ЗМІСТ

ВСТУП

Навчальна дисципліна "Системи штучного інтелекту" вивчається студентами факультету інформаційних технологій у 7-му семестрі. Актуальність вивчення основних понять та засад штучного інтелекту майбутніми спеціалістами в галузі інформаційних технологій зумовлена тим фактом, що на сучасному етапі розвитку суспільства все більшого значення набуває інтелектуальна обробка та аналіз даних. Поступово "розумні пристрої" входять у повсякденний побут людей.

У літературі запропоновано багато різних підходів до розуміння предмета та завдань штучного інтелекту [1-3]. У методичному посібнику розглянуто основні поняття, моделі та застосування систем штучного інтелекту у межах та званого символьно-логічного підходу, який є одним з двох найбільш популярних підходів до розробки та аналізу інтелектуальних систем і технологій.

Посібник складається із шести розділів. Перший розділ присвячений стислому огляду базових засад штучного інтелекту. У другому розділі наведено основні відомості про інтелектуальні системи та керування ними. У третьому розділі розглянуто вербально-дедуктивне визначення знань, а також властивості та моделі знань і принципи побудови експертних систем, які засновані на використанні символьно-логічного підходу. Четвертий розділ присвячений мережевим та фреймовим моделям знань. У п’ятому розділі розглянуто основні поняття та можливості мови програмування Пролог, у якій втілено принципи логічного програмування та зворотного виведення. У останньому розділі наведено відомості про продукційні моделі, які з успіхом використовуються при розробці експертних систем.

Виклад матеріалу у посібнику є достатньо лаконічним. Додаткові теоретичні відомості, обґрунтування тверджень, а також більш детальний розгляд відповідних тем читач може знайти у підручниках [2-11], які наведені у переліку рекомендованої літератури. Так, зокрема, основні моделі та методи конекціоністського підходу до розробки систем штучного інтелекту можна знайти у підручнику [11], а вичерпний опис мови Пролог і основних засад логічного програмування — у монографії [6].

1. БАЗОВІ ПОНЯТТЯ ШТУЧНОГО ІНТЕЛЕКТУ

1.1. Означення та історія виникнення

Штучний інтелект (ШІ, англ. Artificial intelligence) — наука та технологія створення інтелектуальних машин, в особливості інтелектуальних комп'ютерних програм. ШІ пов'язаний з завданням використання комп'ютерів для розуміння людського інтелекту, але не обов'язково обмежується біологічно правдоподібними методами (Джон Маккарті, 1956 р., конференція у Дартмутському університеті). В подальшому було зроблено чимало спроб дати формальне визначення інтелекту взагалі і інтелекту штучного зокрема. Найбільш відомим, очевидно, є визначення предмету теорії штучного інтелекту, що було дане видатним дослідником у галузі штучного інтелекту М. Мінські і яке у більш або менш видозміненому вигляді потрапило до словників та енциклопедій: "*штучний інтелект є дисципліна, що вивчає можливість створення програм для вирішення задач, які при вирішенні їх людиною потребують певних інтелектуальних зусиль*". Це визначення зустріло критику, яка полягала в тому, що під нього можна підвести що завгодно, наприклад, виконання простих арифметичних операцій. Відтак до цього визначення додається поправка: "*сюди не входять задачі, для яких відома процедура їх вирішення*". Таке визначення також важко вважати задовільним [1].

Рассел та Норвіг [2] наводять класифікацію означень ШІ у табличній формі

Системи, які мислять подібно до людини	Системи, які мислять раціонально
Системи, які діють подібно до людини	Системи, які діють раціонально

Єдиної відповіді на питання чим займається ШІ, не існує. Майже кожен автор дає своє визначення. Зазвичай ці визначення зводяться до наступних:

