої лексеми тексту) міститься множина його лексико-граматичних характеристик (частини мови, рід, число, відмінок, особу, час, істотність, пасивність тощо). Але такий словник дуже великий, бо кожне слово міститься у всіх його можливих формах. Проте головний недолік в тому, що складання такого словника дуже

важка задача. Тому стали застосовувати декларативний метод разом з процедурним методом.

У процедурному методі всі слова розподіляються на три групи: слова, що не змінюються; слова у яких змінюється основа; слова, у яких не змінюється основа. Перші дві групи слів оброблюються за допомогою словника готових форм, а третя група (найбільш велика) оброблюється процедурним методом. При процедурному методі застосовуються словники основ та закінчень для перевірки, чи складається слово з певних основи та закінчення. Словники основ та закінчень містять кожен свої морфемні характеристики (тому і слово має ці характеристики. Але цей метод дає неоднозначний результат, бо одне слово може мати різні характеристики. Наприклад, слово «комп'ютер» зустрічається у називному та знахідному відмінках.

У безсловникових методах виконується по буквений аналіз слова за деревом, за яким визначаються можливі у мові наступні букви і можливі відповідні характеристики слова. У цього метода ти самі недоліки. На практиці словники та дерева зменшують за обмеженим словниковим запасом речень ПДО.

На синтаксичному етапі, застосовуючи лексико-морфологічні характеристики, виявляється тип вхідного речення (його структура). Тобто, виявляється, з яких простих речень складається складне речення, з яких граматичних груп складається просте речення (груп іменників, групи дієслів), та зв'язок між простими реченнями та граматичними групами. Структуру речення засновану на частинах мови називають – деревом граматичного розбору.

Структура речення впливає на зміст речення. Класичним прикладом, як структура речення впливає на його зміст, є давній приклад російського речення, який наводили ще в гімназіях XIX століття. У реченні «Казнить нельзя, помиловать» зміст речення залежить від положення коми, яка виділяє граматичні групи речення. Порівняйте зміст речення з таким «Казнить, нельзя помиловать».

Застосовуючи дерево граматичного розбору виявляється структура речення, що базується на членах речення (підметі, присудку, додатках, обставинах, союзах, розділових знаках). Така структура ближче до змісту речення ніж дерево граматичного розбору. Присудок вказує на дію, обставина часу на час дії і т. д..

Граматичні групи називають іменними групами, їм дається ім'я за головним словом групи слів. Наприклад: група іменника, група дієслова або група підмета, група присудка.

На семантичному етапі із структури речення, що базується на членах речення, виявляються дія та семантичні відмінки і будується поверхнева структура речення. Поверхнева структура подає зміст речення в формалізованому виді.

Інтерпретація формалізованого змісту речення на базу знань реалізується для розширення змісту речення. База знань грає головну роль у розумінні вхідних речень. Відомий спеціаліст штучного інтелекту Стенфордського університету (США) Р. Шенк, сказав: «Розуміти – означає пов'язати нову інформацію з вже відомою». Під «новою інформацією», розуміють формалізований зміст речення природною мовою. «Відома інформація» – знання, які має інтелектуальна система.

Для зіставлення змісту речення з базою знань знання повинні представлятися також словами та їх змістом, який виражається через зв'язки з іншими слова. Прикладом реалізації граматичного підходу є система «Поет», спілкування російською мовою, мова програмування ПЛ/1).

Концептуальний підхід. При концептуальному підході намагаються реалізувати дії, які виконує людина при розумінні змісту речення. Концептуальний підхід засновано на ідеї, що структури мови і структури думок різні. Базовою одиницею речення є слово, а базовою одиницею думки є концепт (поняття).

У 80-х роках у Стенфордському університеті працювала група дослідників штучного інтелекту, яка розробила теорію концептуальної залежності(ТКЗ). Дослідники стверджують, що *думка існує незалежно від мови, тобто, що існує мова мислення* [10].

За теорією зміст будь-якого речення природною мовою, в пам'яті людини не залежить від мови (не відображується словами), а відображується концептуальною структурою, що складається з концептів і відношень між ними, концептуальні структури називають концептуалізаціями. Концептуалізації будуються за своїми правилами концептуального синтаксису. Причому, різні речення з однаковим змістом, відображаються однією концептуальною структурою. Кожен

концептуальний вираз розуміється однозначно. Концептуалізації можуть пов'язуватися відношеннями в один концептуальний граф.

У мовах дії часто тільки маються на увазі. Наприклад, дієслово «сидіти» означає кінцевий стан людини, але мається на увазі дія «сідати». Аналогічно іменники і прикметники також можуть вказувати на дію. Наприклад, іменники «їстівне», «робота» припускають дії. Концептуалізації засновуються тільки на діях. Дослідники виявили, що на концептуальному рівні для подавання змісту будь-якої дії для ПДО «Поведінка людини» достатньо одинадцяти елементарних актів. Кожна дія на концептуальному рівні подається одним елементарним актом чи комбінацією кількох елементарних актів з одинадцяти. Але є припущення, що кількість елементарних актів для подавання будь-яких ПДО буде цього ж порядку.

У межах цього посібника не входить детальне розглядання теорії концептуальної залежності, все ж цікаво розглянути приклади концептуалізацій.

Нехай є два речення з різними дієсловами.

1 Джон дав Мері книгу.

2 Мері взяла у Джона книгу.

Відповідно реченням побудуємо їх концептуалізації. Щоб відрізняти ідентифікатор концепту від слова використовують на початку ідентифікатору концепту знак #.

1 (ATTRANS #Джон #книга #Джон #Мері)

2 (ATTRANS #Мері #книга #Джон #Мері)

Концептуальні структури, що виражають зміст речень використовують той самий елементарний акт ATTRANS. Обидві концептуалізації з актом ATTRANS показують: спочатку книга була у Джона, а потім у Мері. У першому реченні діячем був Джон, у другому Мері. Книга є об'єктом дії. Концептуалізація з актом ATTRANS завжди показує зміну абстрактного відношення об'єкту дії до суб'єкту або іншого об'єкту.

У концептуалізації відношення між її концептами називають концептуальними відмінками дії. Концептуальні відмінки

модифікують дію. У першій концептуалізації відношення між дією і Джоном є концептуальним відмінком, який вказує на діяча дії. Переміщення об'єкту від когось до одержувача називають реципієнтним відмінком. Реципієнтний відмінок уточнює дію. Слово реципієнт (латинська recipiens) означає одержувач.

Вказані концепти це ідеї (думки). Концепт у пам'яті позначається набором властивостей.

З прикладів видно, що *зміст дієслова залежить не тільки від базового змісту елементарного акту, а і від змісту інших концептів концептуалізації.* Якщо діяч і реципієнт співпадають, то вживається дієслово «взяв», інакше дієслово «дав».

Прикладом системи, що реалізувала концептуальний підхід до розуміння змісту речень природною мовою є система MARGIE. Система складається з трьох частин: концептуального аналізатора, концептуальної пам'яті і умовиводу, концептуального генератора.

Концептуальний аналізатор компілює тексти природною мовою (англійською) у концептуальні графи. Концептуальна пам'ять і умовивід демонструють здібність до елементарного мислення. Концептуальний генератор створює з концептуального графу текст природною мовою.

Семантичний підхід. Семантичний підхід розуміння текстів природною мовою базується на змісті слів. Слово «семантика» означає «зміст одиниці мови – слова». Звідси назва підходу «семантичний підхід». Існують різні реалізації семантичного підходу.

Одна із самих перспективних реалізацій семантичного підходу є реалізація лінгвістичного процесору для складних інформаційних систем, що була розроблена і виконана групою дослідників російської академії наук інституту проблем передачі інформації під керівництвом Апресяна Ю. Д. у 1992році [11]. Дослідники реалізували програмну систему «лінгвістичний процесор» базуючись на наступних положеннях.

Людина виконує два типи мовної діяльності подавання думок текстом і розуміння текстів. Реалізація першого типу діяльності виконується шляхом трансляції структур мови думок у структури природної мови. Процес розуміння тексту природною мовою

виконується також шляхом трансляції структур природної мови у структури мови думок.

Для реалізації перекладу з однієї мови на іншу була розроблена *формальна функціональна модель мовної поведінки людини*. Тобто, групою Апресяна вперше була поставлена фундаментальна *лінгвістична задача: моделювання мовної поведінки людини*.

Програмну систему, що реалізує модель мовної поведінки людини розроблювачі назвали лінгвістичним процесором. Формальна модель дозволяє робити багатомовні переклади; застосовувати різні природні мови. Новизна лінгвістичного процесору в тому, що він працює незалежно від змісту тексту природною мовою, точно передає його не зважаючи на складність тексту.

Розроблювачі застосовували лінгвістичний процесор для спілкування з БД природною мовою. При розумінні речень тексту виконується їх аналіз, при формуванні тексту виконується синтез речень. Ми розглянемо процес аналізу речень тексту. Виконуючи аналіз, лінгвістичний процесор перетворює речення тексту природною мовою у внутрішнє подавання – універсальну мову запитів до баз даних SQL. Перетворення виконується поступово за етапами:

- морфологічним;
- синтаксичним;
- семантичним.

Для кожного етапу були розроблені: своя мова *формального* подавання: статей словників (морфологічного, комбінаторного, семантичного); формальних синтаксичних і трансформаційних правил перетворення структур речень.

На морфологічному етапі виконується перетворення кожного речення тексту у лексико-морфологічну структуру. *Кожну форму слова, розділовий знак, число, інші лексеми називають словоформою*. У морфологічному словнику розміщують основи слів та їх морфологічні характеристики для створення всіх можливих словоформ. Причому кожна характеристика прив'язується до своєї морфеми слова (основи, суфіксу, закінчення тощо).

Таке подавання лексико – морфологічних характеристик слів у морфологічному словнику дозволяє створювати всі варіанти словоформ, щоб порівнювати їх із відповідним словом речення та

створювати нормальну форму слова. Нормальна форма застосовується для однозначного подавання будь-якої словоформи одного слова. Нормальною формою іменника вважають слово у називному відмінку однини (якщо воно не вживається тільки у множині).

У лексико-морфологічній структурі слово подається основою разом з групою можливих для нього лексико морфологічних характеристик.

Наприклад, слово «студенти» буде містити:

– (основа: студент (імен., істотн.);
– суфікс: #();
– закінчення: и (мн., ч., наз.)),

де – # означає відсутність характеристики

Вся лексико-морфологічна структура речення – це лінійна послідовність основ слів та розділових знаків речення, розташованих у тому ж порядку як в реченні, з їх лексико-морфологічними характеристиками.

На синтаксичному етапі виконується перетворення лінійної лексико-морфологічної структури речення у синтаксичну структуру – дерево залежностей (за формою пов'язаний граф без циклів).

У вузлах дерева знаходяться основи слів речення природної мови разом з їх характеристиками, що одержані на морфологічному етапі. Дуги, які пов'язують вузли, відповідають певним синтаксичним зв'язкам між словами речення. Зв'язки формуються на базі лексико-морфологічних характеристик слів. Важливим для змісту речення є синтаксичне відношення предикативності, що пов'язує підмет і присудок. Предикативність робить речення повідомленням. Всього у лінгвістичному процесорі застосовується від 40–60 синтаксичних відношень для кожної мови, яку досліджували.

Автори лінгвістичного процесору стверджують, що сьогодні майже повно реалізовані тільки морфологічні і синтаксичні моделі. Семантична модель не досягла достатньо повного рівня розвитку, але навіть при наявності повної семантичної моделі виникала б проблема застосування лінгвістичного процесору.

Лінгвістичний процесор повинен вміти настроювати формалізований зміст тексту на інтелектуальну систему (ІС), яка буде компонуватися з ним. Тобто, формалізований текст треба перетворити на

мову зрозумілу будь-якій системі. Створення процедури, яка б могла виконувати таке перетворення – проблема. Тому автори вирішили обмежити спілкування лінгвістичного процесору тільки з СУБД.

Для реляційних БД існує широко розповсюджена *формальна мова запитів* до баз даних SQL. Мова SQL універсальна, її можна застосовувати для запису даних різних ПДО. Вибір мови SQL дозволив настроїти лінгвістичний процесор тільки на один спосіб подавання даних і не загубити універсальність, широту розповсюдження. Мова запитів SQL є реляційною базою даних, тому її дані розміщені в таблицях. Для кожної таблиці вказані її атрибути, а для кожного атрибуту заданий перелік можливих значень атрибуту.

На семантичному етапі синтаксична структура речень перетворюється у їх семантичну структуру. Семантична структура – це дерево, у вузлах якого знаходяться предметні імена (слова мови у нормальній формі) або слова універсальної семантичної мови, розробленої для цього рівня (оператори, символи, назви таблиць із відомостями про ПДО, атрибути таблиць.). Дуги між вузлами подаються універсальними семантичними відношеннями (атрибут, аргумент, належить, кон'юнкція, диз'юнкція, тощо). Тобто семантична структура орієнтована на мову запитів та ПДО.

Автори лінгвістичного процесору висловлюють думку, що надалі з його допомогою можна буде розв'язати два класи задач: автоматичне поповнення баз даних за текстами природною мовою; планування текстів – створення текстів природною мовою за специфікацією.

4.3.2 Основні проблеми створення інтерфейсів природною мовою

Хоча існує багато систем, що в тій чи іншій мірі розуміють зміст текстів природною мовою, не вирішенні проблеми програмного розуміння природної мови залишаються. Розглянемо ці проблеми.

Проблема1. Кілька речень природною мовою можуть мати однаковий зміст, але подаватися різними словами і мати різні синтаксичні структури. Тобто, у базі знань системи виникає кілька подавань одного змісту. В результаті виникає проблема з яким змістовим подаванням працювати процедурам. Встановити змістову еквівалентність таких речень можна за допомогою процедур

умовиводу [10], але область дії таких процедур обмежена. Прикладом речень з однаковим змістом можуть бути речення.

1. Про студента Іванова у файлі відсутня інформація.
2. Файл не містить дані про студента Іванова.

Проблема2. Неоднозначне розуміння змісту речення за неповнотою речення. Людина може висловлюватись неповними реченнями. Такий тип висловлювання можна назвати «когнітивною економією». Зміст таких речень програма може поповнити і зрозуміти, якщо враховувати контекст речення (дискурс). Для поповнення речення можна застосувати базу знань.

Наприклад, зміст речення «Він зараз грає» можна зрозуміти тільки з контексту. Невідомо хто це «він». Можна грати в гру, грати роль, бути не щирим, грати на інструменті. Виникає проблема, як програма виявить значення слова «гра». Зміст вказаного речення невизначено, якщо речення одне. Виявлення змісту речення із контексту та бази знань складна задача.

Проблема3. Неоднозначне розуміння змісту речення із-за форми слів у реченні. Порівняйте речення «Буття визначає свідомість» та «Свідомість визначається буттям». З першого речення неможливо визначити, що визначається – буття чи свідомість. У другому реченні ясно, що свідомість визначається буттям. Для однозначного виявлення змісту речення застосовуються морфеми слів: суфікси, закінчення; частини мови: прийменники і т. п.

Проблема4. Застосування не проективних речень приводить до невірного програмного виявлення змісту речення. Речення називають не проективним, якщо одна іменна група розривається іншою іменною групою. Наприклад, у реченні «Хлопець вийшов у садок з лопатою» іменна група «хлопець з лопатою» розірвана іменною групою «вийшов у садок». Програма може виявити, що «садок з лопатою».

Проблема5. Вживання неологізмів невідомих системі. Неологізми – це нові слова. Існують загальномовні та індивідуальні неологізми. Загальномовні неологізми з'являються із розвитком суспільства. Наприклад, слово смартфон – неологізм. Індивідуальні неологізми створюються авторами. Наприклад, неологізми П. Тичини

«... Діло брать, щоб аж сміялось... Щоб життя в нас *виноградно і пшенично* наповнялось». Програмне виявлення значення неологізму на основі інших слів тексту складна задача.

Проблема6. Програмне розуміння природної мови значно ускладнює її метафоричність. Метафора – це розширення змісту слова за рахунок переносу в нього властивостей іншого слова. Наприклад, метафорами є «гострий розум», «океан думок». Слова метафори пов'язані асоціативно на основі образних представлень дійсності: гострі предмети швидко проникають в щось, океан величезний.

Метафори бувають двох типів: сталі вирази, які застосовують багато людей, і індивідуальні метафори, що активізуються в розмові залежно від ситуації. Метафори застосовують для передачі образного точного та стислого змісту думки. Одна метафора може створювати різні образи об'єкту залежно від контексту. Зміст такої метафори дуже складно виявити.

Будь-яке наукове знання метафорично. На метафору можна дивитися як на інструмент розвитку науки. У науці метафора – це представлення об'єктів однієї категорії в термінах іншої категорії. Метафори дозволяють побачити зв'язки між далекими за природою фрагментами наук. Це дозволяє застосувати поняття та методи однієї області науки у інших областях науки. За словами філософа Д. Девідсона [12] створення і розуміння метафор творчі процеси, які дуже мало підкорюються правилам. Часто важко навіть виявити саму метафору. Програмне виявлення метафори та її змісту – проблема.

4.3.3 Метод ключових слів до створення інтерфейсу діалогової системи

Метод ключових слів [13] базується на виявленні одного або кількох ключових слів речення користувача, які визначають реакцію програмної системи на звертання користувача. Реченнями можуть бути питання або команди. За ключовими словами програма знаходить відповідь або виконує дії відповідні команді.

Метод застосовують у програмах, в яких кількість ключових слів обмежена набором команд або невеликою темою бесіди. На методі ключових слів часто базуються програми типу питання-відповідь, які виставляються у Інтернеті для довільної бесіди. Така бесіда виглядає для людини достатньо природно.

Ми будемо розглядати застосування методу ключових слів на прикладі програми типу питання-відповідь. Для бесіди з користувачем оберемо двох персонажів відомої книги Всеволода Нестайка «Тореадори з Васюківки» Яву та Павлушу. Загальний підхід до створення моделей героїв взятий з книги А. П. Журавлева та Н. А. Павлюка [14].

Бесіда користувача з персонажами літературного твору завжди носить яскравий характер. Особливо це стосується персонажів дитячих творів. У програмі кожен герой книги подається своїм особистим описом, властивим тільки йому – *відношенням до змісту ключових слів і реакцією на них*.

У інтелектуальних системах для оцінювання відношення людини до того чи іншого змісту слова застосовується метод шкалування. Шкалою називають упорядкування об'єктів, подій, і т. п. за конкретною ознакою. Шкала може мати точку відліку і одиницю виміру відстані між поняттями. Такі шкали називають метричними шкалами. Наприклад, на метричній шкалі ваги можна розташовувати об'єкти за збільшенням ваги для порівняння їх ваги. Таку шкалу задають фактами. Наприклад, вага(«Микола», «87», «кг»).

Але нас будуть цікавити не метричні шкали. Не метричні шкали – порядкові застосовують при роботі з поняттями, що не мають числового значення. Причому не важливо, чи є у реальному світі числове значення у поняття. Наприклад, при аналізу тексту вага об'єктів у тексті може подаватися словами «дуже велика», «велика», «мала», така вага об'єктів буде подаватися порядковою шкалою.

Порядкові шкали не мають точки відліку та одиниці виміру, на них тільки впорядковують об'єкти, події, властивості. Порядкові шкали застосовують для оцінок різних типів: прекрасний – огидний, швидкий – повільний тощо. Порядкові шкали із словами антонімами на кінцях шкали називають опозиційними. Опозиційні шкали важливі для оцінювання відношення до об'єкту.

Нам потрібна шкала для оцінювання відношення героя або обох героїв до змісту кожного ключового слова. Будемо оцінювати кожне ключове слово за опозиційною шкалою «добре – погано».

Порядкові шкали можуть бути точковими або інтервальними. Ми будемо застосовувати точкові порядкові шкали, де кожна оцінка відношення до слова зображується на шкалі точкою. На рис. 4.2 показані оцінки слів для шкали «добре – погано». Оцінки

позначаються словом або словом разом із квантифікатором «дуже». Слова квантифікатори дозволяють розширити шкалу. Для можливості порівняння відношень до ключових слів кожній оцінці ставлять у відповідність число.

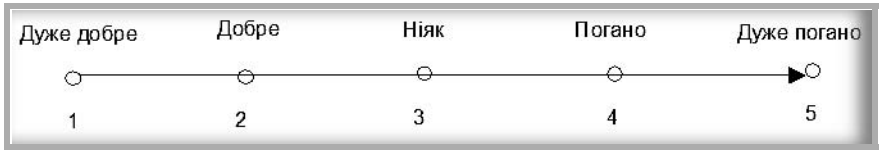


Рисунок 4.2 – Шкала «добре – погано»

Взагалі оцінки відношення до слів за шкалами виявляються у психологічному експерименті. Приймаючим участь у експерименті пропонується розмістити на групі шкал слова. Обробивши результати експерименту статистичними методами, одержують об'єктивне положення кожного слова на кожній шкалі групи. Одержані результати залежать від кількості тестуємих та їх належності певної категорії людей. Результати обробки завжди перевіряються практикою. В нашому випадку ми можемо тільки приписати кожному герою відношення до ключових слів числами – зарядами.

Опис програми. На початку діалогу користувачу пропонуються імена героїв, з якими він може розмовляти та ключові слова, які він може застосовувати у діалозі. Ключові слова застосовуються у питаннях користувача, а відповідь знаходиться за ключовим словом і відношенням героя до нього.