- штучний інтелект вивчає методи розв'язання задач, які потребують людського розуміння. Тут мова іде про те, щоб навчити ШІ розв'язувати тести інтелекту. Це передбачає розвиток способів розв'язання задач за аналогією, методів дедукції та індукції, накопичення базових знань і вміння їх використовувати.
- штучний інтелект вивчає методи розв'язання задач, для яких не існує способів розв'язання або вони не коректні (через обмеження в часі, пам'яті тощо). Завдяки такому визначенню інтелектуальні алгоритми часто використовуються для розв'язання NP-повних задач, наприклад, задачі комівояжера.
- штучний інтелект займається моделюванням людської вищої нервової діяльності.
- штучний інтелект — це системи, які можуть оперувати з знаннями, а найголовніше — навчатися. В першу чергу мова ведеться про те, щоби визнати клас експертних систем (назва походить від того, що вони спроможні замінити «на посту» людей-експертів) інтелектуальними системами.

Останнє визначення, що з'явилося у 1990-х рр., засноване на так званому агентно-орієнтованому підході. Цей підхід акцентує увагу на тих методах і алгоритмах, які допоможуть інтелектуальному агенту виживати в оточуючому середовищі під час виконання свого завдання. Тому тут значно краще вивчаються алгоритми пошуку і прийняття рішення.

Джек Коупленд у праці "Що таке ШІ" відзначає, що незважаючи на наявність численних підходів та визначень як до розуміння ШІ, так і до створення інтелектуальних ін-

формаційних систем, можна виділити два **основні підходи щодо розроблення систем ШІ**:

1) низхідний (Top-Down AI, семіотичний, символьний) — створення експертних систем, баз знань та систем логічного виведення, що моделюють та імітують високорівневі психічні процеси: мислення, міркування, мова, емоції, творчість і т.п.;
2) висхідний (Bottom-Up AI, біологічний, конекціоністський) — вивчення нейронних мереж і еволюційних обчислень, які моделюють інтелектуальну поведінку на основі біологічних елементів, а також створення відповідних обчислювальних систем, таких як нейрокомп'ютери [11].

Другий підхід, власне кажучи не відноситься до визначення ШІ, яке дав Дж. Маккарті. Їх поєднує тільки кінцева мета.

Відсутність чіткого визначення ШІ не заважає оцінювати *інтелектуальність* на *інтуїтивному* рівні. Можна навести як мінімум два методи такої оцінки:

метод експертних оцінок. Рішення про ступінь інтелектуальності приймає досить велика група експертів (незалежно або у взаємодії між собою);

метод тестування. Існує значна кількість так званих інтелектуальних тестів, апробованих практикою, і ці тести широко використовуються для оцінки рівня розумових здібностей людини, а також у психології та психіатрії.

Для прикладу наведемо декілька типових тестових завдань.

Приклад 1. Вставте слово, яке означає те саме, що і два слова поза дужками:

дерево (….) підробка

Приклад 2. Вставте число, яке пропущене:

36 30 24 18 _

Приклад 3. Викресліть зайве слово:

лев лисиця жираф щука собака

І метод експертних оцінок, і метод тестування мають свої недоліки. Головний з них полягає у тому, що оцінка дається виходячи лише з власних уявлень експертів, авторів тестів і т.п. про те, як має бути. Тому ці способи не дуже придатні для оцінки будь-якого інтелекту, крім людського.

Серед психіатрів можна почути вислів: "він міркує логічно вірно, але неправильно". Наприклад, дається тестове завдання: "серед слів соловей, чапля, перепілка, стілець, шпак виділити зайве".

Більшість людей, не задумуючись, дає відповідь стілець, тому що всі інші слова — це назви птахів. І раптом хтось дає відповідь шпак, пояснюючи це тим, що це єдине слово, в якому відсутня літера "л".

Обидві класифікації є логічно вірними і формально рівноправними. Але чому ж перевага віддається одній з них? Тому, що так міркує більшість людей. А чому так міркує більшість людей? Очевидно, тому, що *перша класифікація вважається більш важливою для людської практики*. Це положення приймається на аксіоматичному рівні, без доведення. Але те, що є більш важливим для людської практики, зовсім не обов'язково буде так само важливим для розумного робота або для інопланетянина.