Перше звертання до певного героя повинно мати його ім'я. У наступних питаннях або зауваженнях до цього ж героя програма застосовує те ж саме ім'я, тому в звертанні до героя його можна не вказувати. Вказівка імені у звертанні до героя означає зміну героя, з яким спілкується користувач. Це дозволяє спілкуватися втрьох. Для закінчення роботи програми користувач застосовує у реченні слово «бувай» або слово з тією ж основою.

В процесі діалогу програма одержує питання користувача і відшукує у ньому ключове слово. Ключові слова можуть застосовуватися у різних словоформах. Знаходження ключових слів виконується за основами. У різних героїв може бути різне відношення до змісту одного слова і тому відповіді героїв будуть різного характеру.

Відповідь знаходиться за ім'ям герою, ключовим словом і відношенням героя до його змісту – зарядом. Наприклад, відношення Павлуші до цапа Жори, що з'їв його сорочку, погане (заряд 5). Ява байдужий до цапа Жори, йому тільки неприємно, що над ними будуть сміятися, якщо замість бика буде цап (заряд 4).

Щоб зробити бесіду більш природною будемо різноманітити відповіді героїв за класами об'єктів, які позначаються ключовими словами та номером відповіді для цього класу. Відповідь про річ не може бути застосована для відповіді про людину.

Обидва героя друзі, тому у них багато спільного. При відсутності ключового слова для конкретного героя програма шукає його за ознакою «людина». Деякі питання вимагають жорсткої прив'язки відповіді до конкретного героя. Наприклад, прізвище героя. Такі ключові слова відносяться тільки до одного героя і позначаються зарядом нуль і номером 100.

Якщо речення не містить ключових слів, то програма завжди знайде відповідь. Такі відповіді застосовують для будь-якого героя і мають заряд нуль. У групі відповідей з нульовим зарядом відповіді пронумеровані і тому вибір відповіді виконується за датчиком випадкових чисел.

Ключові слова необхідні для бесіди з конкретним героєм подаються окремою групою фактів з ім'ям герою. Ключові слова, що можуть застосовуватися обома героями подаються групою фактів з ім'ям «людина».

Кожен факт з ключовими словами для питання містить:

– ім'я героя або ознаку «людина»;
– ключове слово;
– особові або спільні оцінки (заряди) цих слів героями;
– клас поняття, що виражається словом;
– номер поняття у класі (0, 1, 2…);
– основу ключового слова.

Факти для відповіді містять:

– ім'я героя або ознаку «людина»;
– відповідь на питання;
– клас поняття, що виражається словом;
– заряд;
– номер поняття у класі (0, 1, 2…).

Нижче поданий текст програми.

```
Global Domains
заряд, номер = integer
діяч, слово, фраза, клас, основа = string

Global Facts
Nondeterm mod1(діяч, слово, заряд, номер, клас, основа)
Nondeterm mod2(діяч, фраза, клас, заряд, номер)

Single name (string)
Nondeterm діяч(string)
Single count (integer)

Predicates
Nondeterm метод()
Nondeterm бесіда(string)
Nondeterm знайти_відповідь(string, string)
Nondeterm ключ(string, string, string)
Nondeterm обрати_діяча(string)
Nondeterm змінити_діяча(string)
Nondeterm відп(string, string, integer, integer, string)
Nondeterm питання()
Nondeterm рахувати(string, string, integer, integer)
Nondeterm довідка()

Goal
довідка(), метод().

Clauses
name («»).
count (0).

довідка():-write(«Програма веде бесіду з двома хлопцями Явою та
Павлушею. Це герої книги Всеволода Нестайка *Тореадори з
Васюківки*.»), nl,
        write («Перше питання визначає співбесідника. Щоб закінчити
бесіду, останнє речення повинно мати слово бувай або похідне від
нього.»), nl,
```

write (« Ключові слова для бесіди: Павлуша, Ява, друг, прізвище, острів, село, дід, метро, льоха(свиня), вчителька, арифметика, бешкетник,»), nl,
write («кіно, тореадор, герой, корида, футбол, телевізор, бик, бугай, цап, сорочка, корова, килимок.»), nl, nl.

mod1(«Ява», «прізвище»,0,100, «ім'я», «прізвищ»).
mod1(«Ява», «острів Ява», 3,0, «острів», «остр»).
mod1(«Ява», «Павлуша», 1,0, «людина», «павлуш»).
mod1(«Ява», «друг»,1,6, «людина», «дру»).
mod1(«Ява», «герой», 1,1, «людина», «геро»).
mod1(«Ява», «корова Контрибуція», 2,0, «корова», «коров»).
mod1(«Ява», «корида»,1,1, «видовище», «корид»).
mod1(«Ява», «цап Жора»,4,0, «цап», «цап»).

mod1(«Павлуша», «прізвище»,0,100, «ім'я», «прізвищ»).
mod1(«Павлуша», «Ява»,1,1, «людина», «яв»).
mod1(«Павлуша», «тореадор»,1,3, «людина», «тореадор»).
mod1(«Павлуша», «дід»,1,4, «людина», «дід»).
mod1(«Павлуша», «друг»,1,6, «людина», «дру»).
mod1(«Павлуша», «корова Манька»,2,0, «корова», «коров»).
mod1(«Павлуша», «килимок»,2,1, «річ», «килим»).
mod1(«Павлуша», «сорочка»,2,1, «річ», «сороч»).
mod1(«Павлуша», «цап Жора»,5,0, «цап», «цап»).
mod1(«Павлуша», «корида»,1,1, «видовище», «корид»).

mod1(«людина», «кіно»,1,0, «видовище», «кіно»).
mod1(«людина», «тореадор»,1,2 , «людина», «тореадор»).
mod1(«людина», «телевізор»,1,2, «пристрій», «телевізор»).
mod1(«людина», «метро»,1,3, «транспорт», «метро»).
mod1(«людина», «футбол»,1,4, «гра», «футбол»).
mod1(«людина», «бешкетник»,1,5, «людина», «бешкет»).
mod1(«людина», «село»,2,2, «поселення», «сел»).
mod1(«людина», «бик»,3,0, «тварина», «бик»).
mod1(«людина», «бугай Петька»,5,0, «бугай», «буга»).
mod1(«людина», «льоха Манюня»,5,1, «льоха», «льох»).
mod1(«людина», «арифметика»,4,2, «наука», «арифмети»).
mod1(«людина», «вчителька»,5,3, «людина», «вчительк»).

mod2(«бувай», «До зустрічі!», «»,0,0).
mod2(«бувай», «Добре, мені теж треба йти, бувайте!», «»,0,1).
mod2(«бувай», «Так, годі вже балакати», «»,0,2).

mod2(«Ява», «Рень», «ім'я»,0,100).
mod2(«Ява», «Павлуша друзяка!», «людина»,1,0).
mod2(«Ява», «Павлуша Завгородній», «людина»,1,6).
mod2(«Ява», «Я заздрю йому», «людина»,1,1).
mod2(«Ява», «Більше за все на світі хочеться їм стати!», «людина»,1,2).
mod2(«Ява», «Це було б здорово», «пристрій»,1,2).
mod2(«Ява», «Благородна ідея - провести під свинарником метро.», «транспорт»,1,3).
mod2(«Ява», «Ми завжди ходимо з Павлушею. Класна гра», «гра»,1,4).
mod2(«Ява», «Найкраща розвага», «видовище»,1,0).
mod2(«Ява», «Мила, розумна та терпелива Контрибуція», «корова»,2,0).
mod2(«Ява», «Дід битиметься», «людина»,1,5).
mod2(«Ява», «Прізвище не походить від назви острова», «острів»,3,0).
mod2(«Ява», «Це таке видовище!», «видовище»,1,1).
mod2(«Ява», «Та це таке сміховисько!», «цап»,4,0).
mod2(«Ява», «Ні, нехай з бугаєм наші вороги б'ються», «бугай»,5,0).

mod2(«Павлуша», «Завгородній», «ім'я»,0,100).
mod2(«Павлуша», «Ваня Рень», «людина»,1,6).
mod2(«Павлуша», «Ми будемо знамениті тореадори», «видовище»,1,1).
mod2(«Павлуша», «Я був би радий», «пристрій»,1,2).
mod2(«Павлуша», «Ой, як хочеться!», «людина»,1,2).
mod2(«Павлуша», «Це мало бути сюрпризом», «транспорт», 1,3).
mod2(«Павлуша», «Ми з Явою часто граємо», «гра», 1,4).
mod2(«Павлуша», «Коли діда не видно», «людина»,1,5).
mod2(«Павлуша», «Ява найкращий друг», «людина»,1,1).
mod2(«Павлуша», «Він ловкач та герой!», «людина»,1,3).

```
mod2(«Павлуша», «Мисливець завзятий, але лається!»,
        «людина»,1,4).
mod2(«Павлуша», «Манька мала ще та без рогу»,
        «корова»,2,0).
mod2(«Павлуша», «Килимок червоний, махати перед биком»,
        «річ»,2,1).
mod2(«Павлуша», «Ти що, не читав Тореадори з Васюківки?»,
        «поселення»,2,2).
mod2(«Павлуша», «Остогидів він. Терпіти не можу його»,
        «цап»,5,0).
mod2(«Павлуша», «У-у, скотиняка! Щоб ти…!», «льоха»,5,1).
mod2(«Павлуша», «Ох, і важка наука!», «наука»,4,2).

mod2(«людина», «Ходимо кожну суботу», «видовище», 1,0).
mod2(«людина», «Васюківки», «поселення»,2,2).
mod2(«людина», «Так видовище, що ж зробиш»,
        «тварина»,3,0).
mod2(«людина», «Страхіття», «бугай»,5,0).
mod2(«людина», «Огидна тварина. Зламала нам метро!»,
        «льоха»,5,1).
mod2(«людина», «Та я не спеціально пропустив екзамен!»,
        «наука»,4,2).
mod2(«людина», «Дуже сувора», «людина»,5,3).

mod2(«людина», «Не знаю, треба подумати», «»,0,0).
mod2(«людина», «Я не хочу про це розмовляти», «»,0,1).
mod2(«людина», «Причепився, як реп’ях до хвоста собаки»,
«»,0,2).
mod2(«людина», «А що це?», «»,0,3).
mod2(«людина», «Не розумію», «»,0,4).
mod2(«людина», «То й що?», «»,0,5).
mod2(«людина», «Ну ти й хлопець з фантазією!», «»,0,6).
mod2(«людина», «І нащо таке питання?», «»,0,7).
mod2(«людина», «Дарма питаєш», «»,0,8).
mod2(«людина», «Хто ж тобі скаже?», «»,0,9).
mod2(«людина», «Ти хочеш, щоб з нас сміялися?»,
        «»,0,10).
```



```
метод():-mod1(C1,_,_,_, «людина»,_), C1<>«людина»,
        mod1(C2,_,_,_, «людина»,_),
        C2<> «людина», C1<>C2, !,
        write (C1, « та», C2,
        «бажають поговорити. Зверніться до когось.»),
        nl, assert(діяч(C1)),
        assert (діяч(C2)), питання().

питання():-readln(Питання), обрати_діяча(Питання),
        бесіда(Питання).

обрати_діяча(S):- діяч(C1), searchstring (S, C1, _),
        assert (name (C1)), !;

        write («Перше питання повинно мати ім’я співрозмовника»), nl,
питання ().

змінити_діяча(Питання):- діяч(W), name(W1), W<>W1,
        searchstring(Питання, W,_),
        assert (name (W));
        True.

бесіда(Питання):-змінити_діяча(Питання),
        знайти_відповідь(Питання, Відповідь),
        write (Відповідь), nl, readln (Питання2),
        !, бесіда(Питання2).

знайти_відповідь(S, S1):-W=«бувай», upper_lower(S, S11),
        searchstring(S11, W,_),
        рахувати(W, «», 0, K),
        K<>0, random(K, M1),
        mod2(W, S2, «»,0, M1),
        write (S2), nl, exit (), !;

        name (N), mod1 (N, W, Z, 100, Kl, C),
        ключ(S, W, C), відп(N, Kl, Z, 100, S1);

        name (N), mod1 (N, W, Z, K2, Kl, C),
        ключ(S, W, C), відп(N, Kl, Z, K2, S1),!;

        name(N), mod1(«людина», W, Z, K2, Kl, C), ключ(S, W, C),
        відп(N, Kl, Z, K2, S1),!;
```

```
        рахувати(«людина», «»,0, K),
        random (K, M1),
        mod2(«людина», S1, «»,0, M1),!.

рахувати(N,_,0, K2):-mod2(N,_, «»,0, _),
        count (K1), K2=K1+1,
        assert (count (K2)), fail.

рахувати(_,_,_, K2):-count(K2), assert(count(0)).

відп(N, Kl, Z, K2, S1):- mod2(N, S1, Kl, Z, K2), !;
        mod2(«людина», S1, Kl, Z, K2),!.

ключ(S, W1, C):- fronttoken(S, W,_), upper_lower(C, C1),
        upper_lower(W, W2), upper_lower(W1, W11),
        concat(C1,_, W11), concat(C1,_, W2),!;

        fronttoken(S,_, R),!, ключ(R, W1, C).
```

Процедури довідка та метод починають роботу програми. Процедура метод вибирає з фактів імена героїв за класом людина, що робить програму незалежну від набору фактів, які застосовуються для бесіди. Імена героїв завжди фіксуються у першому аргументі фактів. Факти можна завантажувати, додавши предикат consult, але ми занесли їх у програму для більшої наочності.

Процедура питання вводить перше питання, викликає послідовно процедури обрати діяча та бесіда.

Процедура обрати діяча шукає у першому питанні до героя його ім'я і робить героя активним.

Процедура бесіда може змінювати активного героя, якщо це потрібно, і відшукує відповідь на питання. Після чого вводить наступне питання і діалог продовжується.

Процедура знайти_відповідь здійснює перевірку на кінець діалогу. При наявності слова «бувай» рахується кількість варіантів прощання і обирається за датчиком випадкових чисел одне з них.

Якщо діалог не закінчується, то перебираються варіанти відповіді у наступному порядку:

– одна фіксована відповідь за ім'ям;
– відповідь за ім'ям, зарядом та класом;
– спільна відповідь за зарядом та класом;
– відповідь без ключового слова.

Процедура відп шукає відповідь за ім'ям, зарядом та класом. Якщо за ім'ям відповідь не знайдена, то пошук продовжується за класом людина. Процедура ключ знаходить за основою ключове слово у реченні.

4.4 Генеративні граматики

Сьогодні проблемами природних мов займаються фахівці різних напрямів: психологи і лінгвісти, математики і спеціалісти штучного інтелекту, спеціалісти з інформатики та програмного забезпечення. Інформаційний пошук у Інтернеті, програмне розуміння текстів природною мовою, спілкування людини з діалоговою системою, програмний переклад текстів – це неповний перелік важливих задач пов'язаних з природними мовами.

В 1957 році американський лінгвіст Н. Хомський випустив свою книгу «Синтаксичні структури» [15], яка повністю змінила погляди на вивчення мов. Хомський запропонував замість прийнятого словесного опису правил синтаксису природних мов розглядати опис синтаксичних правил на базі формального математичного апарату.

У книзі Хомський висловив думку, що при лінгвістичному аналізі речень природної мови основною проблемою є відокремлення вірних речень, що побудовані за правилами граматик, від невірно побудованих речень. Виявлення таких механізмів, що *породжували б тільки вірні речення* природних мов, зняло б цю проблему. Вірність речення перевірялась би через можливість його генерації цими механізмами.

Кінцевою метою синтаксичних досліджень Хомський вважав виявлення фундаментальних властивостей природних граматик, які б дозволяли побудувати єдину лінгвістичну теорію. Застосовуючи таку теорію можна було б вивчати конкретні граматики без звертання до цих мов.

Ідеї Хомського лягли в основу нового напряму формальної лінгвістики – генеративної (породжуючої) лінгвістики. Базовим типом моделей цього напряму стали *трансформаційні породжуючі моделі*.

Сьогодні генеративна лінгвістика займається виявленням *загальних механізмів розуміння мови* у людини та механізмів

створення речень будь-якою природною мовою. Такі механізми треба відрізняти від механізмів реального застосування мови людиною залежно від умов (уваги, цікавості, застосування неповних речень).

Взагалі Н. Хомський виділив чотири типи моделей генеративних граматик, що відрізняються складністю виду граматичних правил та мови, породженою механізмами цього типу. Під мовою розуміють всю множину речень, яку можна одержати за цими правилами.

Для знайомства з типами граматик нам знадобиться нова термінологія. Назвемо термінальними символами слова, з яких складаються речення мови: ранок, йти, школа, абстрактний і т. п.. Нетермінальними символами називають граматичні категорії (іменник, присудок, речення і т. п.), їх позначають великими буквами.

Правила генеративних граматик визначають можливу заміну одного нетермінального символу на групу інших нетермінальних символів або один термінальний символ. Правило має ліву і праву частини. У лівій частині розташовується нетермінальний символ. У правій частині розташовують символи, які підставляються замість нетермінального символу з лівої частини. Процес заміщення повторюється поки всі нетермінальні символи не будуть замінені на термінальні. Термінальні символи не можна замінити на інші символи.

Нижче подані типи граматик генеративної лінгвістики в порядку збільшування вимог до граматичних правил. Чим більше обмежень на *вид правил* граматики, тим простіші речення мови.

1. Граматику першого типу так і називають генеративною. Генеративні граматики найскладніші, в них майже немає обмежень на вид правил граматики, що породжують речення.

2. У правилах контекстна-залежної граматики замість кожного нетермінального символу з лівої частини правила підставляють групу нетермінальних символів або один термінальний символ з правої частини правила *залежно від контексту символу* з *лівої частини правила*. Такі правила називають правилами підстановки.

Правило підстановки в загальному вигляді записується:

$$A \to \frac{Z}{X} - Y$$

де A нетермінальний символ з лівої частини; X та Y дві послідовності символів, які можуть бути і порожніми, і визначають контекст A; Z

послідовність символів, що підставляється замість A у тому ж контексті X та Y. Тобто, виконується підстановка:

$$XAY \to XZY$$

Прикладом правила підстановки, може бути «правило твору»: Нехай є два речення: «Ці звуки музики були прекрасні.» і «Ці звуки пісні були прекрасні». Слова «музики» та «пісні» мають однаковий контекст, тому за правилом можна створити нове речення: «Ці звуки пісні та музики були прекрасні». У прикладі X «Ці звуки» Y «були прекрасні». В першому реченні A «музики» у другому реченні A «пісні». Правило твору дозволяє замінити обидва A на їх композицію Z. Природні мови по більшості контекстна-залежні.

3. У правилах підстановки контекстна-вільних граматик заміна нетермінального символу з лівої частини правила на символи з правої частини не залежать від контексту. Тому контекстна-вільні граматики лише приблизно описують природні мови. Механізм контекстна-вільної граматики працює як стековий автомат і його застосовують для генерації команд або операторів у програмуванні. У стекових автоматах є сильна залежність наступного стану автомату від попереднього. Контекстно-вільні мови відносять до алгебраїчних мов, їх вивчають у комп'ютерній лінгвістиці та інформатиці.

4. Регулярні граматики – породжують найпростіші граматично вірні речення. Механізмами регулярної граматики є породжуючі кінцеві автомати.

Множини правил породжуючих граматик вкладаються одна в одну наступним чином:

$$1 \supset 2 \supset 3 \supset 4$$

4.4.1 Основні поняття породжуючих граматик

Правила генеративних граматик формально визначаються множиною:

$$\text{П. Г.} = \{\sum, N, P, S\},$$

де $\sum$ – алфавіт термінальних символів;

N – алфавіт нетермінальних символів;

P – набір синтаксичних правил підстановки;

S – початковий нетермінальний символ, з якого завжди починається вивід.

Для подавання моделі граматики – системи правил, що створюють механізм породження речень, застосовують метамову. Вибір метамови залежить від типу граматики. Опис правил для контекстна – залежної граматики застосовує назви категорій змістових граматик природних мов.

Правила контекстна-залежної граматики зручно записувати правилами Прологу.

група_іменника():- прикметник(), іменник().

Для запису правил контекстна – вільних граматик застосовують метамову Бекуса-Наура (БНФ).

Наприклад, поняття предикату можна визначити:

<Предикат >::=< Ім'я > | < Ім'я >(<аргумент>[,<аргумент>])

Твердження читається: «Предикат це ім'я, або ім'я з одним аргументом, або кількома аргументами.

У регулярних граматиках правила записують у таких формах:

$A \rightarrow$ с (тип 1),

$A \rightarrow$ сМ (тип 2),

$A \rightarrow$ ε, (тип 3),

де нетермінальні символи позначають великими буквами, а термінальні символи малими буквами. Порожній рядок позначається ε. Правило типу 1 замінює нетермінальний символ на термінальний. Правило типу 2 замінює нетермінальний символ на ланцюг з термінального і нетермінального символу (права граматика). Правило типу 3 замінює нетермінальний символ на порожній рядок. Можлива граматика, у якої читання правила типу 2 виконується справа наліво (ліва граматика). За правилом спочатку замінюється правий нетермінальний символ:

$A \rightarrow$ Мс (тип 2).

Процес породження речення у генеративних граматиках завжди починається з нетермінального початкового символу *S*.

Всі речення, які можна одержати за системою правил називають мовою породжуючої граматики. Речення, що вводиться у діалогову систему, є вірним, якщо його можна одержати за цією граматикою.

4.4.2 Регулярні граматики і кінцеві автомати

Природна мова живий організм, що відображує навколишній світ. Це означає, що в мові весь час з'являються нові слова (неологізми), а старі слова перестають вживатися (архаїзми). Звідси нескінченна кількість слів мови та речень, тому подати всі слова мови списком неможливо. Н. Хомський запропонував подавати слова не списком, а механізмом, що породжує з слів кінцеві граматично вірні речення. Такий механізм називають кінцевим автоматом.

Автомат має кінцеву кількість станів і в кожний момент може приймати один з них. Коли автомат переходить з одного стану у інший він друкує слово. При вході в початковий стан автомат виводить перше слово. Для кожного випадку роботи автомату треба мати ознаку кінця. Для переходу з початкового стану у кінцевий автомат приймає ряд проміжних станів і виводить кінцеву кількість слів. Таким чином друкується речення, що має кінцеву кількість слів. Кожне речення, що породжується автоматом, назвемо вірним реченням, а всі речення, що можуть породжуватися автоматом, мовою автомату. На рис. 4.3 показана діаграма станів простого кінцевого автомату. Стан автомату подається вузлом діаграми. Кожне речення може закінчуватися тільки у вузлі з подвійним колом.

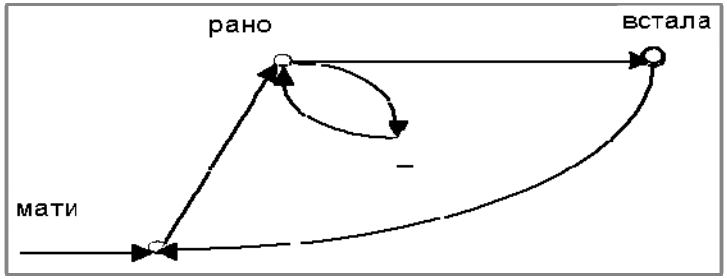


Рисунок 4.3 – Діаграма станів кінцевого автомату

Прикладами речень, що виводить автомат можуть бути наступні:

1. мати рано встала
2. мати рано-рано встала
3. мати рано встала мати рано встала
4. мати рано-рано встала мати рано встала, і. т. д.

Автомат з кінцевою кількістю станів описується марковським процесом з кінцевою кількістю станів системи S_1, S_2, …S_N, де N ціле число. Процес задається матрицею ймовірності переходів $P(t1, t2)$, де кожний елемент матриці $p_{i, j}(t1, t2)$ подається умовною ймовірністю, що система в момент $t2$ буде знаходитися у стані S_j, при умові, що в момент $t1$, система знаходилася у стані S_i.

Ясно, що з точки зору природних мов, далеко не всі речення породженні кінцевим автоматом є вірними з точки зору граматики природної мови. Виявилось, і це доказав сам Хомський, що формальну теорію кінцевих автоматів неможна застосувати до створення всіх граматично-вірних речень природної мови. Це пов'язано з кількома причинами. Самою явною причиною є та, що кінцевий автомат формує речення послідовно додаючи по слову, в той час, як структура речень природною мовою ієрархічна і подається деревом.

В той же час кінцевими автоматами можна вирішувати інші проблеми природної мови, якщо вони непов'язані із граматичними структурами. Останнім часом кінцеві автомати застосовують у формальній теорії інформаційного пошуку [16].

Створюється мовна ймовірна *модель правдоподібності запиту* для кожного документу з колекції документів. Модель документу подається ймовірним кінцевим автоматом, в якому для кожного стану вказана ймовірність його генерації. Автомат послідовно видає терміни документу і визначає ймовірність породження кожної послідовності слів. Всі послідовності термінів складають модель документу. Для такої моделі неважливий порядок термінів у послідовностях. Документи ранжуються за ймовірностями того, що запит може бути виведений з моделі. Якщо вивід можливий, то кажуть, що модель документа релевантна (відповідає) запиту. Такий метод пошуку дає хороші результати.

Механізм кінцевих автоматів можна застосовувати з різними цілями. Розглянемо реалізацію кінцевого автомату на прикладі, в якому контролюється арифметичний вираз на вірність запису.

У програмі факти перехід контролюють послідовність лексем у виразі. Процедура аналіз контролює вид лексем у виразі.


```
Predicates
Nondeterm перехід (string, string)
Nondeterm вираз (string)
Nondeterm тип_лексеми (string, string)
Nondeterm аналіз_лексеми (string, string)
Nondeterm роби ()

Global Facts
Single лексема (string)

Clauses
лексема(«»).
перехід(«Число», «Буква»).
перехід(«Число», «Операція»).
перехід(«Число», «Закрита дужка»).
перехід(«Буква», «Закрита дужка»).
перехід(«Операція», «Число»).
перехід(«Операція», «Буква»).
перехід(«Буква», «Операція»).
перехід(«Операція», «Відкрита дужка»).
перехід(«Відкрита дужка», «Число»).
перехід(«Відкрита дужка», «Операція»).
перехід(«Відкрита дужка», «Буква»).
перехід(«Закрита дужка», «Операція»).
перехід(«Закрита дужка», «»).
перехід(«Число», «»).
перехід(«Буква», «»).

Тип_Лексеми(«+», «Операція»).
Тип_Лексеми(«–», «Операція»).
Тип_Лексеми(«*», «Операція»).
Тип_Лексеми(«/», «Операція»).
Тип_Лексеми(«(«, «Відкрита дужка»).
Тип_Лексеми(«)», «Закрита дужка»).

вираз(«»):-write(«Вираз вірний»), nl.

вираз(А):-fronttoken(А, Лексема2, Залишок),
        аналіз_лексеми(Лексема2, Тип_Лексеми2),
```

```
        лексема(Тип_Лексеми),
        перехід(Тип_Лексеми, Тип_Лексеми2),
        assert (лексема (Тип_Лексеми2)), вираз(Залишок);

        fronttoken(A, Лексема, _),
        write («У виразі невірно розміщена лексема»,
        Лексема), exit.

аналіз_лексеми(Лексема, Тип_Лексеми):-
        str_int (Лексема, _), Тип_Лексеми = «Число»,!;
        str_char (Лексема, Лексема_С),
        Лексема_С>= ‘a’, Лексема_С<= ‘z’, Тип_Лексеми = «Буква»,!;
тип_лексеми (Лексема, Тип_лексеми),!;

nl, write(«У виразі невірна лексема», Лексема), exit().

Роби():-Write («Введіть арифметичний вираз»), nl,
        readln(A), fronttoken(A, Лексема, B),
         not (Лексема = «*»), not (Лексема = «/»),
        аналіз_лексеми(Лексема, Тип_Лексеми),
        assert(лексема(Тип_Лексеми)), вираз(B),!;
        write («Невірно розміщена перша лексема «), nl.
Goal
Роби().
```

4.4.3 Контекстна-залежні граматики

Система правил контекстна-залежної генеративної граматики має забезпечити вивід необмеженої кількості речень природної мови за необмеженою кількістю структур.

Вона породжує всі речення мови шляхом поступової трансформації простих речень, одержаних за базовим набором структур, тому граматику називають трансформаційною.

Трансформаційна граматика містить систему правил таких типів:

- синтаксичні правила;
- семантичні правила;
- фонологічні правила.

Відповідно цьому, трансформаційну граматику можна розглядати як пристрій, який згідно типам її правил розподіляється на три компонента: синтаксичний, семантичний, фонологічний.

Природна мова складна система, але цю складність можливо зменшити за рахунок введення *лінгвістичних рівнів* роботи з мовою. Для цього синтаксичний компонент розподіляється на два компонента: базовий і трансформаційний.

Правила базового синтаксичного компоненту породжують прості речення та відповідні їм базові синтаксичні структури.

Правила, які є основою базової синтаксичної структури, називають деривацією. Деривація визначає, яким повинно бути розташування слів у простих реченнях. Базові синтаксичні структури речень називають базовими С-показниками. За С-показниками будують речення. З базових С-показників складаються більш складні структури. Найпростіші речення мають структуру, що складається з одного С-показника.

За трансформаційними правилами базові синтаксичні структури перетворюються у поверхневі структури і породжуються речення з відповідними поверхневими структурами.

Базові структури ближче до семантики, вони базуються на поняттях дія, суб’єкт дії, об’єкт дії. Поверхневі структури ближче до синтаксису, вони вказують на синтаксичні зв’язки між словами.

Базова та поверхнева синтаксичні структури речень обидві подаються деревом, але вміст першого дерева більше наближений до змісту, а другого показує синтаксичні зв’язки між словами.

У семантичному компоненті лексичні правила виводять з базових структур речень певне семантичне представлення, ставлять у відповідність кожній базовій структурі певний зміст. Зміст і речення не можна ототожнювати, вони подаються різними засобами.

У фонологічному компоненті [17] фонологічні правила ставлять у відповідність поверхневим структурам речень фонетичне представлення, певне звучання.

Є припущення, що система базових правил однакова для всіх природних мов. До базових речень відносять прості оповідальні недвозначні активні речення. Тобто, речення, в яких на діяча точно вказано і зміст таких речень недвозначний. Наприклад речення «Діти грають» є базовим реченням. У реченні «Виклик абонента» неясно, «абонент» суб’єкт чи об’єкт дії. Речення такого типу не відносяться до базових. Такі речення можна одержати за трансформаційними правилами, трансформуючи базове речення «Абонент викликається».

Лексикон

Термінальні та нетермінальні символи, які застосовуються для породження речень, зберігаються у словнику, який називають Лексикон. Лексикон містить граматичні нетермінальні символи типу: *S* – речення, *VP* – група дієслова, *NP* – група іменника, *N* – іменник і т. п., а також термінальні символи: слова типу: «дівчина», «працювати», «істотність», «абстрактність» тощо. Термінальні символи розподіляють на лексичні та граматичні. Наприклад, слово «дівчина» відноситься до лексичних символів. До граматичних термінальних символів можна віднести слова «істотність», «абстрактність».

Кожний компонент лексикону містить слово (лексичний термінальний символ), його граматичні нетермінальні символи, граматичні термінальні символи. Знак «+» вказує на наявність ознаки. Подамо приклад такого компоненту фактом на Пролозі.

лексикон(«дівчинка»,[«+N», «+Істотний», «+Людяний»]).

Надалі для спрощення замість нетермінальних символів ми будемо застосовувати у словнику тільки термінальні символи.

4.4.4 Створення базової синтаксичної структури речень

Розглянемо на прикладі програмне створення базової синтаксичної структури речень – С-показника та відповідних речень.

Для створення базової синтаксичної структури речень треба мати словник – лексикон та правила наступних типів.

1. Контекстна-вільні правила підстановки категоріальних символів (деривації).
2. Контекстна-вільні правила підстановки синтаксичних ознак та операцій над комплексними символами.
3. Контекстна-залежні правила суворої субкатегоризації та правила селекції.

СТВОРЕННЯ ЛЕКСИКОНУ

Граматичні нетермінальні символи представляють собою граматичні категорії, а граматичні термінальні символи – лексичні ознаки. Будемо надалі їх так називати.

Щоб зрозуміти вміст компонентів лексикона розглянемо речення: «Дівчинка хоче взяти яблуко».

Компонент лексикона для слова «яблуко» буде містить граматичну категорію +N (іменник) та наступні лексичні ознаки:

лексикон(«яблуко»,[«+N», «+Загальність», «+Зліченність»,«–Істотність»]).

Знак «+» вказує на наявність ознаки. Знак «–» вказує на протилежний зміст ознаки. Для слова «яблуко» знак «–» означає «неістотний». Лексичні ознаки «загальність» і «зліченність» позначають загальне слово, а не власну назву, та можливість порахувати об’єкти. Лексичні ознаки слів застосовують у традиційних граматиках.

У компоненті категорія слова та його лексичні ознаки подаються у квадратних дужках, їх називають *комплексним символом*. Така назва пов’язана з тим, що у формальній граматиці словник містить символи і дії виконуються над символами.

Лексичні ознаки іменників можна класифікувати за схемою, яка подана на рис. 4.4 [18].

Кожний ланцюг класифікаційної схеми задає множину лексичних ознак, які відносять слова природної мови категорії іменників, до певного класу. Ознаки слів необхідні для побудови вірних речень.

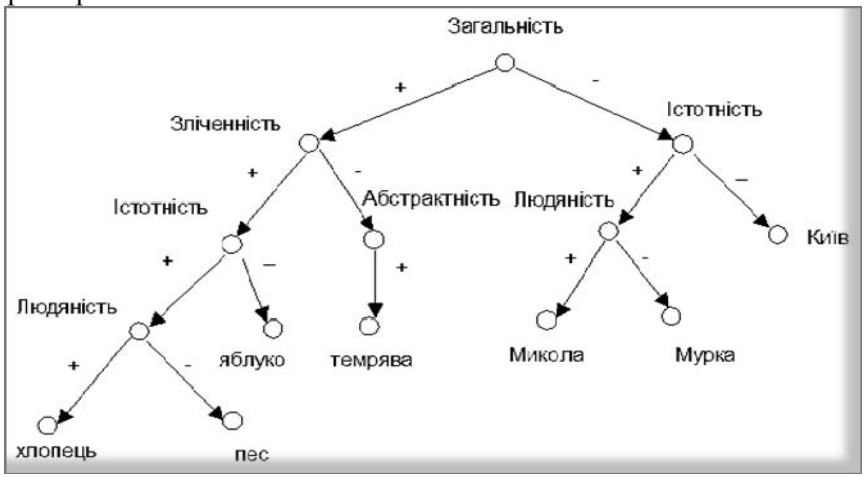


Рисунок 4.4 – Схема класифікації лексичних ознак іменників

СТВОРЕННЯ ТА РОБОТА ПРАВИЛ

Правила підстановки категоріальних символів називають *деривацією*. Вони задають *загальні синтаксичні структури* речень що породжуються граматикою. В основі цих правил лежать правила традиційні граматики. Природно виглядає така система правил [18].

$$
\begin{array}{l}
S \rightarrow NP, Aux, VP \\
NP \rightarrow N \\
VP \rightarrow V, NP \\
Aux \rightarrow M \\
N \rightarrow \text{«дівчинка»} \\
N \rightarrow \text{«яблуко»} \\
M \rightarrow \text{«хоче»} \\
V \rightarrow \text{«взяти»}
\end{array} \qquad (4.1)
$$

де *S* – речення, *NP* – група іменника, *VP* – група дієслова, Aux – допоміжне дієслово, Aux+*VP* – предикатна складова, *N* – іменник, *M* – модальне дієслово, *V* – дієслово.

До допоміжних дієслів можна віднести модальні дієслова, що вказують на відношення людини до дії (може, хоче), або будь-яке інше дієслово, що уточнює основний зміст наступного дієслова. При цьому допоміжне дієслово може вказувати на час (буду їхати) або іншу ознаку. Порівняйте два речення: «Я люблю молоко» і «Я люблю пити молоко». У другому реченні дієслово «люблю» є допоміжним, воно вказує на відношення до дії, хоча і не являється модальним дієсловом.

Множина правил підстановки категоріальних символів задає послідовність заміщення кожного категоріального символу з лівої частини правила на категоріальні символи з правої частини правила. Заміщення категорії з лівої частини правила не залежить від контексту цієї категорії, тому правила називають контекстно-вільними. В результаті за ними можна одержати як вірні речення, так і невірні. Наприклад, за ними можна одержати речення:

«Дівчинка хоче взяти яблуко» (4.2)

«Яблуко хоче взяти дівчинка» (4.3).

У реченні (4.3) не враховується, що допоміжне дієслово «хоче» не може з'явитися після іменника «яблуко». Для цього іменник

повинен мати лексичну характеристику «+Істотність», а правила підстановки категоріальних символів контекстно-вільні.

Взагалі, щоб програма не створювала речення типу (4.3), треба визначити, які *функції виконують слова у реченні* відносно дії або відносно всього речення.

Розглянемо функції слів у реченні (4.2). Кожне слово речення (4.2) виконує в ньому певну функцію. Слово «Дівчинка» є суб'єктом дії «взяти». Слово «яблуко» є об'єктом дії «взяти». Слово «хоче» вказує на бажання дії. Але ці слова у іншому реченні можуть виконувати інші функції. Наприклад, у реченні «Яблуко може впасти» слово «яблуко» виконує функцію суб'єкту. Тому у лексиконі не можна зберігати функції слів. В той же час поняття функцій слів у реченні відносяться до семантичних понять, а ми будуємо синтаксичну структуру. Тому необхідно знайти, яким способом можна виразити семантичні поняття «суб'єкт дії», «об'єкт дії» тощо, через лексичні ознаки, які відносяться до синтаксичних категорій.

Для розуміння, як враховуються лексичні категорії треба розглянути весь механізм роботи правил. Правила реалізуються на основі методу наданого в [18].

1. Будується базова структура майбутніх речень за правилами підстановки граматичних категорій (4.1). На цьому етапі компоненти лексикону не застосовуються.

Під час побудови базової структури для кожної категорії «іменник» виявляються всі можливі варіанти її лексичних ознак за контекстна - вільними правилами типу (4.4) див. рис. 4.4:

$$
\begin{array}{l}
\text{Ё} \quad \begin{array}{l}
N \rightarrow [\text{«+N»}, \text{«+Загальність»}] \\
[\text{«+Загальність»}] \rightarrow [\text{«±Зліченність»}] \\
[\text{«+Зліченність»}] \rightarrow [\text{«±Істотність»}] \\
[\text{«–Зліченність»}] \rightarrow [\text{«+Абстрактність»}] \\
[\text{«+Істотність»}] \rightarrow [\text{«±Людяність»}] \\
[\text{«+Зліченність»}] \rightarrow [\text{«±Абстрактність»}]
\end{array}
\end{array} \qquad (4.4)
$$

Тим самим одна базова структура породжує множину структур – *С-показників* майбутніх речень, які будуть відрізнятися наборами лексичних ознак. Правила (4.4) заміщують категоріальні символи N на комплексні символи з лексичних ознак та породжують претермінальні ланцюги лексичних ознак.

2. Одночасно з побудовою С-показників речень будуються претермінальні ланцюги, які застосовують для побудови термінальних ланцюгів – речень. На цьому етапі у претермінальному ланцюгу відсутні слова та лексичні ознаки змістового дієслова та модального дієслова. Претермінальний ланцюг для речень типу: «Дівчинка хоче взяти яблуко» будемо подавати фактом з аргументами:
Претермінальний_ланцюг

([«+N», «+Загальність», «+Зліченність»,«+Істотність», «+Людяність»], «M», «V»,
[«+N», «+Загальність», «+Зліченність», «–Істотність»]), (4.5)

де M і V символи, які будуть надалі заміщуватися на комплексні символи.

3. Лексичні ознаки дієслова залежать від того, які групи граматичних категорій застосовуються у контексті дієслова, а також від лексичних ознак граматичних категорій з контексту.

Правила, які визначають контекст дієслова за граматичними категоріями називають правилами *суворої субкатегоризації*. Ми розглядаємо приклад, де групи іменників містять тільки по одному іменнику. Для нашого прикладу правило (4.6) відноситься до правил суворої субкатегоризації, воно визначає перехідність дієслова. Дієслово має лексичну ознаку «перехідність», якщо за ним розташована група іменника «–NP». У правилах суворої конкретизації знак «-» означає ознаку «перехідність».

$$V \rightarrow [«+V», «–NP»] \qquad (4.6)$$

У контексті дієслова можливі різні групи граматичних категорій. Наприклад, група іменника може складатися з: прийменника іменника; прикметника іменника і т. п. [16].

Лексичні ознаки дієслова визначаються за контекстна-залежними селекційними правилами. Селекційні правила містять умови, при яких дієслово одержує свої ознаки. Селекційні правила подані в (4.7).

V → [«+Істотність»], «Aux–»;
[«–Істотність»], «Aux–»; (4.7)

–[«+Істотність»];
–[«–Істотність»].

Правила (4.7) працюють для дієслів, які успадковують ознаки від іменників з свого контексту. Перші два правила вказують: якщо перед допоміжним дієсловом та основним дієсловом стоїть іменник з ознакою «±Істотність», то іменник виконує роль *суб'єкту при дії*. В цьому випадку при створенні речень з слів лексикону можуть бути обрані тільки дієслова, які мають характеристики «±Істотність».

Останні два правила вказують на *об'єкт при дії*. Правила вказують, що після дієслова може стояти тільки іменник з ознакою «±Істотність». Для скорочення запису селекційних правил в них вказуються тільки умови вибору дієслів, бо вони і є характеристиками дієслова.

При роботі селекційних правил виявляються лексичні ознаки дієслова, за якими треба обирати іменники з лексикону для контексту дієслова. Зв'язок слів за лексичними ознаками дозволяє створювати семантично вірні твердження.

Зауважимо також, що у реченні може бути кілька дієслів, поняття суб'єкту або об'єкту може відноситися до певного дієслова або до всього речення. В результаті будуть створені С-показники речень та відповідні їм претермінальні ланцюги, що містять лексичні ознаки слів майбутніх речень. На рис. 4.5 поданий С-показник для речень типу «Дівчинка хоче взяти яблуко».

На рисунку символи *F, M, V,* H подають комплексні символи. Тепер у претермінальному ланцюгу (4.5) замість комплексних символів M та V подається їх представлення через лексичні ознаки контексту дієслова.

Претермінальний_ланцюг([«+N», «+Загальність»,
«+Зліченність», «+Істотність», «+Людяність»],
[«+Aux», «+Модальне», «+Істотність»], (4.8)
[«+V», «+Істотність», «Aux–», «–Істотність», «–NP»]
[«+N», «+Загальність», «+Зліченність», «Істотність»]).