Необхідно підкреслити, що поняття “штучний інтелект” не можна зводити лише до створення пристроїв, які імітують людину в усій повноті її діяльності. Насправді ж, спеціалісти які працюють в цій області вирішують іншу задачу: виявити механізми, які лежать в основі діяльності людини, щоб застосувати їх при вирішенні конкретних науково-технічних задач**.** І це лише одна з можливих проблем.

1.2. Приклади інтелектуальних задач

До переліку інтелектуальних задач можна віднести [1, 3]:

- розпізнавання образів;
- логічне мислення;
- аналіз ситуації;
- розуміння нової інформації;
- навчання і самонавчання;
- планування цілеспрямованих дій;

Більш детально зупинимося на перших двох задачах.

На інтуїтивному рівні можна сформулювати декілька типових задач розпізнавання:

- *ідентифікувати* об’єкт, що спостерігається людиною, тобто вирізнити його серед інших (наприклад, побачивши іншу людину, впізнати у ній свою дружину);
- здійснити розпізнавання у класичній постановці, тобто віднести об’єкт, що спостерігається людиною, до одного з заздалегідь відомих класів об’єктів (наприклад, відрізнити легковий автомобіль від вантажного);
- провести кластеризацію (розбиття множини об’єктів на класи);

Людина робить класифікацію просто. Чоловік, повернувшись додому, відразу ж пізнає свою дитину, собаку і т.д. Але він рідко може *пояснити, як* він це робить. Якби це можна було зробити, алгоритми розпізнавання можна було б легко програмувати і широко застосовувати.

Теорія розпізнавання, яка інтенсивно розвивається, необхідна для того, щоб навчити вирішувати задачі розпізнавання і штучні інтелектуальні системи. Зокрема, сформульовано такий ключовий принцип:

будь-який об’єкт у природі – унікальний; унікальні об`єкти – типізовані.

У відповідності до цього принципу, розпізнавання здійснюється на основі аналізу певних характерних *ознак*. Вважається, що в природі не існує двох об’єктів, для яких співпадають *абсолютно всі* ознаки, і це теоретично дозволяє здійснювати ідентифікацію. Якщо ж для деяких об’єктів співпадають *деякі* ознаки, ці об’єкти теоретично можна об’єднувати в групи, або класи, за цими співпадаючими ознаками. "Близькість" значень ознак дає змогу проводити розбиття множини на кластери.

Проблема полягає у тому, що різноманітних ознак дуже багато. Незважаючи на легкість, з якою людина проводить розпізнавання, вона дуже рідко в змозі виділити ознаки, суттєві для цього. До того ж, об’єкти, як правило, змінюються з часом.

Розпізнавання об’єктів і ситуацій має виняткове значення для орієнтації людини в навколишньому світі і для прийняття правильних рішень. Розпізнавання, як правило, здійснюється людиною на інтуїтивному, підсвідомому рівні, людина навчилася цьому за мільйони років еволюції. На цьому фоні спроби навчити розпізнаванню складних об’єктів

штучні інтелектуальні системи, навіть шляхом автоматизованого виділення характерних ознак, мають досить слабкі досягнення.

Логічне мислення — перехід від початкових положень до їх наслідків за формалізованими законами логіки. І тут пересічна людина рідко в змозі пояснити, за якими алгоритмами вона здійснює логічні побудови. Але методики, за якими можна автоматизувати логічне мислення, досить відомі.

Перш за все, це формальна логіка Аристотеля на основі конструкцій, які отримали назву *силогізмів*. Класичний приклад.

Перше положення. *Усі люди смертні.* Друге положення. *Сократ — людина.*

Висновок: *Сократ смертний.*

Якщо перше та друге положення у силогізмі істинні та задовольняють певним загальним формальним вимогам, тоді і висновок буде істинним незалежно від змісту тверджень, що входять до силогізму.