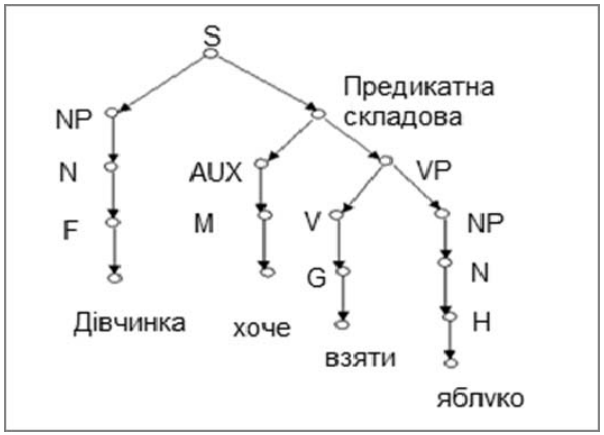


Рисунок 4.5 – С–показник для речень типу «Дівчинка хоче взяти яблуко»

4. Після створення С- показників та претермінальних ланцюгів створюються речення. Перебираються компоненти лексикону і слова, що підходять за лексичними ознаками вставляються у претермінальний ланцюг замість комплексних символів. Нижче поданий текст програми, що генерує всі можливі речення обраного типу. Разом з реченнями подаються їх С-показники.

```
Global domains
слово = string
list = слово*
l = list*
р = речення(г_і,  п_с)
п_с = предикатна_складова(д_д, г_д)
г_д = група_дієслова(д, г_і)
г_і = група_іменника(і)
і = іменник(list)
д = дієслово(list)
д_д = допоміжне_дієслово(list)

Global Facts
Nondeterm лексикон(слово, list)
Nondeterm претермінальний_ланцюг(list, list, list, list)
single ознаки_слова(list)
```



```
single іменник1(list)
single допоміжне_дієслово(list)
single дієслово(слово)

Predicates
Nondeterm база(р)
Nondeterm пр_речення(р)
Nondeterm пр_предикатна_складова(п_с)
Nondeterm пр_група_дієслова(г_д)
Nondeterm пр_група_іменника(г_і)
Nondeterm пр_іменник(і)
Nondeterm пр_дієслово(д)
Nondeterm пр_допоміжне_дієслово(д_д)
Nondeterm комплексний_символ(слово)
Nondeterm субкатегорії (слово, слово)
Nondeterm append(list, list, list)
Nondeterm породження()
Nondeterm роби()
Nondeterm пошук(слово, list)

Goal
роби().

Clauses
ознаки_слова([]).
іменник1([]).
дієслово(«»).
допоміжне_дієслово([]).

append([], L, L).
append([H|L1], L2,[H|L3]):-!, append(L1, L2, L3).

лексикон(«яблуко»,[«+N», «+Загальність», «+Зліченність»,
        «–Істотність»]).
лексикон(«взяти»,[«+V», «+Істотність», «Aux–»,
        «–Істотність», «–NP»]).
лексикон(«налякати»,[«+V», «+Абстрактність», «Aux–»,
        «+Істотність», «–NP» ]).
лексикон(«хоче», [«+Aux», «+Модальне», «+Істотність»]).
```

```
лексикон(«може», [«+Aux», «+Модальне», «+Істотність»]).
лексикон(«може»,[«+Aux», «+Модальне», «+Абстрактність»]).
лексикон(«дівчинка»,[«+N», «+Загальність»,
        «+Зліченність», «+Істотність», «+Людяність»]).
лексикон(«хлопчик»,[«+N», «+Загальність»,
        «+Зліченність», «+Істотність», «+Людяність»]).
лексикон(«пес», [«+N», «+Загальність», «+Зліченність»,
        «+Істотність», «–Людяність»]).
лексикон(«темрява»,[«+N»,»+Загальність»,
        «–Зліченність», «+Абстрактність»]).

роби():- база(S),
          S = речення (група_іменника(іменник(N1)),
        предикатна_складова(допоміжне_дієслово(D_D),
        група_дієслова(дієслово(D),
        група_іменника(іменник(N2)))))),
        asserta(претермінальний_ланцюг(N1, D_D, D, N2)),
        породження(), fail.
роби().

база(S):- пр_речення(S).

пр_речення(речення (NP, Пр_скл)):-
        пр_група_іменника(NP),
        NP=група_іменника(іменник(N)), assert(іменник1(N)),
        пр_предикатна_складова(Пр_скл).

пр_предикатна_складова(предикатна_складова(М_Д, Г_Д)):-
        пр_допоміжне_дієслово(М_Д), пр_група_дієслова(Г_Д).

пр_група_дієслова(група_дієслова(V, NP)):-
          пр_дієслово(V), дієслово(К),
          пр_група_іменника(NP),
          NP=група_іменника(іменник(N)), пошук(К, N).

пр_група_іменника(група_іменника(NP)):- пр_іменник(NP).

пр_іменник(іменник (N)):-комплексний_символ(«N»),
        ознаки_слова(N).

пр_допоміжне_дієслово(допоміжне_дієслово(М)):-
```



```
        субкатегорії(«М»,_), ознаки_слова(М),
        assert(допоміжне_дієслово(М)).

субкатегорії(«М»,_):-іменник1(N),
        пошук(«+Істотність», N),
        assert(ознаки_слова([])),
        ознаки_слова(S1),
        append([«+Aux», «+Модальне», «+Істотність»],
        S1, S2), assert(ознаки_слова(S2)),
        assert(допоміжне_дієслово(S2));

        іменник1(N), пошук(«+Абстрактність», N),
        assert(ознаки_слова([])), ознаки_слова(S1),
        append([«+Aux», «+Модальне», «+Абстрактність»],
        S1, S2), assert(ознаки_слова(S2)),
        assert(допоміжне_дієслово(S2)).

субкатегорії(«V»,_):-іменник1(N),
        пошук(«+Істотність», N),
        допоміжне_дієслово(D_D), пошук(«+Істотність», D_D),
        субкатегорії(«+Істотність», «\Aux-»),
        ознаки_слова(S1), append([«+V», «+Істотність»,
        «\Aux-»], S1, S2), assert(ознаки_слова(S2));

        іменник1(N), пошук(«+Абстрактність», N),
        допоміжне_дієслово(D_D),
        пошук(«+Абстрактність», D_D),
        субкатегорії(«+Абстрактність», «Aux-»),
        ознаки_слова(S1),
        append([«+V», «+Абстрактність», «\Aux-»], S1, S2),
        assert(ознаки_слова(S2)).

субкатегорії(«+Істотність», «\Aux-»):-
      assert(ознаки_слова([])), ознаки_слова(S1),
      append([«+Істотність», «-NP»], S1, S2),
      assert(ознаки_слова(S2)),
```

```
        assert(дієслово(«+Істотність»));

        assert(ознаки_слова([])), ознаки_слова(S1),
        append([«-Істотність», «–NP»], S1, S2),
        assert(ознаки_слова(S2)),
        assert(дієслово(«-Істотність»)).

        субкатегорії(«+Абстрактність», «\Aux–»):–
        assert(ознаки_слова([])), ознаки_слова(S1),
        append([«+Істотність», «-NP»], S1, S2),
        assert(ознаки_слова(S2)),
        assert(дієслово(«+Істотність»)).

пр_дієслово(дієслово(V)):– субкатегорії(«V»,_),
          ознаки_слова(V).

комплексний_символ(«N»):–
          комплексний_символ(«+Загальність»),
          ознаки_слова(S1),
          append([«+N», «+Загальність»], S1, S2),
          assert(ознаки_слова(S2));

        комплексний_символ(«-Загальність»),
        ознаки_слова(S1),
        append([«+N», «-Загальність»], S1, S2),
        assert(ознаки_слова(S2)).

комплексний_символ(«+Загальність»):–
        комплексний_символ(«+Зліченність»),
        ознаки_слова(S1), append([«+Зліченність»], S1, S2),
        assert(ознаки_слова(S2));

        комплексний_символ(«-Зліченність»),
        ознаки_слова(S1),
        append([«-Зліченність»], S1, S2),
        assert(ознаки_слова(S2)).

комплексний_символ(«+Зліченність»):-
```



```
        комплексний_символ(«+Істотність»),
        ознаки_слова(S1),
        append([«+Істотність»], S1, S2),
        assert(ознаки_слова(S2));

        assert(ознаки_слова([])), ознаки_слова(S1),
        append([«–Істотність»], S1, S2),
        assert(ознаки_слова(S2)).

комплексний_символ(«–Загальність»):–
        комплексний_символ(«+Істотність»),
        ознаки_слова(S1),
        append([«+Істотність»], S1, S2),
        assert(ознаки_слова(S2));

        assert(ознаки_слова([])), ознаки_слова(S1),
        append([«–Істотність»], S1, S2),
        assert(ознаки_слова(S2)).

комплексний_символ(«+Істотність»):–
        assert(ознаки_слова([])), ознаки_слова(S1),
        append([«+Людяність»], S1, S2),
        assert(ознаки_слова(S2));

assert(ознаки_слова([])), ознаки_слова(S1),
        append([«-Людяність»], S1, S2),

        assert(ознаки_слова(S2)).

комплексний_символ(«–Зліченність»):–
        assert(ознаки_слова([])), ознаки_слова(S1),
        append([«+Абстрактність»], S1, S2),
        assert(ознаки_слова(S2));

        assert(ознаки_слова([])), ознаки_слова(S1),
        append([«–Абстрактність»], S1, S2),
        assert(ознаки_слова(S2)).

пошук(A,[A|_]).
пошук(A,[_|T]):-!, пошук(A, T).
```

```
породження():-претермінальний_ланцюг(N1, D_D, D, N2),!,
        лексикон(Слово1, N1), лексикон(Слово2, D_D),
        лексикон(Слово3, D), лексикон(Слово4, N2),
        write(Слово1, «», Слово2, «»,
        Слово3, «», Слово4, «»), nl, fail.

породження().
```

Результат роботи програми:

```
дівчинка хоче взяти яблуко.
дівчинка може взяти яблуко.
хлопчик хоче взяти яблуко.
хлопчик може взяти яблуко.
пес хоче взяти яблуко.
пес може взяти яблуко.
темрява може налякати дівчинка.
темрява може налякати хлопчик.
темрява може налякати пес.
Yes
```

Три останні речення вимагають обробки трансформаційними правилами для заміни називних відмінків на знахідні.

Резюме

Вимога спілкування діалогової системи з користувачем природною мовою пов'язана з необхідністю спілкування в будь-якому місці діалогу будь-якими реченнями. Інтерфейс може бути інтелектуальним або імітувати інтелектуальність.

Основними підходами до реалізації інтелектуальних інтерфейсів є граматичний, концептуальний і семантичний. Основними компонентами інтелектуального інтерфейсу являються компонент розуміння змісту речень природною мовою і компонент генерації речень природною мовою.

Імітація інтелектуальності не вимагає розуміння речень природною мовою. Серед методів імітації інтелектуальності інтерфейсу найбільш відомими є метод ключових слів.

Одною з основних проблем створення інтелектуального інтерфейсу є проблема відокремленні невірних граматично речень від

вірних. Генеративна лінгвістика пропонує вирішити цю проблему шляхом породження вірних речень за правилами породжуючих граматик. Генеративні граматики мають словник, який містить термінальні та нетермінальні символи; правила за якими будуються вірні для цієї граматики речення; початковий символ, з якого починається робота правил. Генеративні граматики бувають 4-х типів: вони різняться обмеженнями, які накладаються на систему правил.

Регулярні граматики породжують найпростіші типи речень. Механізм, що породжує ці речення називають кінцевим автоматом, що породжує речення. Автомат приймає кінцеву кількість станів і в кожний момент може приймати один стан. Коли автомат переходить з одного стану у інший він друкує слово. Такий автомат описується марковським процесом з кінцевою кількістю станів. Регулярні граматики застосовують для реалізації пошуку в Інтернеті.

Контекстнао – залежна генеративна граматика забезпечує вивід необмеженої кількості речень природної мови. Породження речень відбувається за трансформаційними правилами: базовими, що породжують С-показники речень та прості типи речень; трансформаційними правилами, що породжують поверхневу структуру та складні речення.

Базовий синтаксичний компонент має лексикон та правила. Правила синтаксичного базового компоненту граматики підрозділяються на типи правил: контекстна-вільні базові правила – деривації, контекстна – вільні лексичні правила, контекстна - залежні субкатегоріальні та селективні правила. Спочатку на базі деривації та системи лексичних правил будується претермінальний ланцюг та С-показник, які містять граматичні категорії та лексичні характеристики іменників. За контекстна – залежними правилами вони поповнюються лексичними ознаками дієслів. Переглядаються компоненти лексикону і якщо слово з лексикону має лексичні ознаки відповідні ознакам у претермінальному ланцюгу, то комплексний символ в ньому заміщується на слово. В результаті породжуються речення.

Контрольні питання

1 Охарактеризуйте метод шаблону і метод ключових слів.

2 Як застосовують базу знань для розуміння змісту речення?

3 На яких етапах виділяється зміст речення при граматичному підході?

4. Як формалізується зміст речення при граматичному підході?

5 Поясніть, які проблеми виникають при програмному розумінні природної мови?

6 Для чого застосовують метод шкалування? Які типи шкал вам відомі та як їх можна застосувати?

7 З яких компонент складається генеративна граматика?

8 Для чого застосовується деривація, лексичні правила?

10 Поясніть поняття «функція слова в реченні»?

11 Для чого застосовують селективні правила? На якому етапі породження речень застосовують лексикон і як?

Вправи

1 Написати та налагодити програму, яка застосовує метод ключових слів до інтерфейсу інформаційно-пошукової системи. Предметну область оберіть відому вам.

2 Додати до контекстна-залежних субкатегоріальних правил прикметник, змінити програму, що породжує речення і одержати речення.

РОЗДІЛ V. ПОДАВАННЯ ТА ОБРОБКА ЗНАНЬ

5.1 Реалізація інтелектуальної діяльності у системах

Історія виникнення інтелектуальних систем знала чимало систем, що реалізовували певну інтелектуальну діяльність людини, але спеціалісти не називали їх інтелектуальними системами.

Щоб зрозуміти чому, розглянемо як створювалися такі системи:

- людина сама розроблювала *вміст і структуру знань* про деяку ПДО, *вкладала в систему знання*;
- людина сама *створювала алгоритми розв'язування задач обраного нею типу*;
- *людина сама вкладала в систему* готові алгоритми. *Система не могла змінювати алгоритми*.

У таких системах людина все робить сама, а система тільки автоматично виконує дії.

Виходячи з вище сказаного, можна зробити вивід, що повинна *вміти інтелектуальна система*, щоб самостійно виконати конкретну інтелектуальну роботу.

Інтелектуальна система зобов'язана мати наступні властивості:

1. *Мати знання*. *Вміти навчатися в процесі своєї роботи*: одержувати нові знання від людини, з навколишнього середовища або виводом на знаннях;

2. *Утворювати необхідні алгоритми для розв'язування задачі*: динамічно, під час роботи системи. Застосовувати вже існуючі в базі знань алгоритми, адаптуючи їх до умов задачі або узагальнюючи алгоритми.;

3. *Розуміти та аналізувати текст задачі природною мовою*, щоб утворювати або обирати алгоритми розв'язування задачі;

4. *Використовувати для подавання знань та ведення діалогу з людиною* обмежену *природну мову*. Для цього система повинна мати інтелектуальний інтерфейс;

5. *Вміти формувати програми* за створеними алгоритмами і ініціювати їх.

Насправді, далеко не всі сучасні ІС мають вказані властивості. Деякі властивості непотрібні певній ІС. Системи, що керують

процесами автоматично, для повідомлення людині про аварійну ситуацію застосовують фіксовані речення або звук. Більшість систем лише частково реалізують перелічені властивості.

Системи, що мають вказані можливості, або мають їх частково називають ІС. Але властивості 1–2 повинні бути обов'язково.

У попередніх розділах ми розглядали методи створення інтелектуального інтерфейсу, метод розв'язування задачі шляхом її декомпозиції та застосування механізмів Прологу до створення динамічних алгоритмів. Цей розділ ми присвятимо базовій компоненті інтелектуальної системи – базі знань. Ясно, що без наявності знань, не можуть функціонувати інші властивості інтелектуальної системи.

Резюме

Основними функціями інтелектуальної системи є вміння самонавчатися; вміння створювати алгоритми та модифікувати існуючи для розв'язування поточної задачі; спілкуватися з людиною природною мовою, якщо це потрібно.

Контрольні питання

1. Чим відрізняється інтелектуальна система від системи, що написана традиційними методами.
2. Що мається на увазі, коли кажуть, що система самонавчається?
3. Для чого інтелектуальній системі спілкування природною мовою?
4. Пояснити, як при застосуванні формального логічного виводу формується алгоритм розв'язування задачі.

5.2 Знання. Бази знань

5.2.1 Класифікація знань

Систему, що немає знань, не може їх здобувати, ніхто не назве інтелектуальною. *Знання людини* – результат вивчення людиною реальної або абстрактної ПДО. Знання подаються у пам'яті людини фактами про об'єкти ПДО та можуть бути перевірені на практиці. Знання людини містять конкретні факти і факти-закономірності про об'єкти певної ПДО.

Знання бувають різних типів: житейські; особові; елементарні; наукові; прикладні.

Житейські знання людини містять конкретні факти і факти закономірності про певну ПДО з життя людини. Наприклад, факт – закономірність: «Сонце сходить на сході.». До *особових знань* відносять знання певної особи, які вона набула сама. Наприклад, знання здобуті в результаті спостереження за природою. Знання про підсвідому поведінку мають, як люди, так і тварини. Такі знання називають *елементарними*. Наприклад, знання про послідовність рухів при ходьбі. Елементарні знання відносять до навиків істоти. Такі навики працюють автоматично без втручання свідомості.

Наукові знання містять конкретні факти з певної ПДО, пояснюють їх у системі знань відповідної науки. Наукові знання містять закономірності загальні для множини конкретних фактів. На основі закономірностей можна прогнозувати інші конкретні факти. Наукові знання дозволяють людині розуміти дійсність. Наукові факти характеризуються логічною обґрунтованістю, їх існування можна довести, а при повторному експерименті при тих же умовах новий науковий факт повинен мати місце.

Наукові знання, які одержано в результаті практичного досвіду, називають *емпіричними*. *Теоретичні наукові знання* одержують у результаті творчого процесу – міркування над предметною областю. Теоретичні наукові знання дозволяють людині мати цілісний погляд на закономірності й важливі зв'язки реального світу. До прикладних знань відносять наукові знання, які застосовують у практичній діяльності людини.

Знання про предметну область повинні містити: *спеціальні знання* про певну ПДО і *загальні знання* про закони навколишнього світу. *Спеціальні знання про предметну область* містять принципи, закони, семантичні та причино – наслідкові зв'язки між поняттями ПДО, конкретні факти, які були одержані в результаті практичної та наукової діяльності. Знання повинні дозволяти фахівцям ставити та *розв'язувати задачі предметної області.* До *загальних знань* відносять: часові закони та зв'язки між часовими поняттями, просторові закони та зв'язки між просторовими поняттями, закони причини-слідства, закони про логічні зв'язки між фактами.

Загальні й спеціальні знання розподіляються на *декларативні* та *процедурні* знання. Декларативні знання – це конкретні факти ПДО. Процедурні знання містять алгоритми, процедури, функції, правила,

методи, методики. В знаннях можуть зберігатись алгоритми, які створює система, щоб застосовувати їх для аналогічних задач.

В знаннях також зберігаються базові процедури, з яких будуються алгоритми. Базові процедури називають *метапроцедурами.* Прикладом метапроцедур могли б бути оператори мови програмування, якщо б інтелектуальна система сама вміла формувати програми. Існує напрям штучного інтелекту, що займається питаннями автоматичного програмуванням на мові програмування Ліспп.

5.2.2 Властивості знань

Знання ІС подаються в поняттях. Кожне поняття розкриває значення слова і представлено групою тверджень. Кожне твердження поняття вказує на певну властивість об'єкту, який подається словом. Подамо поняття «студент» фактами мовою Пролог:

база(«АКО», «студент», «людина»).

база(«вчиться», «студент», «університет»).

На слово можна дивитися як на ідентифікатор поняття, і застосовувати його для звертання до поняття. У фактах слово «студент» є ідентифікатором поняття. Одержати всі твердження поняття «студент» можна відібрав факти за другим аргументом.

Першим аргументом кожного факту є властивість слова, що визначається, а третім аргументом значення вказаної властивості для слова. Наприклад, властивість **АКО** (a king off) вказує, що студент відноситься до класу людина.

Основні властивості знань:

1. *Внутрішня інтерпретація знань.* Кожне твердження знань подається зв'язком між словами, які в свою чергу є ідентифікаторами інших понять. Таким чином, кожне поняття знань визначається через інші поняття знань. У *знаннях людини* жодне поняття не може визначатись через поняття, що знаходяться зовні її знань, бо зрозуміти нове слово – це означає пов'язати слово із значеннями вже відомих слів.

Вказана властивість не випадковість, що залежить від обраних речень, а *фундаментальна властивість знань людини.* У реальних ІС така властивість реалізується лише для спеціальних знань ПДО або

додатково до спеціальних знань ПДО застосовуються знання суміжних ПДО (суміжні світи Хомського) [22].

Проілюструємо вказану властивість знань на прикладі значень слів з тлумачного словника російської мови мовознавців РАН С. І. Ожогова і Н. Ю. Шведової.

У словнику є означення слів:

«Размер – это *величина* в каком-то измерении».

«Величина – размер, объем, протяженность предмета».

«Объем – величина чего-нибудь в *длин*у высоту и ширину».

«Протяженность – одна из основных характеристик пространства, выражающая его *размеры».*

«Длина – Величина, протяжённость чего-н. в том направлении, в котором две крайние точки линии, плоскости, тела лежат, в отличие от ширины, на наибольшем расстоянии друг от друга».

Застосуємо розглянуті означення слів для утворення фрагменту знань. На рис. 5.1 показано, що поняття кожного слова визначається через поняття інших слів. Для спрощення показані не всі слова, щоб обмежити фрагмент знань.

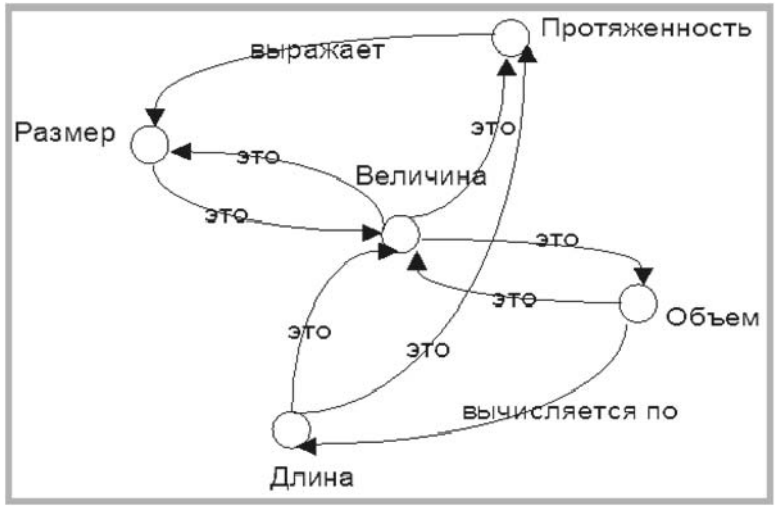


Рисунок 5.1 – Внутрішня інтерпретація знань

2. *Одиниці знань повинні мати унікальні ідентифікатори*. Унікальні ідентифікатори дозволяють звертатися до будь-якої одиниці знань однозначно. До поняття можна звертатися за словом. Одне слово може мати різні значення. Щоб зробити слово унікальним ідентифікатором до слова додають номер. Наприклад: «заміна1» – дія; «заміна2» – той, хто замінює когось.

3. *Знання цілісна, єдина структура.* Знання однієї ПДО являють собою цілісну, єдину структуру, всі поняття якої взаємопов'язані за змістом. У знаннях не може бути ізольованого слова, бо для нього не буде визначено поняття. Знання про одну ПДО не можуть складатися з кількох структур.

Цілісність знань дозволяє рухатися за поняттями знань. Наприклад, за поняттям «студент» перейти до поняття «університет»:

```
Goal
readln(Слово), знайти(Слово),!.
Clauses
база(«вчиться», «студент», «університет»).
база(« вчиться», «студент», «університет»).
база(«АКО», «університет», «учбовий заклад»).
база(«дає», «університет», «фах»).
знайти(Слово2):- база(_, Слово2, Слово3), !, роби(Слово3).
роби(Слово2):-база(Слово1, Слово2, Слово3),
                write(Слово1, «», Слово2, «», Слово3), nl, fail.
роби(_).
```

Для вводу нового поняття у цілісну структуру знань, слова його тверджень треба пов'язати із словами, що вже існують у знаннях. При відсутності слова для нього можна розробити нове поняття і поповнити базу знань. Вказаний метод поповнення знань дозволяє їх поповнювати не руйнуючи цілісності структури. Знання ІС завжди обмежені ПДО або проблемною областю.

4. *Замкненість знань.* Серед понять структури знань немає базових понять, через які можна визначати інші поняття знань, а їх не визначати. Якщо зображувати структуру знань графічно, у вигляді мережі (рис. 5.1), то кожне слово буде вузлом мережі, а зв'язки між

словами пов'яжуть вузли мережі. Замкненість знань відобразиться тим, що кожен вузол має хоча б один вхід і хоча б один вихід. Для базового поняття вузол мав би тільки входи.

Якщо у прикладі з поняттями «величина», «размер», «длина», «объем», прийняти за базове поняття «величина», то для вузла «величина» були б тільки входи. Здається, відсутність базових понять може заганяти людину у глухий кут. Однак, у людини завжди є певний запас слів та можливість звернутися до об'єктів реального світу, щоб зрозуміти значення слова.

5. *Гнучкі семантичні зв'язки понять.* Організація знань дозволяє пов'язувати у твердженні будь-які слова за змістом, не фіксуючи зв'язки попередньо у об'яві предикатів для фактів. Обмеженням до пов'язування є тільки семантика. Властивість необхідна, бо неможливо знати, якими твердженнями будуть поповнюватися знання. Вказана властивість знань робить їх відкритими для поповнення.

Знання ІС постійно поповнюються, тобто система самонавчається. Щоб *знання могли постійно поповнюватися, їх відокремлюють від програмного забезпечення.* Інакше програмне забезпечення (ПЗ) прийшлось би постійно компілювати. Цю властивість знань мають і бази даних.

6. *Гнучка структурованість знань.* Необхідно мати можливість включати будь-яку одиницю знань в іншу будь-яку одиницю знань. Обмеженням до включення є тільки семантика. Гнучка структурованість характерна для знань.

У програмах мовою Пролог не можливо робити аргументом факту інший факт, але можна аргументи фактів робити структурами будь-якої вкладеності. Для цього всі структури треба попередньо об'явити, тобто гнучкої вкладеності знань мовою Пролог реалізувати неможливо. На відміну від Прологу мова програмування Lisp дозволяє гнучко вкладати твердження понять.

Прикладом структур, що мають кілька рівнів вкладеності є сценарій або опис ситуації. Сценарій складається із сцен, сцени в свою чергу складаються з опису послідовності дій. Складовою частиною сценарію може бути інший сценарій. Сценарій подається як факт, аргументами якого є тип сценарію, ім'я сценарію та *цілісна структура з сцен*. До факту звертаються за іменем сценарію та типом.

Ситуація (з латинської «становище») є сукупністю тверджень, що показують зафіксований стан певного об'єкту та умови, при яких мають місце ці твердження у ПДО. Ситуацію також можна подати фактом, аргументами якого є ім'я ситуації, список умов та список тверджень, що описують стан. Ситуація фіксується у певний момент або у певному інтервалі часу.

Розглянемо приклад ситуації: «Весною під час паводку залило городи селян, що жили біля річки.». Умовами ситуації є «паводок» і «проживання селян біля річки», дія вказана у іменній групі «залило городи селян», дія відбулася у певний момент «весною». Кажуть, що у описі ситуації поняття взаємопов'язані ситуативними відношеннями, у сценарії сценарними відношеннями.

7. *Метрика на знаннях.* При роботі із знаннями часто треба порівнювати поняття за певною ознакою, для чого об'єкти, події, ситуації впорядковують за цією ознакою. Як ми бачили у попередньому розділі для порівнювання понять можна ввести відстань між поняттями за змістом (метричні шкали). Відстань між поняттями може також подаватися словами з квантифікаторами на шкалі (лінгвістичні порядкові шкали). А для впорядкування дій (подій, ситуацій) достатньо зафіксувати для кожної з них місце відносно інших дій (подій, ситуацій) на шкалі (порядкові шкали).

На початку XX сторіччя американський психолог Ч. Осгуд засновуючись на шкалах спробував побудувати семантичний простір понять. Для чого він відібрав біля 400 опозиційних шкал за трьома групами: сили, оцінки, активності. Прикладами кожного з напрямів є: шкала сили «великий – малий»; шкала оцінки «добрий – злий»; шкала активності «швидкий – повільний». Осгуд провів психологічний експеримент, у якому кожній людині, що брала участь у експерименті, пропонувалось розмістити на шкалах слова. Кожне слово треба було розмістити на всіх 400 шкалах. Обробивши результати статистичними методами, Осгуд одержав положення кожного слова на кожному напрямі числом. Осгуд обрав кожну групу шкал за осі координат і одержав трьохмірний семантичний простір. Виявилось, що слова близькі за темою, знаходились ближче одне до одного, ніж слова за іншими темами. Тобто, відбувалася кластеризація (структурування) понять. Тим самим Ч. Осгуд підтвердив, що на знаннях існує метрика.

8. *Знання активні*. Існує 2 погляди на активність знань:

а) знання містять алгоритми, методи, методики, правила – продукції, процедури, функції, їх називають процедурними знаннями. Причому, процедурні знання первинні, а декларативні знання (факти) вторинні. Це означає, що декларативні знання подаються разом із процедурними знаннями, які їх застосовують.

б) знання цілісний, багаторівневий організм, йому притаманні самонавчання, самоорганізація і виконання великої кількості потаємних обчислень. Потаємні обчислення активізуються, якщо у знання надходить нова інформаційна структура. *Метою активізації обчислень є розуміння інформаційної структури, що надійшла, та заходження її місця у знаннях.*

Другий підхід цікавіший, але його важче здійснити. Підхід реалізується в системі MARGIE [19]. Для розуміння необхідно знайти зв'язки між вхідною інформаційною структурою та існуючими знаннями. Активізація активних процесів (процедурних метазнань) виконується без ініціації функцій, автоматично, просто у відповідь на нову інформаційну структуру. Такі процеси називають умовиводами.

У людини по більшості умовиводи виконуються несвідомо, без її контролю. Умовиводи виконуються паралельно і асоціативно, тобто кожне твердження структури стимулює одночасно всі типи умовиводів, що шукають асоціативні за змістом твердження. Умовиводи слабко направлені до цілі. Ціль формується під час роботи умовиводів. Ціллю роботи активних процесів є збільшення кількості зв'язків між новою інформацією і структурами знань та утворення значно більшої інформаційної структури на базі вхідної. Розширена структура дозволяє виявити причини виконання дії; умови виконання дії; прогнозувати наступні події, оцінювати стан об'єкту, тощо.

9. Знання – є система. Системою називають об'єкт або процес, елементи якого пов'язані між собою в єдине ціле багатьма різними зв'язками та спільною метою. Знання теж цілісна, єдина структура об'єднаних метою.

Структурою системи називають сукупність стійких зв'язків між елементами системи, що забезпечують її цілісність та зберігання основних властивостей при зовнішньому впливу або внутрішніх змінах. Поповнення знань також не змінює їх цілісність. Вказаний метод поповнення знань дозволяє їх поповнювати не руйнуючи внутрішньої структури знань.

10. *Спеціальна організація знань ІС не повинна залежать від змісту інформаційних одиниць знань.* Це дозволяє створювати універсальне програмне забезпечення для ведення знань – систему керування базою знань. Знання містять різні типи інформаційних одиниць: твердження, поняття, ситуації, сценарії. Знання повинні мати таку організацію, щоб дозволяла однаково звертатися до них незалежно від їх змісту.

Резюме

Знання бувають різних типів: житейські; особові; елементарні; наукові; прикладні. Наукові знання розподіляються на емпіричні та теоретичні. Теоретичні наукові знання дозволяють людині мати цілісний погляд на закономірності й важливі зв'язки об'єктів реального світу. Спеціальні знання – це знання про конкретну ПДО. Декларативні знання – це конкретні факти ПДО. Процедурні знання містять алгоритми, процедури, функції, правила, методи, методики необхідні для розв'язування задач. Процедурні знання можуть бути закладенні в систему керування знаннями ІС. Знання повинні дозволяти фахівцям ставити та *розв'язувати задачі ПДО. З*агальні знання - це знання про закони та відомі факти навколишнього світу.

Контрольні питання

1. Як подають поняття у базі знань?
2. Чи реалізується гнучка структурованість інформаційних одиниць даних у базах даних?
3. Чим відрізняються гнучкі семантичні зв'язки у твердженнях знань від зв'язків між даними у базах даних?
4. Поясніть, що означає твердження «знання активні».

5.3 Моделі баз знань

Існують різні моделі організації знань. Найбільш відомі – це логічна, мережева, продукційна моделі [20].

Часто, до перелічених моделей додають також фреймову модель, але фреймова модель, це багаторівнева мережева модель, вузли, якої будуються на основі фреймів (структур). Інакше кажучи, фрейм – спосіб подавання інформаційних одиниць знань у мережевій моделі знань. Відомо багато різних модифікацій основних типів моделей за рахунок варіацій подавання інформаційних одиниць знань.

Назвемо *базою знань* спеціальну їх організацію за певною моделлю, що забезпечує перелічені вище властивості знань.

Мовою Пролог зручно реалізовувати логічні та продукційні моделі організації знань. Певний час логічні моделі вважали застарілими, хоча сьогодні тезауруси – база знань логічної моделі, набувають знову популярності але як словники, в яких значення слів подається через групу схожих за змістом слів.

Пролог дозволяє наступне.

1. Легко подавати інформаційні одиниці знань фактами.
2. Завантажувати або змінювати факти під час роботи програми, не перебудовуючи саму програму.
3. Легко створювати компоненти продукцій логічними формулами.
4. Подавати продукції правилами Прологу.
5. Автоматично перебирати велику кількість фактів за багатьма ключами і реалізовувати це достатньо просто.
6. Реалізовувати логіку людини при міркуванні над знаходженням розв'язку задачі.

5.3.1 Продукційна модель бази знань

Сьогодні найбільш популярною моделлю бази знань являється продукційна модель. Системи, що базуються на цій моделі називають продукційними. Успіх продукційних систем можна пояснити тим, що вони *моделюють в певній мірі як міркування людини так і організацію знань людини.*

Подавання *процедурних знань системи правилами типу «якщо, то» (продукцій)* дозволяє наочно імітувати логіку людини, зокрема експерта.

Теж саме можна сказати і про організацію продукційної системи. За роботами З. Фрейда людина має довгочасну і короткочасну види пам'яті. Довгочасна пам'ять зберігає постійні знання необмежений строк. Короткочасна пам'ять зберігає тільки певний час невеликий об'єм знань необхідний в цей момент. Для короткочасної пам'яті характерно витискування непотрібної інформації та поновлення інформації на той час, на який вона потрібна.

Подібно сказаному сучасна продукційна програмна система складається з бази знань – постійної пам'яті; робочої пам'яті – короткочасної пам'яті; програмного інтерпретатора продукцій; області

черги, де зберігаються правила, що чекають виконання; списку підзадач, які потрібно розв'язати для розв'язку основної задачі; області метаправил, що керують роботою системи продукцій (рис.5.2).

База знань продукційної системи містить множину процедурних знань – продукцій; постійні декларативні знання: визначення, що відомі як достовірні; концепції; узагальнення базових об'єктів. У робочу пам'ять входять опис задачі, поточні результати, поточні гіпотези. Декларативні знання зручно подавати фактами мови Пролог.

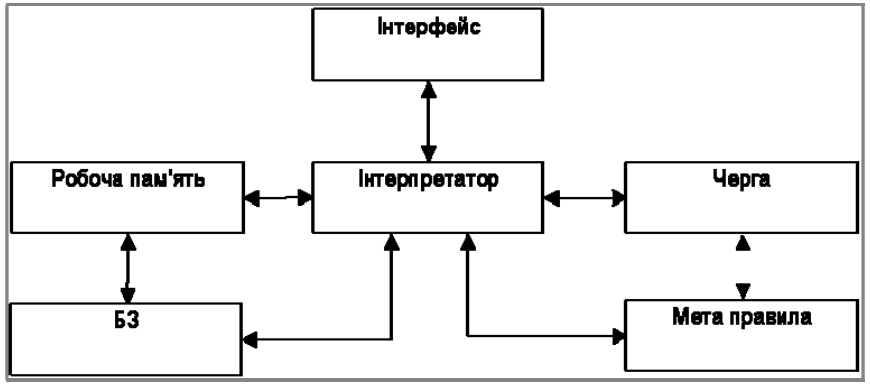


Рисунок 5.2 – Схема продукційної системи

Інтерпретатор обирає за необхідністю продукцію ефективну для рішення задачі. Інтерпретація продукції зводиться до пошуку фактів умови правила (якщо) і при знаходженні фактів умови, включення фактів про результат виконання дії (то).

Метаправила застосовують з різними цілями. Вони подають стратегію розв'язування задачі. Умови застосування метаправил описують умови розв'язуються підзадачі, що треба виконати для її розв'язку. Метаправила також допомагають інтерпретатору обирати ефективні правила. Вони можуть подавати моделі правил, за якими можна сформувати нові правила. За метаправилами можна змінити базу знань: додати нові продукції, вилучити або замінити продукції.

Робота продукційної системи. У базах знань продукційних систем знання подаються модулями, де кожний модуль знань застосовується для своєї проблемної області. Під проблемною

областю будемо розуміти знання ПДО необхідні для розв'язку задачі. Складна задача може декомпозуватися на кілька підзадач, для розв'язування кожної підзадачі можуть застосовуватися знання про свою проблемну область. Множина продукцій разом з відповідними їм постійними декларативними знаннями розподіляється за проблемними областями.

На початку роботи системи робоча пам'ять містить факти про умову задачі. Вибір поточної продукції інтерпретатором засновано на процедурі уніфікації Прологу. Механізм інтерпретації продукції застосовує ті продукції, яким відповідає зміст робочої пам'ять, але таких продукцій може бути кілька. Звідси виникає неясна ситуація, яку з продукцій треба виконати. Вибір продукції до виконання можна побудувати на різних принципах залежно від задачі. Принципи вибору продукції подані у параграфі 5.3.2. Результат роботи продукції зберігається у робочій пам'яті, що дозволяє обирати для роботи інші продукції. Факти робочої пам'яті можуть заміщатися новими, вилучатися і поповнюватися новими подібно короткочасній пам'яті людини. Таким чином вистроюється ланцюг дій, що приводить до розв'язку задачі. Такий метод організації та роботи із знаннями називають *продукційною моделлю*.

Вивід результату може виконуватися в прямому напрямі від умов задачі до результату або в зворотному напрямі від цілі до умови задачі. Другий спосіб реалізується у Пролозі і застосовується частіше.

Взагалі метод вибору продукцій інтерпретатором залежить від типу задач, що розв'язуються продукційною системою.

Формально кожна продукція подається моделлю:

$$\{I\ ;\ Q\ ;\ P\ ;\ A => B\ ;\ N\}, \qquad (5.1)$$

де I – *ідентифікатор продукції*, без ідентифікатору не можна маніпулювати нею. Q – в*казує на проблемну область застосування даної продукції*. Для складної задачі проблемні області створюють ієрархічну структуру. Наприклад:

$$Q1 \supset Q2\ \&\ Q3;\ Q2 \supset Q4\&\ Q5;\ Q3 \supset Q6\&\ Q7\ldots \qquad (5.2)$$

Вказівка проблемної області дозволяє зробити вірний вибір продукції та прискорити його.

A => *B* називають ядром продукції. Знак => називають секвенцією (слідування). Ядро продукції читається: «якщо *A* має місце, то *B* теж має місце». Секвенція може означати логічний наслідок. Однак, інтерпретація ядра продукції залежить від того, що означає *A* і *B*. Наприклад, *A* умова або стан, а *B* дія або подія, що відбувається при вірності умови (стану). Продукції дозволяють виконувати вивід нових тверджень без жорстких обмежень характерних для логічного виводу. Ядро може мати більш складну конструкцію: Якщо *A*, то *B*, інакше *C*. Компоненти ядра подаються логічними формулами або фактами.

Ядра продукцій класифікуються на детерміновані та не детерміновані. У детермінованих ядрах при інтерпретації ядра після виконання *A*, *B* виконується обов'язково (необхідність). У не детермінованих ядрах при інтерпретації ядра після виконання *A*, *B* може виконуватися або ні (можливість).

Для не детермінованих ядер продукцій вказують оцінку реалізації В. Наприклад, Якщо кинути кубик (*A*), то з ймовірність 1/6 випаде шість (*B*). Оцінка може задаватися числом або значенням лінгвістичної змінної (наприклад, «навряд»). Такі продукції називають прогнозуючими наслідки.

P – це умова застосування ядра продукції. Умова *P* описує певну ситуацію, що має місце для області *Q*, яка розглядається. Умова *P* подається логічним виразом. Якщо *P* істинно, то перевіряється умова *A* з ядра продукції. Умова *A* описує стан певних об'єктів в умовах ситуації *P*. Якщо ситуація незмінна, то умова *P* не застосовується.

N – пост умова продукції. Пост умова активізується, якщо дія ядра В виконана. *N* – стан або дія – наслідок виконання дії *B*. Пост умова подається у формі логічного виразу.

Щоб зрозуміти наочно застосування компонент продукцій розглянемо приклад фрагменту тексту, за яким можна зробити продукцію. *«Взимку, під час хуртовини, діти, що жили далеко, не ходили до школи, щоб не заблукати.».* У прикладі умовою застосування ядра *P* є логічна формула «взимку» & «під час хуртовини», стан дітей *A* – «жили далеко», вказує на наслідок *B* – «не ходили до школи», пост умова *N* – «не заблукати».

5.3.2 Принципи реалізації системи керування продукціями

Для системи продукцій розроблюють процедуру керування продукціями. Процедура активізує ті продукції, для яких умова Р істинна і обирає з них одну продукцію для інтерпретації (виконання продукції). Вибір продукції з групи активізованих продукцій є проблемою.

Системи керування можуть базуватися на різних принципах: вичерпного перебору, оцінки, керуванні метаправилами, «стопки книжок», задовгої умови, пріоритетного вибору, класної дошки тощо.

1. *Принцип «вичерпного перебору»* реалізується перебором всіх можливих виводів за активізованими продукціями. Обирається той ланцюг продукцій, який приводить до цілі. Принцип вживається, якщо ціль сформульовано точно і при невеликої кількості варіантів перебору.

2. *За принципом «оцінки»* спочатку активізуються продукції, умови яких підходять. З них для роботи вибирається продукція з максимальним значенням оцінки. Максимальне значення оцінки може обиратися за різними критеріями.