Виконання формальних вимог є важливим, інакше легко припуститися логічних помилок, подібних до таких:

Всі студенти вузу А знають англійську мову. Петров знає англійську мову. Отже, Петров — студент вузу А.

Або: Іванов не готувався до іспиту і отримав двійку. Сидоров не готується до іспиту. Отже, і Сидоров отримає двійку.

Аристотелем було запропоновано декілька формальних конструкцій силогізмів, які він вважав достатньо універсальними. У XIX столітті почала розвиватися сучасна математична логіка, яка розглядає аристотелевські силогізми як один із часткових випадків.

Основою більшості сучасних систем, призначених для автоматизації логічних побудов, є *метод резолюцій* Робінсона. Але практична автоматизація логічного мислення зіткнулася з двома серйозними проблемами.

Перша з них — феномен, який Р. Беллман називав *прокляттям розмірності*. Зовнішній світ являє собою винятково складне переплетіння різноманітних об’єктів та зв’язків між ними. Для того, щоб тільки ввести всю цю інформацію до пам’яті інтелектуального комп’ютера, може знадобитися не одна тисяча років.

Інша проблема — *алгоритмічна нерозв’язність*. Відомо, що в рамках будь-якої досить складної формальної теорії існують положення, які є істинними, але які не можна ні довести, ні спростувати (теорема Геделя про нерозв’язність формальної арифметики.

Реальні програми, які здійснюють логічне виведення (вони часто називаються *експертними системами*) мають досить обмежене застосування. Вони мають обмежений набір фактів та правил з певної, більш-менш чітко окресленої предметної галузі і можуть використовуватися лише у цій галузі.

Що ж стосується людини, то якість її логічного мислення часто буває далеким від бездоганного. Люди рідко проводять логічні побудови до кінця , часто роблять логічні помилки, а інколи взагалі керуються принципами, невірними з точки зору нормальної логіки, а рішення приймається на підсвідомому, інтуїтивному рівні. Зрозуміло, що таке рішення може бути помилковим. Але, якби це було не так, людина була б практично не здатною ні до якої діяльності — ні до фізичної, ні до продуктивної розумової.

Для задач, які розглядалися вище, характерною була спільна риса: їх *погана формалізованість*, відсутність або невизначеність чітких алгоритмів вирішення. Саме такі задачі і являють собою основний предмет розгляду в теорії штучного інтелекту.

До зовсім іншого класу відносяться *обчислювальні задачі*. Важко відповісти на запитання, як саме людина здійснює ті чи інші обчислення. Добре відомими є і низька швидкість, і невисока надійність цього виду людської діяльності. Але були запропоновані ефективні принципи комп'ютерних обчислень.

1.3. Тест Тюринга

Праця А. Тюринга "Обчислювальні машини та інтелект" з'явилася у 1950 р. і була однією з перших публікацій з даної тематики. Тюрінг розглянув питання про те, чи можна примусити машину думати по-справжньому. Відзначивши, що існує певна невизначеність у самому питанні (що таке "машина" і що значить "думати"), яка виключає можливість раціональної відповіді, Тюрінг запропонував замінити питання про інтелект більш чітким емпіричним тестом.

Тест Тюринга порівнює здібності гадано "розумної" машини із здібностями людини — єдиним і найкращим стандартом розумної поведінки. В цьому тесті машину і її людського супротивника (слідчого, експерта) поміщають у окремі кімнати. Ще у одній кімнаті знаходиться "людина". Експерт не може бачити машину чи людину і може спілкуватися з ними лише за допомогою текстового пристрою, наприклад комп'ютерного термінала. Слідчий повинен відрізнити комп'ютер від людини виключно за допомогою їх відповідей на запитання, які він їм задає. Якщо слідчий не може відрізнити комп'ютер від людини, тоді, як стверджує Тюринг, машину потрібно визнати розумною.

Особливості тесту Тюринга:

1. Тест дає об'єктивне уявлення про інтелект, тобто реакцію розумної людини на певний набір запитань.
2. Виключаються якісь особливі вимоги про внутрішню будову комп'ютера.
3. Виключає можливість упередженого ставлення на користь живих істот;

Методики, засновані на тесті Т. стали незамінними при розробці та перевірці сучасних експертних систем.