Прикладами критеріїв можуть бути:

- ступінь важливості фактів, що зуживаються в умовах Р;
- застосування факту в умовах Р для попередньої продукції;
- ступінь важливості правила для досягнення цілі;
- найбільш специфічне для задачі правило;
- найменша різниця між результатом і ціллю (див. 4.2).

3. *За принципом «мета правила»* стан системи описують мета правилом. Вибору продукцій метаправилами віддають перевагу, бо застосовується той же механізм інтерпретації, що для продукцій. Прикладом застосування мета правила є подавання мета правила правилом формальної граматики, що за іменами продукцій звужує групу продукцій, яку треба інтерпретувати.

4. *Принцип «стопки книжок»* реалізується аналогічно реальній ситуації, коли студент готується до сесії. Необхідні книжки складені у стопку на столі. При частому використанні книги зверху знаходиться та

книга, якою часто користувалися. За цим принципом вибирається продукція з максимальною частотою використання при певних умовах.

Продукційна система, що самонавчається, реалізує цей метод для адаптації до зовнішнього середовища при певних умовах. Для цього система попередньо зберігає досвід спілкування з середовищем: опис ситуації, дії, частоту застосування, результати.

5. *Принцип «задовгої умови»* вимагає, щоб факти і продукції були прив'язані до типових ситуацій і впорядковані за відношенням «загальне – частинне». Якщо у множині продукцій, що відносяться до вузьких ситуацій, є продукція з найбільш довгою умовою і умова істина, то за сенсом обирається така продукція. У довгій умові про вузьку ситуацію більше інформації. Опису вузькій ситуації є більше довіри ніж до опису широкої ситуації.

6. *Принцип «пріоритетного вибору»* реалізується вибором продукції, з більшим статичним або динамічним пріоритетом. Статичні пріоритети продукцій вказує експерт за важливістю продукцій для певної задачі, підзадачі. Динамічні пріоритети формуються під час роботи системи продукцій.

7. *Принцип «класної дошки»* зручно розглянути на прикладі. Нехай групі експертів, що сидять біля великої класної дошки, пропонується розв'язати разом *складну комплексну проблему*. Кожний експерт спеціаліст у своїй ПДО, яка має відношення до проблеми. На дошці записано формулювання проблеми і вихідні дані. Якщо хтось з експертів вирішує, що він може щось сказати, то він робить обчислення і результат записує на дошці. Цей результат може допомогти іншим експертам також додати свої результати, які ведуть до рішення проблеми. Процес закінчується, коли проблема вирішена.

Аналогічно прикладу, у оперативні пам'яті виділяється область «класна дошка». Класна дошка – це робоча спільна пам'ять для паралельних процесів з різних ПДО, що працюють над проблемою. Під процесом будемо розуміти певну систему продукцій з модуля бази знань, що розв'язує свою задачу.

При сумісній роботі на «класну дошку» в спеціальне місце робочої пам'яті тимчасово заносять фрагмент знань над яким працюють всі процеси. На класній дошці пишуть об'яви для процесів різних областей. *Паралельні процеси знаходять на «класній дошці»*

необхідну їм інформацію для ініціації дій – умови застосування продукцій; виконують дії і самі заносять результати своїх дій, що може згодитися іншим процесам. *Активізовані продукції всі мають доступ до класної дошки, але працювати буде тільки та продукція, для якої з'явилась інформація на класній дошці.* Робота продовжується до тих пір, поки всі задачі не будуть розв'язані, а проблема вирішена.

5.3.3 Реалізація продукційної системи засобами Прологу

Множина продукцій разом з відповідними їм статичними фактами розподіляється за проблемними областями.

Мовою Пролог зручно записати кожну компоненту продукції окремим правилом (логічною формулою). Для прикладу складемо компоненти продукції №1 за реченням: *«Взимку, під час хуртовини, діти, що жили далеко, не ходять до школи, щоб не заблукати.»*.

```
p(1):- факт(«взимку») and факт(«хуртовина»).
a(1):- факт(«діти жили далеко від школи»).
в(1):- assert(факт(«діти не ходять до школи»)).
n(1):- assert(факт(«діти не заблукали»)).
```

Інтерпретація продукцій може виконуватися за правилом:

```
продукція(N):- p(N), a(N), в(N), n(N).
```

Кожну компоненту продукції ідентифікують номером продукції, що дозволяє інтерпретувати компоненти продукції відповідно номеру.

У програмі компоненти продукцій групуються окремими множинами: умовами застосування ядер продукцій *P*; умовами ядер продукцій *A,* діями *B* та постдіями *N*.

Наприклад:

```
P(1):-…   A(1):-…   B(1):-…
P(2):- …  A(2):-…   B(2):-…
```

Такий підхід зручно реалізовувати, якщо кількість продукцій невелика. При достатньо великій кількості продукцій, їх записують динамічними фактами і розподіляють по файлах, які завантажують в програму за необхідністю [21]. Наприклад, продукцію можна подати динамічним фактом, в якому перший аргумент умова *A* ядра продукції, другий аргумент дія *B*.

продукція(4,»Пасажир Джонс забув алгебру, яку вивчав у коледжі.», «Пасажир Джонс не спеціаліст з математичної фізики»).

Застосовуючи продукції зручно передавати логіку мислення людини, тому для прикладу реалізації продукційної системи була обрана проблемна область «Логічні задачі».

Розглянемо створення продукційної системи для розв'язку задачі «Сміт, Джонс і Робінсон». Ситуація, яка описана в умові задачі незмінна, тому компоненту продукцій Р ми не застосуємо.

Задача. Сміт, Джонс і Робінсон працюють у одній бригаді потягу машиністом, кондуктором і кочегаром. Порядок прізвищ і професій не обов'язково збігається. У потязі, в якому працює бригада їдуть три пасажири з тими ж прізвищами. Про пасажирів і робітників бригади відомі наступні дані.

Пасажир Робінсон живе в Лос-Анджелесі. Кондуктор живе в Омаху. Пасажир Джонс давно позабув алгебру, якої навчався у коледжі. Пасажир – однофамілець кондуктора живе в Чикаго. Кондуктор і один з пасажирів, відомий спеціаліст з математичної фізики, ходять у одну церкву. Пасажир Сміт завжди виграє у кочегара, коли їм випадає зустрітися за партією в більярд. Як звуть машиніста?

У завданні відсутні умови застосування ядра. Тому система керування вибором продукції застосовує метод оцінки до компонент ядер А. Система відбирає всі продукції, умови А яких в певний момент задовольняються. Критерієм вибору продукції для інтерпретації є важливість вибору продукції для ефективного досягнення цілі. Продукції розташовані в такому порядку, що продукція з меншим номером знаходить факт, який необхідний для продукцій з більшим номером. Тому з групи продукцій, умови яких задовольняються обирається продукція з меншим номером. Продукції, що вже застосовані, «вилучаються».

```
/* Робоча пам'ять */
Global Facts
Nondeterm факт (integer, string)
Nondeterm номер_продукції (integer)
Nondeterm вилучено (string, integer)
Single кількість (integer)
Single кількість_продукцій (integer)
```



```
Single мін (integer)

/* Статична База знань */
Predicates
Nondeterm інтерпретатор ()
Nondeterm вибір ()
Nondeterm метод_оцінки ()
Nondeterm критерій_мін ()
Nondeterm інтерпретація ()
Nondeterm A (integer, string, string, string)
Nondeterm B (integer)
Nondeterm N (integer)
Nondeterm кількість_фактів ()
Nondeterm write_A (string, string, string)

Goal
інтерпретатор ().

Clauses
Кількість (0).
Кількість_продукцій (0).
Мін (100).
Вилучено («А»,0).

    факт(1, «Пасажир Робінсон живе в Лос-Анджелес.»).

    факт(2, «Кондуктор живе в Омаху.»).

    факт(3, «Пасажир однофамілець кондуктора живе в Чикаго.»).

    факт(4, «Пасажир Джонс забув алгебру, яку вивчав у коледжі.»).

    факт(5, «Кондуктор і один з пасажирів, відомий спеціаліст з математичної фізики, ходять в одну церкву.»).

    факт(6, «Сміт завжди виграє в кочегара у більярд, коли їм випадає зустрітися»).


    A (1, S1, «», «») :- факт (4, S1).
    A (2, S1, S2, «»):- факт (5, S1), факт (2, S2).

    A (3, S1, S2, S3):- факт (1, S1), Факт (_, «Пасажир спеціаліст з математичної фізики живе в Омаху.»),
```

```
Факт (_, «Пасажир Джонс не спеціаліст з математичної фізики»),
        S2= «Пасажир спеціаліст з математичної фізики живе в Омаху.»,
        S3= «Пасажир Джонс не спеціаліст з математичної фізики».

        A(4, S1, S2, «»):- факт(1, S1), Факт(_, «Пасажир Сміт живе в
Омаху»), S2= «Пасажир Сміт живе в Омаху».

        A(5, S1, S2, «»):- факт (_, «Пасажир Джонс живе в Чикаго»),
        Факт (_, «Пасажир однофамілець кондуктора живе в Чикаго.»),
        S1= «Пасажир Джонс живе в Чикаго»,
        S2= «Пасажир однофамілець кондуктора живе в Чикаго.».

        A (6, S1, «»,  «»):- факт (_, «Сміт завжди виграє в кочегара у
більярд, коли їм випадає зустрітися»),
        S1= «Сміт завжди виграє в кочегара у більярд, коли їм випадає
зустрітися».

        A (7, S1, S2, «»):- факт (_, «Кочегар не має прізвище Сміт»),
        Факт (_, «Кондуктор має прізвище Джонс»),
        S1=«Кочегар не має прізвище Сміт»,
        S2=«Кондуктор має прізвище Джонс».


        B (1):- кількість (К), К1=К+1,
        Assert (факт (К1, «Пасажир Джонс не спеціаліст з
математичної фізики»)),
        Write («ТО»), write («Пасажир Джонс не спеціаліст з
математичної фізики»), nl, nl, ! .

        B (2):- кількість (К), К1=К+1,
        assert (факт (К1, «Пасажир спеціаліст з математичної фізики
живе в Омаху.»)),
        Write («ТО»), write(«Пасажир спеціаліст з математичної фізики
живе в Омаху.»), nl, nl, ! .

        B (3):- кількість (К), К1=К+1,
        assert (факт (К1,»Прізвище пасажира спеціаліста з
математичної фізики Сміт»)),
        Write («ТО»), write («Прізвище пасажира спеціаліста з
математичної фізики Сміт»), nl,
        К2 = К1+1, assert (факт (К2, «Пасажир Сміт живе в Омаху»)),
```



```
        Write («Пасажир Сміт живе в Омаху»), nl, nl, !.

        B (4):- кількість (К), К1=К+1,
        assert (факт (К1, «Пасажир Джонс живе в Чикаго»)),
        Write («ТО»), write («Пасажир Джонс живе в Чикаго»), nl, nl, !.

        B(5):- кількість (К), К1=К+1,
        assert(факт(К1, «Кондуктор має прізвище Джонс»)),
        Write («ТО»), write («Кондуктор має прізвище Джонс»), nl, nl, !.

B (6):- кількість (К), К1=К+1,
        assert (факт (К1, «Кочегар не має прізвище Сміт»)),
        Write («ТО»), write («Кочегар не має прізвище Сміт» ), nl, nl, !.

        B(7):- кількість (К), К1=К+1,
        assert (факт(К1, «Машиніст має прізвище Сміт»)),
        Write («ТО»), write («Машиніст має прізвище Сміт»), nl, nl.

N (Номер):-assert (вилучено («А», Номер)).

Інтерпретатор ():-факт (_, «Машиніст має прізвище Сміт»).
Інтерпретатор ():- кількість_фактів (), метод_оцінки (),
Інтерпретація (), retractall (номер_продукції (_)),
        Assert (мін (100)), !, інтерпретатор ().

кількість_фактів ():- факт (_,_), кількість (К), К1=К+1,
        assert (кількість (К1)), fail.
кількість_фактів().

вибір():- А(Номер,_,_,_), not (вилучено («А», Номер)),
        assert (номер_продукції (Номер)), fail.
Вибір ().

метод_оцінки ():- вибір (), критерій_мін ().

критерій_мін () :- номер_продукції (Номер), мін (М),
        Номер<М, assert (мін (Номер)), fail.
критерій_мін ().

Інтерпретація ():- мін (Номер), А (Номер, S1, S2, S3),
        write_A (S1, S2, S3), B (Номер), N (Номер).

write_A (S1, S2, S3 ) :- write («ЯКЩО»), write (S1), nl, S2= «», ! ;
```

```
write (S2), nl, S3 = «», ! ;
write (S3), nl.
```

У першій продукції виконується заміна умови про Джонса, що пов'язує два висловлювання з умови задачі. Не мати знання з алгебри, означає, що він не може бути спеціалістом з математичної фізики. Якщо ця продукція не буде обрана, то розв'язати задачу неможливо.

Інтерпретатор працює в циклі: оцінити продукції, обрати одну для інтерпретації, інтерпретувати продукцію. Інтерпретатор закінчує роботу за появою твердження відповіді на запитання задачі.

5.3.4 Мереживна модель бази знань

Формальна мереживна модель представлення знань визначається:

$$CM = \{I_m, C_1..., C_k, G_n(L) \}, \qquad (5.3)$$

де I_m – вузли мережі; $C_1... C_k$ – зв'язки між вузлами; $G_n(L)$ – відображення, за яким для конкретної задачі L обирають певний набір зв'язків між вузлами. Індекси *m, k, n* – непов'язані один із одним і приймають значення: 1, 2,... .

Між вузлами семантичної мережі для ПДО може бути кілька відношень. Відображення G_n фіксує зв'язки для конкретної проблеми ПДО. Відображення задають списком зв'язків, закономірністю.

У вузлах мережі розміщуються ідентифікатори понять (слова, група слів) ідентифікатори сценаріїв, інших складних структур. Ідентифікатори тверджень унікальні і формуються програмно.

Типи мереживних моделей визначаються залежно від типів відношень між вузлами. Наприклад, мереживну модель, у якої зв'язки між вузлами тільки класифікаційні, називають класифікаційною.

Мереживна модель у якої відношення між вузлами можуть подаватися словами з різною семантикою, називають *семантичною мереживною моделлю*. Семантична модель знань найбільш загальний опис навколишнього світу. На рис. 5.3 подано фрагмент семантичної мережі, що подає одне з визначень слова грунт.

Визначення слова «грунт» складається з фрази: «Розсипчаста речовина темно-бурого кольору, частина кори землі.». У вузлах семантичної мережі можуть розташовуватися об'єкти, процеси, властивості, явища і т. п.. Аналогічно відношеннями можуть бути будь-які поняття. Вибір залежить від змісту знань.

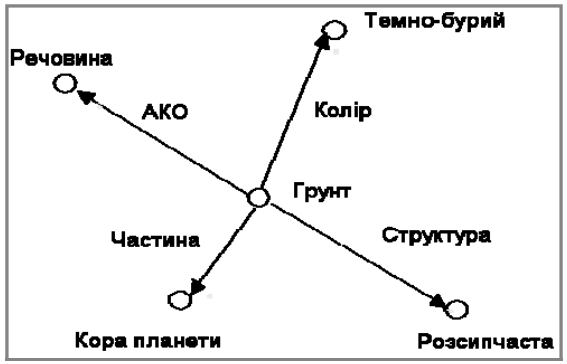


Рисунок 5.3 – Фрагмент семантичної мережі для поняття грунт

Існує кілька рівнів формалізації знань про ПДО [22]. Назвемо інтенсіоналом поняття його визначення через поняття більш високого рівня абстракції з вказівкою специфічних властивостей. Поняття інтенсіоналу застосовують у логічній семантиці для уточнення поняття змісту. Подане визначення поняття «грунт» є інтенсіональним. Екстенсіоналом поняття називають визначення поняття через поняття нижчого рівня абстракції, через дійсні факти.

На семантичних мережах можна робити вивід продукційними правилами [23]. Для цього у мережі шукається умова застосування ядра продукції А, такий вивід називають виводом за зразком. Залежно від роботи продукції дія В може замінити А або на інший факт або додати до семантичної мережі інший факт. Таким чином семантична мережа відображується на себе.

Резюме

Основними моделями організації знань інтелектуальної системи є логічна, мережева, продукційна та їх модифікації. Засоби мови Пролог зручно застосовувати для створення продукційної системи.

Продукційна система моделює міркування та організацію пам'яті людини. Логіка людини моделюється продукціями. Інтерпретатор системи активізує продукції, яким відповідають факти робочої пам'яті. При наявності кількох активізованих продукцій система керування продукціями застосовує певний принцип вибору продукції. Принципи обирають залежно від задачі. Продукції поступово змінюють зміст робочої пам'яті від цілі до розв'язку.

Формально продукція подається ідентифікатором; описом ситуації, при які може працювати ядро; ядром A => B (якщо A , то B); постумовою. Ядро може бути виду «якщо *A*, то *B*; інакше C». Дія *B* може виконуватися з певною ймовірністю.

Мереживна модель бази знань складається з вузлів, відношень між ними. При наявності кількох відношень між двома вузлами можна для задачі фіксувати певні відношення. Від типу відношень залежить назва мережі. У семантичній мережі застосовують різні відношення.

Існує кілька рівнів формалізації знань. Інтенсіоналом поняття називають його визначення через поняття більшої абстракції. Екстенсіоналом поняття називають визначення поняття через поняття більш нижчого рівня абстракції, через дійсні факти.

Контрольні питання

1. Для чого продукції групують за областю застосування?
2. Пояснити, чим відрізняються компоненти продукції *P* і *A*.
3. Яку функцію виконує система керування продукціями?
4. У яких задачах застосовують принцип «вичерпного перебору»? Чому? Як працює принцип «оцінки»?
5. Наведіть приклад речення, в якому: а) був би критерій «ступінь важливості фактів, що зуживаються в умовах *P*»; б) відсутні факти важливі в умовах *P*».

Вправи

1. Створити продукційну систему для розв'язку задачі «Обрати мобільний телефон». Опит властивостей телефону та додавання їх у базу знань виконувати за продукціями.
2. Дані санскритські дієслівні форми та їх переклади на російську мову, які записані у іншому порядку.

nayasi, icchati, anayam, nayâmi, icchasi, icchâmi, anayat

я хочу, ты ведешь, он хочет, я веду, я вел, ты хочешь, он вел

Створити продукційну систему, що вірно їх перекладає [24].

РОЗДІЛ VI. МЕТОДИ ПРЕДСТАВЛЕННЯ ТА ОБРОБКИ ЗНАНЬ СКЛАДНОСТРУКТУРОВАНИХ ПРОБЛЕМНИХ ОБЛАСТЕЙ

Однією з ознак складної системи є ієрархічність. Ця ознака особливо істотна для сучасних предметних областей (ПДО), при описі яких часто використовується термін «складноструктурованість». Для них характерна наявність величезного числа систем, пов'язаних одна з одною інформаційними зв'язками які складаються в свою чергу з підсистем, і т. д. до елементів.

Слід зазначити, що використання механізмів виведення в складноструктурованих ПДО є найбільш важливим тому що дозволяє:

- «стиснути» екстенсіональну складову бази знань та даних (БЗД), тобто отримати факти на основі логічних виразів (аксіом), що містяться в інтенсіональній складовій БЗД;
- здійснити перевірку інформації, що міститься в БЗД, на несуперечність;
- поповнювати новими знаннями інтенсіональну складову БЗД.

У даному посібнику ми розглянемо дві сучасні моделі представлення знань складноструктурованих ПДО – онтологічну та багаторівневу логіку (Multi-layer logic – MLL).

6.1 Онтологічні моделі представлення багаторівневих концептуальних знань

У штучному інтелекті онтології використовуються для формальної специфікації понять і відносин, які характеризують певну галузь знань. Оскільки комп'ютер не може розуміти, як людина, стан речей у світі, йому необхідно подання всієї інформації в формальному вигляді. Таким чином, онтології служать своєрідною моделлю навколишнього світу, а їх структура така, що легко піддається машинній обробці і аналізу. Онтології дають інтелектуальній системі інформацію про описану семантику заданих слів і вказують ієрархічну будову області, взаємозв'язок елементів. Все це дозволяє комп'ютерним програмам за допомогою онтологій робити логічний вивід з представлених знань та маніпулювати ними.

У філософському сенсі онтологію можна уявляти як певну систему категорій, які є наслідком певного погляду на світ. В

сучасних інформаційних технологіях найбільш часто згадується і використовується визначення онтології, сформульоване Н. Грубером: «Онтологія – це специфікація концептуалізації» [25]. Ця дефініція є своєрідним узагальненням, формальною інтерпретацією багатьох інших визначень. Центральним у ньому є поняття «концептуалізація», яке було введено в роботі [26]. Концептуалізація передбачає опис багатьох об'єктів і понять, знань про них та зв'язків між ними. Таким чином, формально онтологія складається з термінів, організованих в таксономію, їх визначень і атрибутів, а також пов'язаних з ними аксіом і правил виводу.

Часто набір припущень, що становлять онтологію, має форму логічної теорії першого порядку, де терміни словника є іменами унарних і бінарних предикатів, які називаються відповідно концептами і відносинами. У простому випадку онтологія описує тільки ієрархію концептів, пов'язаних відносинами категоризації. У складніших випадках у неї додаються відповідні аксіоми для вираження інших відносин між концептами і для того, щоб обмежити їх передбачувану інтерпретацію. Враховуючи вищезазначене, онтологія являє собою БЗД, що описує факти, які передбачаються завжди істинними в рамках певної галузі знань на основі загальноприйнятого сенсу словника, що використовується.

Побудову онтологічної моделі представлення багаторівневих концептуальних знань розглянемо на прикладі розробки моделі концептуальних знань для області «Ергономічне забезпечення проектування ерготехнічних систем» у вигляді трирівневої онтології, яка дозволяє задати точну специфікацію концептуалізації даної області, забезпечити узгодження та інтеграцію знань із кількох предметних областей.

6.2 Онтологічна модель знань для проблемної області «Ергономічне забезпечення проектування ерготехнічних систем»

Поняття «ерготехнічна система» було розглянуто раніше при вивченні курсу «Людино-машинні інтерфейси». Нагадаємо, що ерготехнічні системи це підклас гуманістичних систем (систем, у складі яких є людина), в яких метою діяльності фахівця (трудового колективу) є отримання продукту праці. Різновидом ерготехнічних систем є автоматизовані системи керування технологічними процесами (АСК ТП).

Широке впровадження автоматизованих і інтелектуальних систем в сучасних системах керування технологічними процесами, приводить до підвищення інформаційної насиченості контурів управління, які замикаються на оператора, що приводить до зростання технологічної складності ухвалення рішень. Розвиток інформаційних технологій, підвищення ступеня автоматизації і перерозподіл функцій між людиною і технікою загострило проблему взаємодії людини-оператора з системою керування.

Для сучасних АСК ТП характерно те, що людина все більше віддаляється від об'єктів контролю та керування та здійснює свої керуючі функції дистанційно за допомогою інформаційної моделі − мнемосхем, карт, графіків, зображень тощо. Інформаційна модель – «це організована за певними правилами сукупність, інформації про стан і функціонування об'єкта керування і зовнішнього середовища. Інформаційна модель є для людини-оператора джерелом інформації, на підставі якої він формує образ реального стану об'єкта керування, проводить аналіз і оцінку ситуації, що склалася, приймає рішення, планує керуючі дії і оцінює результати їх реалізації» [27].

У зв'язку з цим, актуальною є задача ергономічного забезпечення проектування інформаційних моделей операторів технологічних процесів людино-машинних систем різних рівнів (диспетчерів, машиністів і т. д.).

У відповідність з відомими методиками проектування інформаційних моделей, наприклад [27], обов'язковим є опис і формалізація знань про наступні предметні області: ПДО «Технологічний процес і об'єкти контролю і керування» і, власне, знання про ПДО «Ергономічне забезпечення проектування інформаційних моделей».

Необхідність вирішення задач керування технологічними процесами, в яких потрібно враховувати зміни, що відбуваються у навколишньому середовищі під час роботи системи керування, обумовлює динамічність моделі знань. Згідно [28], для динамічної експертної системи необхідні знання про методи взаємодії із зовнішнім оточенням та знання про модель зовнішнього світу. Тобто, у процесі ухвалення рішень оператор використовує знання асоційовані не лише з даною ПДО, але і знання вищої міри спільності – метазнання (описи властивостей часу, простору, особистості і т. д.). У зв'язку з цим, виникає задача розробки моделі метазнань

(концептуальних знань), семантично об'єднуючої дві вищезгадані ПДО, тобто розробка формальної моделі концептуальних знань для бази знань та даних експертної системи «Ергономічне забезпечення проектування інформаційних моделей».

Метазнаннями для ПДО «Ергономічне забезпечення проектування інформаційних моделей» є знання про область «Ергономічне забезпечення проектування ерготехнічних систем», формальна модель якої розроблена в [29]. Для ПДО «Технологічний процес і об'єкти контролю та керування» метазнаннями можна вважати узагальнену модель бази знань і даних інформаційно-управляючих людино-машинних систем, наведену в [27]. Таким чином, ПДО «Ергономічне забезпечення проектування інформаційних моделей» і ПДО «Технологічний процес і об'єкти контролю та керування» пов'язані відношенням Part-of-ієрархії (*POF*) – «частина-ціле» з вищезгаданими областями відповідно.

У зв'язку з цим, ставиться задача розробки на базі [27, 29] однієї об'єднуючої моделі метазнань – «Концептуальні знання про ергономічне забезпечення проектування ерготехнічних систем» (далі «КЗ»). Область «КЗ» належить до складноструктурованих областей знань, для яких характерна наявність великої кількості систем, зв'язаних одна з одною інформаційними зв'язками різних типів. Згідно [19] найбільш повно різні типи зв'язків реалізовані в моделях типа семантичної мережі.

Різновидом мережевої моделі представлення знань на метарівні є онтологія [20], яка представляється трійкою

$$O = <K, R, F>, \qquad (6.1)$$

де K – кінцева множина сутностей області «КЗ», яку представляє онтологія; R – кінцева множина відношень між концептами; F – кінцева множина функцій інтерпретації, заданих на сутностях та (або) відношеннях.

Для формалізації простору знань, що охоплюють ПДО «Ергономічне забезпечення проектування інформаційних моделей» і ПДО «Технологічний процес і об'єкти контролю та керування», розроблена динамічна модель розширеної онтології *OR* опису представлення «КЗ» в базі знань і даних

$$OR = <OKZ, \{OKP, OZ\}, LV>, \qquad (6.2)$$

де *OKZ* – онтологія області «КЗ»; *OKP* – онтологія кожної окремої ПДО; *OZ* – онтологія задач кожної ПДО; *LV* – механізм логічного виводу.

Метаонтологія *OKZ* задає ієрархію сутностей і відношень між ними, які не залежать від конкретної ПДО. Були виділені наступні концепти метаонтології першого і другого рівня (наведені в дужках) з їх подальшою деталізацією: час (дата, момент); простір (реальний, віртуальний); об'єкти (живі, неживі, концептуальні); взаємозв'язки (дії, властивості, імплікації); операції; стани (етапи, режими, події, ситуації) і допоміжні концепти (одиниці виміру, логічні операції, параметри, шкали і значення) наведені на рис. 6.1.

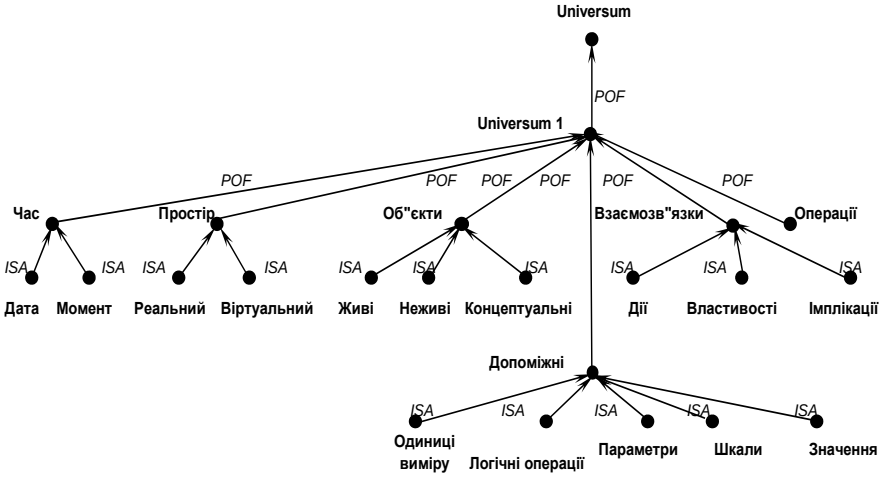


Рисунок 6.1 – Метаонтологія концептуальних знань

Базу знань та даних експертної системи «Ергономічне забезпечення проектування інформаційних моделей» позначимо як універсальний клас *UNIVERSUM* (*U*). Для області знань «КЗ» введемо підклас *UNIVERSUM1* (*U1*). Очевидно, що клас *U1* зв'язаний з класом *U* відношенням *POF*. Цим же відношенням зв'язані з класом *U1* концепти метаонтології першого рівня. Концепти другого рівня зв'язані з відповідними концептами першого рівня відношеннями класифікації *ISA* – «об'єкт *X* є елементом множини *X*».

Предметна онтологія *OKP* містить поняття, що описують окремі ПДО «Технологічний процес і об'єкти контролю та керування» та «Ергономічне забезпечення проектування інформаційних моделей» і семантично значимі для них відношення, а також декларативні та процедурні інтерпретації цих понять і відношень. Приклад представлення смислового змісту висловлювань «У людини є слуховий аналізатор. Цей аналізатор має деякі властивості» у вигляді ієрархічної семантичної мережі наведено на рис.6.2.

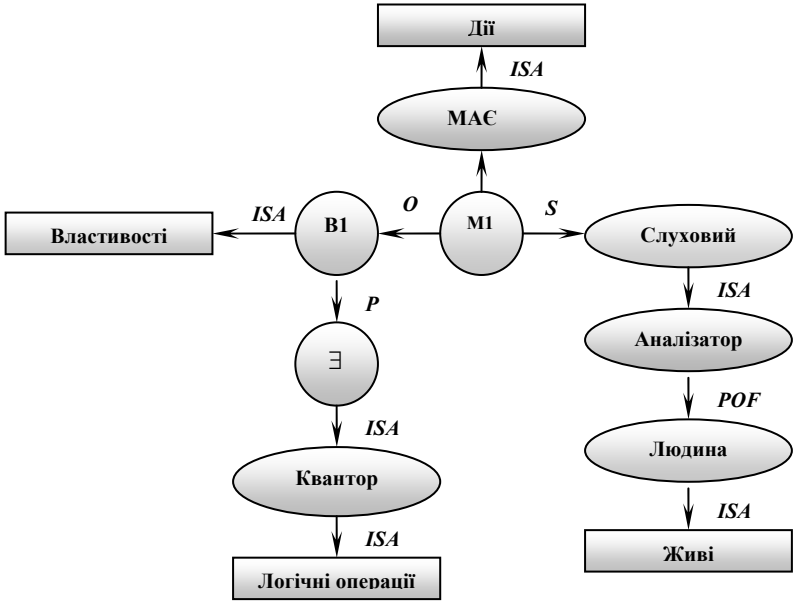


Рисунок 6.2 – Фрагмент семантичної мережі предметної онтології

У цьому фрагменті семантичної мережі прямокутниками позначені вершини, що належать метаонтологии *OKZ*, а колами позначені актанти та предикативні слова, що входять у дані висловлювання. Вершини мережі зв'язані відношеннями *ISA* та відмінковими відношеннями (*P* – «параметр», *O* – «об'єкт висловлювання», *S* – «суб'єкт висловлювання») з розширеного в [30] набору відмінків К. Філмора.

У онтології задач *OZ* у якості понять виступають типи задач, що вирішуються проектувальником інформаційних моделей.

Їх номенклатура була визначена за результатами системного аналізу діяльності проектувальника [31].

Логічний вивід *LV* починається при активації початкових умов у вигляді понять або відношень, що описують вихідну ситуацію. Останов процедури виводу відбувається або при досягненні цільової ситуації, або при перевищенні тривалості часу, відведеного для вирішення задачі.

Таким чином розроблена модель організації знань «КЗ» у вигляді трьох рівнів онтології: загального (верхнього), предметного і задачного. Така ієрархічна мережева модель представлення знань дозволяє: задати точну специфікацію концептуалізації даної області; забезпечити узгодження і інтеграцію знань з декількох ПДО, а також інтелектуальну підтримку проектувальника інформаційних моделей, шляхом логічного виводу відповідних рекомендацій щодо поточної проблемної ситуації.

6.3 Багаторівнева логіка – мова представлення складноструктурованих знань

Всі відомі механізми логічного виводу, в силу спільності, можуть бути застосовані до складноструктурованих ПДО. Але ефективність їх використання в таких ПДО різко зменшується за рахунок того, що для опису складноструктурованих ПДО використовується величезна кількість структурних залежностей, для формалізації яких в численні предикатів першого порядку немає інших засобів, крім предикатів. Використання предикатів для завдання структурних залежностей загромаджує опис ПДО, знижує його наочність і різко зменшує ефективність процедур виведення [19]. Тому в даний час вельми актуально створення мов представлення знань і механізмів виводу в них, що дозволяють описувати структурні відношення, не вдаючись для цього до предикатів.

Однією з таких мов подання знань є багаторівнева логіка MLL, створена японськими вченими *С.* Осуга і *Х.* Ямаучи, які також розробили і механізми виводу в ній.

MLL можна розглядати як інтеграцію логічного підходу та підходу, заснованого на семантичній мережі, до побудови мови представлення знань. Розглянемо її опис і модифікацію, що дозволяє збільшити ефективність процедури дедуктивного виводу.

Для формального опису складноструктурованих ПДО базовими являються відношення *ISA* і *POF*. Для їх представлення в MLL використовується ієрархічна абстракція і ієрархічна структура. Ієрархічна абстракція представляє граф, вершинам якого відповідають класи об'єктів або їх представники, а ребрам – відношення «клас-підклас», або «частина – ціле». Класи і їх представники розташовуються на різних рівнях і виділяються відповідно до принципу спадкоємства властивостей. Механізм спадкоємства властивостей дозволяє стиснути базу знань та даних, тобто зробити її компактнішою. У ієрархічній абстракції, що відповідає відношенню *POF*, рівні виділяються відповідно до принципу декомпозиції, який є основоположним при моделюванні складноструктурованих ПДО.

Атрибути класів об'єктів або їх представників (об'єктів) і відношення між класами об'єктів (за винятком структурних відношень), в ієрархічній абстракції можуть бути описані правильно побудованими формулами. Правильно побудована формула, що описує клас об'єктів (чи об'єкти) в ієрархічній абстракції, з'єднується з вершиною, що відповідає йому. Приклад правильно побудованої формули, що описує об'єкти в ієрархічній абстракції, наведений на рис. 6.3, де # – позначення константи; $\rightarrow$ – позначення структурних відношень; ---> – позначення посилання на правильно побудовану формулу.

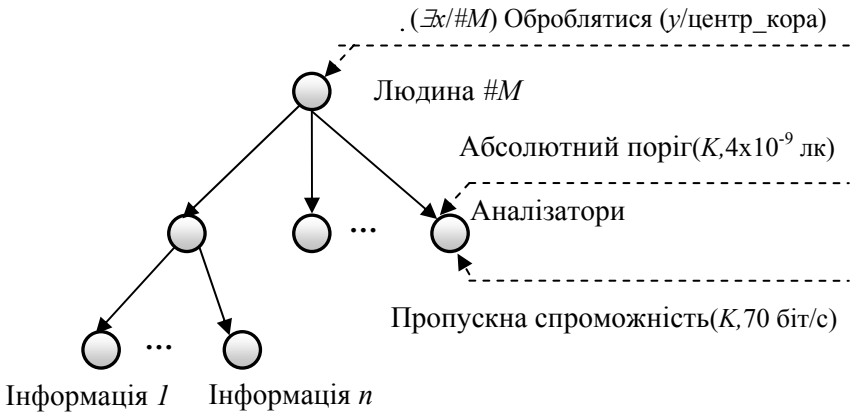


Рисунок 6.3 – Фрагмент моделі предметної області

Крім того, будь-яка правильно побудована формула, яка задає деякий опис ПДО, повинна використовувати в якості термів класи об'єктів (чи об'єкти), з якими вона з'єднана в ієрархічній абстракції. Ці класи об'єктів (чи об'єкти) розташовані в ієрархічній абстракції на нижчих рівнях від вершини, якій відповідає опис. Приклад такої правильно побудованої формули, що описує вершину «Людина», пов'язаної з класом об'єктів «Аналізатори», розташованому на нижчому рівні ієрархії, наведений на рис. 6.3: «Існує деякий аналізатор людини *#M* такий, що уся інформація, що приймається ним, обробляється центром в корі головного мозку».

Розглянемо базові відношення, які задаються в MLL [19].

1. «Element of» (*EOF*), яке позначається $x \in X$ (об'єкт X є елементом множини X) і є засобом для завдання відношення «елемент-множина».

2. «Power set of», яке позначається $Y=*X$ і використовується для відображення того, що Y є множиною, що складається з підмножин множини X (не включаючи порожньої множини). Наприклад, якщо $d=\{1,2,3\}$, то $*d = \{\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\}, \{1,2,3\}\}$.

3. «Product set of», яке позначається $Y= X_1 \times X_2 \times ... \times X_n$ або $Y = \prod_{i=1}^{n} Xi$ і використовується для завдання того, що Y є декартовий добуток з $X_1 \times X_2 x ... \times X_n$.

4. «Component of» (COF), яке позначається $Y \Delta X$ і задає, що X є компонентом Y. Якщо Y містить декілька компонент, тобто $Y \Delta X_i$, $Y \Delta X_2$,..., $Y \Delta Xs$, то спеціальний метасимвол < > використовується для позначення: $<Y>=\{X_1, X_2, ..., X_s\}$. Змістовно, X_i, $i = 1,..., s$ являє собою підмножину елементів одного сорту, кожен з яких є частиною Y.

Шляхом композиції базисних відношень визначаються наступні похідні відношення:

- «Subset of», яке позначається $X \subseteq Y$ і визначає, що X є підмножиною Y. Це відношення є композицією двох: $Z=*Y$ та $X \in Z$, тобто $\subseteq = \in \circ *$ і задає ISA-ієрархію на множині класів об'єктів ПДО;
- POF, яке позначається $Y \blacktriangleright x$ і визначає, що x є часткою Y. Це відношення є композицією двох відношень: $\blacktriangleright = \in \circ \Delta$ і використовується для завдання Part-of-ієрархії на множині класів об'єктів ПДО. Графічне представлення відношення *POF* наведено на рис. 6.4.

Ієрархічна структура, яка зображена на рис. 6.4, є багаторівневою. Класи об'єктів і їх представники, які знаходяться у

відношенні *EOF*, розташовуються на різних рівнях, а класи об'єктів, які знаходяться у відношенні *COF*, – на одному рівні, оскільки між ними відсутнє спадкоємство властивостей.

Для завдання ієрархічної абстракції, відповідної *ISA*-ієрархії, MLL використовує розширення цього формалізму. Так, MLL дозволяє працювати з сортами, що є структурованими одиницями, елементами доменів яких можуть бути множини, безліч підмножин і т. д.

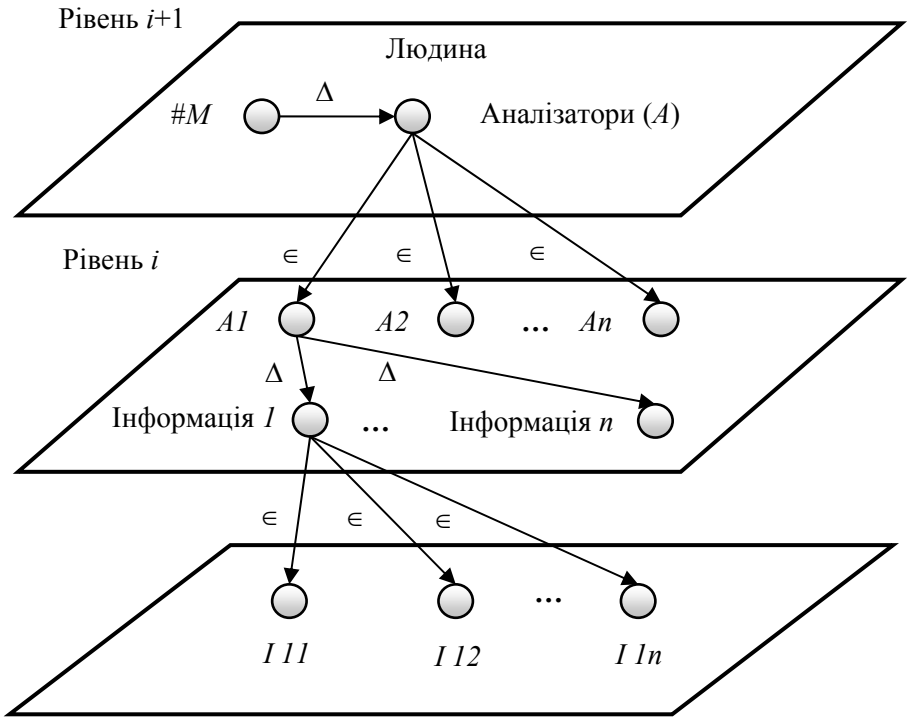


Рисунок 6.4 – Приклад представлення ієрархічної структури в MML

З метою спрощення опису ПДО і підвищення ефективності дедуктивного виведення, для опису *POF*-відношень замість предикатів в MLL введено три види спеціальних знаків слешів.

Простий слеш (*Qx/X*) використовується для позначення того, що x є елементом множини X ($x \in X$), зворотний слеш (*Qx\X*) позначає, що

x визначений на множині, елементами якої є компоненти об'єкту X($X\Delta x$), подвійний слеш (*Qx//X*) позначає, що x визначений на множині, елементами якої є частини об'єкту $X(X \blacktriangleright x)$, де $Q \in \{\forall, \exists\}$.

Якщо об'єкт Y має в якості компоненти об'єкт X, то для того, щоб визначити властивості об'єкту $x \in X$, який є частиною об'єкту Y, можна написати формулу

$$(Qx//\#Y)\ G(x), \tag{6.3}$$

де $G(x)$ – предикат, що описує властивості об'єкту x (атрибут).