Багато спеціалістів вважали, що тест Тюринга є цілком задовільним для визначення рівня інтелектуальності комп'ютерної системи. Але виявилось, що це не зовсім так.

В основі тесту Тюринга лежить неявне припущення про те, що необхідною умовою ведення діалогу є розуміння співрозмовника. Але у кінці 60-х рр. американський кібернетик Дж. Вейценбаум створив дві програми — ЕЛІЗА і ЛІКАР. Вони були створені для використання в психіатрії, але виявилось, що за їх допомогою можна обманути експерта в умовах тесту Тюринга. В основі лежить ідея *фатичного діалогу* [1].

Фатичним діалогом називається діалог *без розуміння співрозмовника*, на основі формального перефразування почутого.

Так, наприклад, на висловлювання пацієнта "*Ви не відверті зі мною*" програма може відповісти "*Чому Ви думаєте, що я не відверта з Вами?*" У ряді випадків програма може видати заздалегідь заготовлену фразу. Так, на слова пацієнта "*Мій батько мене не любив*" вона може відповісти "*Розкажіть мені про Вашу сім'ю*".

Звичайно, це не має ніякого відношення ні до розуміння, ні до інтелекту, але багато експертів, які проводили з цими програмами тест Тюринга, вирішили, що мають справу з людиною.

Фатичний діалог може бути і менш примітивним, але системи, які здатні тільки на підтримку такого діалогу, не можна вважати інтелектуальними. *Розуміння* співрозмовника

є абсолютно необхідним для нормального діалогу, а для такого розуміння *необхідно мати певну суму знань про світ.*

Поява систем, що реалізують фатичний діалог, завдала сильного удару по визначенню Тюринга. Безумовно, воно зберігає своє методологічне значення, але вже не може претендувати на універсальність (так само, як і будь-яке інше з відомих визначень).

Нижче у дещо спрощеному вигляді наведено фрагмент діалогу, характерного для програми ЛІКАР:

ЛІКАР: *Я Вас слухаю.*
ЛЮДИНА: *Мене бентежить погода.*
ЛІКАР: *Як давно Вас бентежить погода?*
ЛЮДИНА: *Відтоді, як померла моя мати.*
ЛІКАР: *Розкажіть про Вашу сім'ю.*
ЛЮДИНА: *Всі вони захоплюються комп'ютерами.*
ЛІКАР: *Машини Вас лякають.*
ЛЮДИНА: *Ні.*
ЛІКАР: *Будь ласка, не будьте так небагатослівні…*
І так далі до нескінченності.

Класичний метод написання програм, що реалізують фатичний діалог — застосування *співставлення зі зразком*. В основі методу лежить *співставлення* речень, які вводяться людиною, зі *зразками*, що зберігаються програмою. В залежності від результату співставлення реалізується та чи інша заздалегідь запрограмована операція. І самих зразків, і операцій, що їм відповідають, як правило, порівняно небагато.

Можна стверджувати, що співставлення зі зразком є однією з найбільш фундаментальних загальноінтелектуальних процедур. Спробуємо формалізувати це поняття.

Визначення. Нехай M — набір зразків, що зберігаються в пам'яті інтелектуальної системи, F — деяка множина функцій або алгоритмів, і $r(a, b, f)$ — деякий критерій якості. Тоді під співставленням елемента q, що спостерігається інтелектуальною системою, зі зразком будемо розуміти знаходження елементу $m \in M$ і функції $f \in F$, таких, що $f(q) = m$ або $f(m) = q$.

Природно також накласти одну з двох умов:

- максимізація критерію якості: $r(m, q, f) \to \max$;
- досягнення задовільного рівня якості: $r(m, q, f) > \varepsilon$, де ε — заданий поріг.