Формула (6.3) записана в стандартній для MLL формі. Об'єкт Y може мати в якості компонент декілька об'єктів (рис. 6.4). Для того, щоб задати потрібну нам компоненту X об'єкту Y, необхідно використати селектор, який представляється предикатом $F(X, Y)$. Тоді, щоб визначити властивості $x \in X$, об'єкту, що є частиною об'єкту Y, можна написати формулу

$$(\exists X/\#Y)\ (Qx/X)\ [F(X, Y)\&G(x)], \tag{6.4}$$

яка може бути перетворена до стандартної форми:

$$(Qx//\#Y)[F(x, Y)\&G(x)]. \tag{6.5}$$

Використовуючи формалізм завдання структурних відношень в префіксі формули, в [19] запропонована заміна селектору, який представляється предикатом і використовується для знаходження потрібної компоненти деякого об'єкту, на композицію префікса. Розглянемо це розширення синтаксису MLL.

Нехай об'єкт $\#Y$ має в якості компонент декілька об'єктів, тобто $<\#Y>= \{X_i\}$, $i = 1... n$. Тоді префікс може містити запис виду

$$(Q(x/X_i)//\#Y), \tag{6.6}$$

де X_i задає сортність x, якій в прикладному численні відповідає клас об'єктів ПДО. Формула (6.2) змістовно означає, що x визначено на множині частин $\#Y$, які мають сортність X_i.

Виходячи з цього, модифіковані правила утворення правильно побудованих формул (ППФ) в MML наступні:

F1. Якщо P є n-місним предикатним символом і t_1, t_2, …, t_n є терми, то $P(t_1, t_2,..., t_n)$ теж ППФ.

F2. Якщо F і G – ППФ, то $\neg(F)$, $(F\&G)$, $(F \vee G)$, $(F \rightarrow G)$ теж ППФ.

F3. Якщо F – ППФ і x – предметна змінна, то:

– *(∀x/y) F* і *(∃x/y) F* є ППФ, де *y* – константа або змінна;

– *(∀x//y) F* і *(∃x//y) F* є ППФ, де *y* – константа або змінна;

– *(∀(x/Z)//y) F* і *(∃(x/Z)//y) F* є ППФ, де *y* – константа або змінна, а *Z* є константна множина.

F4. Інших правил утворення ППФ немає.

Переваги модифікованих правил побудови правильно побудованих формул:

• не витрачається час на уніфікацію предиката-селектора, а розширений синтаксис префікса логічної формули дозволяє зробити означення термів по структурах ПДО і далі здійснити перевірку значень змінних на задоволення умові, що задається матрицею логічної формули;

• не витрачається пам'ять на зберігання численних фактів ПДО для означення змінних в предикаті-селекторі.

З розглянутого вище виходить, що це розширення синтаксису MLL підвищує ефективність дедуктивного виведення по пам'яті і швидкодії.

Розглянемо приклад логічного виразу, що міститься в інтенсіональній складовій бази знань та даних, і на ньому покажемо переваги використання апарату MLL.

Приклад 6.1. Програмна компонента *x*, що входить до складу експертної системи «Ергономічне забезпечення проектування інформаційних моделей» *#ES*, забезпечує інтелектуальну підтримку проектування інформаційних моделей *y*, для пульта управління технологічним процесом оператора *#O*, якщо є:

• ЕОМ *t*, на якій функціонує *x*;

• багатофункціональна панель *m* пульта управління технологічним процесом оператора *#O*, на якому відображається інформаційна модель *y* і пов'язана з ЕОМ *t*;

• потік інформації *#I0*, що містить потік повідомлень t_0, який виробляється компонентою «Інтелектуальний інтерфейс» *#II* експертної системи «Ергономічне забезпечення проектування інформаційних моделей» *#ES* і що містить клас повідомлень r_0 (наприклад, про задачі, що вирішуються оператором *#O*), який оброблюється компонентою *x*, для проектування інформаційних моделей *y*;

• потік інформації #I0, що містить потік повідомлень t_{01}, який виробляється компонентою *#II* експертної системи «Ергономічне забезпечення проектування інформаційних моделей» *#ES* і що містить клас повідомлень r_{01} (наприклад про умови, в яких працює оператор *#O*), який оброблюється компонентою *x*, для проектування інформаційних моделей *y*;

• потік інформації *#I1*, що містить потік повідомлень t_1, який виробляється компонентою «Логічний вивід» *#LV* експертної системи «Ергономічне забезпечення проектування інформаційних моделей» *#ES* і що містить клас повідомлень r_1 у вигляді рекомендацій проектування ІМ *y*, який оброблюється компонентою *x*.

Запис в MLL:

(∃(*x*/компонента*)//#ES*)(∀(*y*/*IM)//#OB*)

(∃(*t*/ПЕОМ)//*#O*)(∃(*m*/Панель)//*#O*) (∃(t_0/потік_повідомлень)//*#I0*) (∃(r_0/клас_повідомлень)//t_0)

(∃(t_{01}/потік_повідомлень)//*#I0*) (∃(r_{01}/клас_повідомлень)//t_{01})

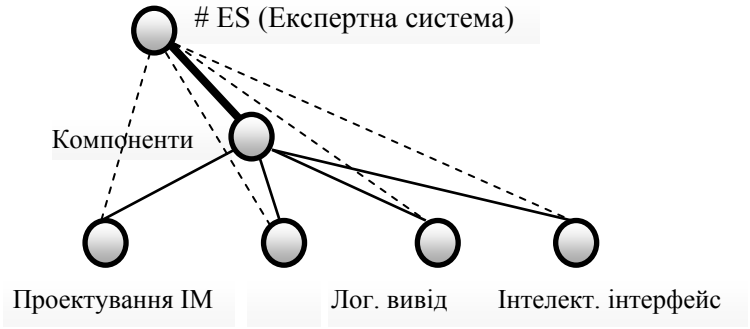
(∃(t_1/потік_повідомлень)//*#I1*) (∃(r_1/клас_повідомлень)//t_1)

Функціонує(*x*, *t*) & Відображає(*y*, *m*) & Пов'язаний(*m*, *t*) & Виробляє(t_0, *x*) & Обробляє(*x*, r_0) & Виробляє(t_{01}, *x*)& Обробляє(*x*, r_{01}) & Виробляє(t_1, *x*) & Обробляє(*x*, r_1) → Забезпечує_підтримку_проектування(*x*, *y)*

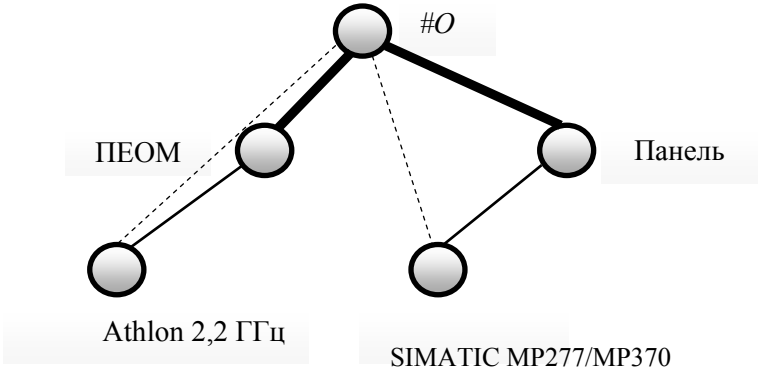
Структури класів «Експертна система» (*#ES*), «Пульт управління» (*#O*), «Інформаційна модель (ІМ)» (*#OB*), потоків інформації *#I0* та *#I1* предметної області, що розглядається, надані на рис. 6.5, 6.6, 6.7, 6.8, 6.9 відповідно.

Таким чином, MLL дозволяє зробити опис складноструктурованих ПДО більш наочним за рахунок завдання структурних відношень в префіксі логічної формули, без використання предикатів, і, як наслідок цього, дозволяє збільшити ефективність дедуктивного виведення за рахунок істотного скорочення простору пошуку.

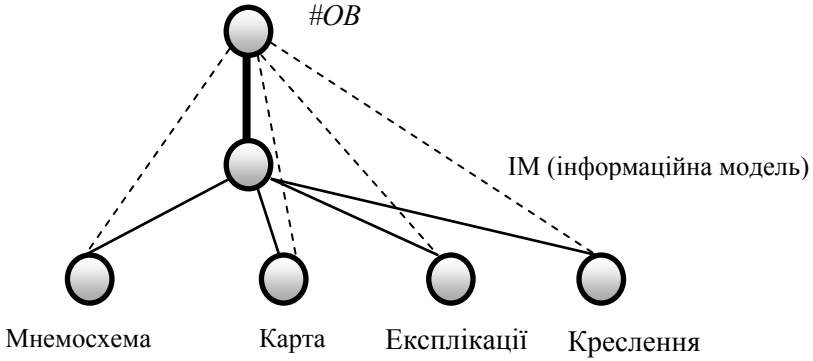
У MLL в процесі дедуктивного виведення використовуються два алгоритми: алгоритм сколемізації і алгоритм уніфікації. Ці алгоритми в MLL є подальшим розвитком відповідних алгоритмів числення предикатів першого порядку.



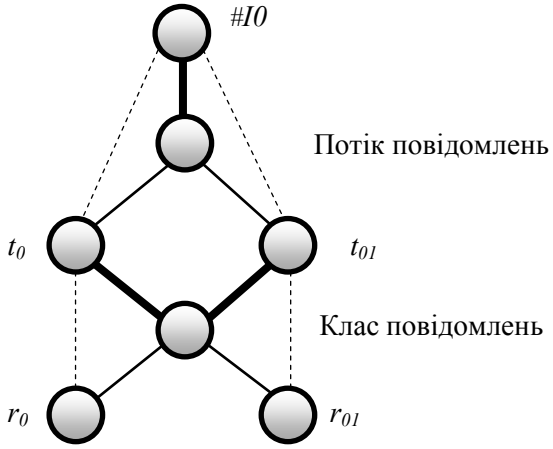
а) Структура «Експертна система» (*#ES*)



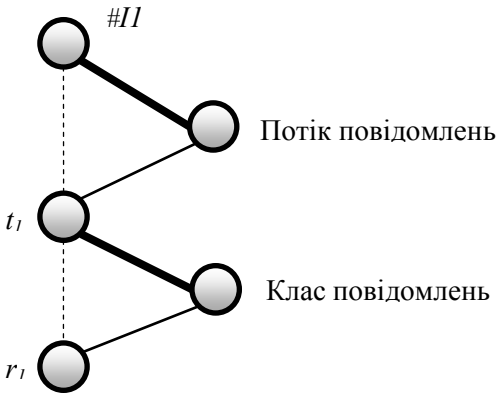
б) Структура "Пульт управління" (*#O*)



в) Структура "Інформаційна модель" (*#OB*)



г) Структура "Потік повідомлень" *#I0*



д) Структура «Потік повідомлень» *#I1*

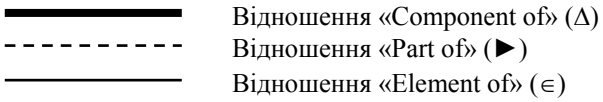


Рисунок 6.5 – Структури даної ПДО

Резюме

Однією з ознак сучасних предметних областей є складноструктурованість. Було розглянуто дві моделі представлення знань складноструктурованих ПдО – *онтологічну* та *багаторівневу логіку* – MLL.

Онтології дають інтелектуальній системі інформацію про описану семантику заданих слів і вказують ієрархічну будову предметної області, взаємозв'язок елементів. Онтологія являє собою БЗД, що описує факти, які передбачаються завжди істинними в рамках певної галузі знань на основі загальноприйнятого сенсу словника, що використовується. Використання онтологій в інтелектуальних системах дозволяє: задати точну специфікацію концептуалізації даної області; забезпечити узгодження і інтеграцію знань з декількох ПдО; здійснювати інтелектуальну підтримку осіб, що приймають рішення, шляхом логічного виводу відповідних рекомендацій щодо поточної проблемної ситуації.

Багаторівневу логіку MLL можна розглядати як інтеграцію логічного підходу та підходу, заснованого на семантичній мережі, до побудови мови представлення знань.

Для представлення базових відношень ISA і POF в MLL використовується ієрархічна абстракція і ієрархічна структура.

Багаторівнева логіка MLL дозволяє зробити опис складноструктурованих предметних областей більш наочним за рахунок завдання структурних відношень в префіксі логічної формули, без використання предикатів, і, як наслідок цього, дозволяє збільшити ефективність дедуктивного виведення за рахунок істотного скорочення простору пошуку.

У багаторівневій логіці MLL в процесі дедуктивного виведення використовуються два алгоритми: алгоритм сколемізації і алгоритм уніфікації.

Контрольні питання

1 Що дозволяє використання механізмів виведення в складноструктурованих ПдО?

2 Чим є онтологія у філософському сенсі?

3 Поясніть термін «онтологія» с точки зору сучасних інформаційних технологій?

4 Що таке інформаційна модель?

5 Які знання є метазнаннями для предметної області «Ергономічне забезпечення проектування інформаційних моделей»?

6 Як формально задати онтологію?

7 Коли починається логічний вивід у онтологічній моделі?

8 Які недоліки використання предикатів для опису складноструктурованих ПдО?

9 Які відношення є базовими для опису складноструктурованих ПДО?

10 Наведіть приклад правильно побудованої формули, що описує об'єкти в ієрархічній абстракції.

11 Які базові відношення задаються в MLL?

12 Які відношення задаються в MLL шляхом композиції базисних відношень?

13 Що введено в MLL для спрощення опису предметної області?

14 Які модифіковані правила утворення правильно побудованих формул в MML ви знаєте? У чому їх перевага?

15 Які алгоритми використовуються в MLL у процесі дедуктивного виведення?

ПЕРЕЛІК ПОСИЛАНЬ

1. Соревнования MINOS 07 Micromouse competition [Електрон. ресурс]. – 2007. – Режим доступу: http://myrobot. ru/news/2007/05/20070506_1. php

2. Пильщиков В. Н. Библиотечка программиста: Язык Плэнер / В. Н. Пильщиков. – М. : Наука, 1983. – 208 с.

3. Клоксин У., Программирование на языке Пролог / Клоксин У., Меллиш К. – М. : Мир, 1987. – 336с.

4. Адаменко А. Логическое программирование и Visual Prolog / Анатолий Адаменко, Андрей Кучуков. – СПб. : БХВ-Петербург, 2003. – 990с.

5. Керн Г. Энциклопедии: Архитектура. Скульптура, Тайны, сенсации, факты, катастрофы / Г. Керн. – М. : Азбука, 2007. – 432с.

6. Гарднер М. Математические головоломки и развлечения [пер. с англ.] / Мартин Гарднер. – 2-е изд., испр. и доп. – М. : Мир, 1999. – 447с.

7. Лорьер Ж. Л. Системы искусственного интеллекта / Ж. Л. Лорьер. – М. : Мир, 1991. – 568 с.

8. Попов Э. В. Общение с ЭВМ на естественном языке / Э. В. Попов. – 2 -е изд. – М. : Финансы и статистика, 2004. – 360 с.

9. Шенк Р. Скрипты, планы и знание / Р. Шенк, Р. Абельсон // Труды IV межд. конф. по искусств, интеллекту в 6-ти т., М. : Научн. совет по компл. пробл. «Кибернетика» АН СССР, 1975. – Т.6. – С. 208 – 220.

10. Шенк Р. Обработка концептуальной информации: пер. с англ. / Р. Шенк. – М. : Энергия, 1982. – 279 с.

11. Апресян Ю. Д. Лингвистический процессор для сложных информационных систем / Ю. Д. Апресян, И. М. Богуславский, Л. Л. Иомдин и др. – М. : Наука, 1992. – 256с.

12. Дэвидсон Д. Что означают метафоры / Д. Дэвидсон. – М. : Прогресс, 1990. – 234 с.

13. Марселлус Д. Программирование экспертных систем на Турбо-Прологе / Д. Марселлус. – М. : Финансы и статистика, 1994. – 254 с.

14. Журавлев А. П. Язык и компьютер / А. П. Журавлев, Н. А. Павлюк. – М. : Просвещение, 1989. – 159 с.

15. Хомский Н. Синтаксические структуры / Н. Хомский. – М. : Мир, 1956. – 526с.

16. Маннинг К. Д. Введение в информационный поиск: пер. с англ. / К. Д. Маннинг, П. Рагхаван, Х– М. Шютце. ООО «Вильямс», 2011. – 528 с.: ил.

17. Хомский Н. Введение в формальный анализ / Н. Хомский, Дж. Миллер. – К. : Едиториал УРСС, 2003. – 290 с.

18. Хомский Н. Аспекты теории синтаксиса / Н. Хомский. – М.: МГУ, 1972. – 259 с.

19. Вагин В. Н. Достоверный и правдоподобный вывод в интеллектуальных системах / В. Н. Вагин, Е. Ю. Головина, А. А. Загорянская, М. В. Фомина; под. ред. В. Н. Вагина, Д. А. Поспелова. – М. : ФИЗМАТЛИТ, 2004. – 704 с.

20. Гаврилова Т. А. Базы знаний интеллектуальных систем / Т. А. Гаврилова, В. Ф. Хорошевский. – СПб.: Питер, 2000. – 384 с.

21. Ин Ц. Использование Турбо-Пролога / Ц. Ин, Д. Соломон. – М. : Мир, 1993. – 608 с.

22. Приобретение знаний / под редакцией С. Осуги, Ю. Саєки. М. : Мир, 1990. – 304 с.

23. Поспелов Д. А. Моделирование рассуждений. Опыт анализа мыслительных актов / Д. А. Поспелов. – М. : Радио и связь, 1989. – 184 с.

24. Алпатов В. М. Лингвистические задачи: пособие для учащихся/ В. М. Алпатов и др. – М. : Просвещение, 1983. – 223 с.

25. Gruber Th. What is an Ontology [Електрон. Ресурс]/ Th. Gruber. – Режим доступу: http://www-ksl. stanford. edu/kst/what-is-an-ontology. html

26. Genesereth M. R. Logical Foundation of Artificial Intelligence / M. R. Genesereth, N. J. Nilsson. Los Altos, California : Morgan Kaufmann Publishers, 1987. – 405 с.

27. Информационно-управляющие человеко-машинные системы: исследование, проектирование, испытания. Справочник /

А. Н. Адаменко, А. Т. Ашеров, И. Л. Бердников и др.; под общ. ред. А. И. Губинского и В. Г. Евграфова. – М. : Машиностроение, 1993. – 528 с.

28. Попов Э. В. Статические и динамические экспертные системы / Э. В. Попов, И. Б. Фоминых, Е. В. Кисель, М. Д. Шапот. – М. : Финансы и статистика, 1996. – 320с.

29. Сердюк С. Н. Разработка метода интеллектуальной поддержки процесса эргономического проектирования информационных моделей: дис. … канд. техн. наук: 05.02.20 / С. Н. Сердюк. – СПб. : СПб ЭТУ, 1993. – 364 с.

30. Сокирко А. В. Семантические словари в автоматической обработке текста (По материалам системы ДИАЛИНГ): дис. … канд. техн. наук: 05.13.17 / А. В. Сокирко. – М. : РГГУ, 2001. – 120 с.

31. Камінська Ж. К. Аналіз проблем автоматизації процесу ергономічного проектування інформаційних моделей технологічних процесів. / Ж. К. Камінська // Вісник Житомирського державного технологічного університету. Технічні науки. Випуск 1 (52), Житомир: ЖДТУ, 2010. – С.103–108.

АЛФАВІТНО-ПРЕДМЕТНИЙ ПОКАЗЧИК

ПЕРЕЛІК СКОРОЧЕНЬ

DB2 – родина систем керування реляційними базами даних корпорації IBM.

ЕОМ – електронно – обчислювальна машина

FTP – (File Transfer Protocol) – протокол передавання файлів

HTML – (Hyper Text Markup Language) – мова розмітки гіпертексту

HTTP – (Hyper Text Transfer Protocol) – «протокол передавання гіпертексту»

IEC 13211-1 – (International Electro technical Commission Standard) – Міжнародна електротехнічна комісія стандартів; стандарт основних елементів мови Prolog

IEC 13211-2 – (International Electro technical Commission Standard) – Міжнародна електротехнічна комісія стандартів; стандарт підтримки модульного програмування мови Prolog

ISO (International Organization for Standardization) – Міжнародна організація зі стандартизації

OCI – (Oracle Call Interface) – набір функцій інтерфейсу, що дозволяють маніпулювати об'єктами ORACLE сервера

ODBC – (Open Database Connectivity) – програмний інтерфейс (API) доступу до баз даних

Prolog – (logic programming) – мова програмування; «програмування в термінах логіки»

SQL – (Structured Query Language) – «мова структурованих запитів»

TCP/IP (Transmission Control Protocol/Internet Protocol) – протокол керування передавання; стек протоколів

БНФ – мета мова Бекуса-Наура

ЗНТУ – Запорізький національний технічний університет

ІС – інтелектуальна система

ОС – операційна система

ПДО – предметна область

Рис. – рисунок

ТКЗ – теорія концептуальної залежності

Навчальне видання

ДЕЙНЕГА Лариса Юріївна
КАМІНСЬКА Жанна Костянтинівна
ЛЕВАДА Ірина Василівна
СЕРДЮК Сергій Микитович

ПРАКТИЧНЕ ПРОГРАМУВАННЯ МОВОЮ VISUAL PROLOG

Навчальний посібник

Обкладинка *Чепіга Н. К.*
Комп'ютерний набір: *Левада І. В.*
Дейнега Л. Ю.
Верстання *Гринь Д. В.*

Оригінал-макет підготовлено
в редакційно-видавничому відділі ЗНТУ

Підписано до друку 15.05.2015. Формат 60×84/16. Ум. друк. арк. 13,78.
Тираж 300 прим. Зам. № 523

Запорізький національний технічний університет
Україна, 69063, м. Запоріжжя, вул. Жуковського, 64
Тел.: (061) 769–82–96, 220–12–14

Свідоцтво суб'єкта видавничої справи ДК № 2394 від 27.12.2005.