Ця постановка задачі співставлення є дуже загальною. Її конкретизації залежать від того, які обмеження накладаються на множини M і F та на критерій якості. Можна вважати, *що більш інтелектуальна система повинна вміти здійснювати більш складне співставлення*.

Можна розглянути кілька варіантів порівнянь, кожний з яких може бути легко запрограмований.

Варіант 1 (повний збіг). Якщо речення, що вводиться, повністю збігається з одним із зразків, може прозвучати відповідь: *"Так, Ви маєте рацію"*, або навпаки: *"Ви помиляєтесь, оскільки..."*, і після *"рацію"* або *"оскільки"* програміст може написати будь-який текст, що імітує глибоке розуміння специфіки предметної області. Наприклад, дуже непогано виглядатиме такий діалог:

Людина**:** *Квадрат гіпотенузи дорівнює сумі квадратів катетів.*

Програма: *Так, але є подібний результат і для непрямокутних трикутників — це теорема косинусів.*

Навряд чи після кількох подібних відповідей у когось залишаться сумніви в інтелектуальних здібностях програми. Але, звичайно, повні збіги трапляються дуже рідко. Тому доводиться використовувати інші типи порівнянь.

Варіант 2 (використання замінювачів). Типовим є використання замінювачів * і ?. Із замінювачем * зіставляється довільний фрагмент тексту, із замінювачем ? — будь-яке окреме слово.

Наприклад, шаблон *(*комп'ютери* *) успішно зіставляється з будь-яким реченням, в якому згадується про комп'ютери; шаблон *(Я люблю ? яблука)* — з такими реченнями, як *(Я люблю червоні яблука), (Я люблю солодкі яблука)* тощо (але не з реченням *Я люблю їсти зелені яблука).*

Варіант 3 (надання значень змінним у процесі зіставлення). При цьому можливості програми, що реалізує фатичний діалог, значно розширюються. Вона набуває здатності до генерації відповідей, які залежать від запитань. Так, правило

$$(Я\ ?\ *) \overset{a:=?}{\rightarrow} \{Що\ Ви\ ще\ <a>\ ?)$$

дозволяє на речення *"Я люблю яблука"* відповісти *"Що Ви ще любите?"*, а на речення "*Я ненавиджу дощі" — "Що Ви ще ненавидите?".* У цьому прикладі при успішному зіставленні змінній *a* надається значення слова, з яким збігається замінювач ?. Безумовно, при використанні українських фраз потрібно стежити за узгодженням суфіксів і закінчень.

Варіант 4 (універсальний зразок). Зі зразком (*) зіставляється будь-яке речення. Звичайно, і відповіді, що відповідають цьому зразкові, повинні бути такими ж універсальними. Наприклад:

(Я Вас не дуже розумію)
(Не будьте такими небагатослівними)
(Чому це має для Вас значення?)

і т. п.

Варіант 5 (зіставлення більше ніж з одним зразком). Речення може зіставлятися не з одним зразком, а кількома. Наприклад, якщо в програмі задані підстановки

(люблю *) —> (Що Ви ще любите?)*

та

(комп'ютери *) —> (Ви маєте здібності до техніки),*

то речення *(Я люблю комп'ютери)* зіставляється з обома зразками.

2. ІНТЕЛЕКТУАЛЬНІ СИСТЕМИ

2.1. Керування складними системами

2.1.1. Алгоритмічний та декларативний підходи до керування

На найбільш загальному рівні, можна виділити два підходи до керування складними системами та до програмування роботів і комп'ютерів, що могли б розв'язувати ті чи інші задачі. При програмуванні сучасних комп'ютерів в основному реалізований традиційний **алгоритмічний підхід**, який можна ще назвати імперативним. Цей підхід вимагає заздалегідь продумати та детально розписати, як треба вирішувати певну проблему. Написання програми вимагає задання чітких послідовностей інструкцій. Якщо таку послідовність вдається написати — комп'ютер зможе вирішувати дану задачу, якою б складною вона не була. Але, ясна річ, він виконувати інструкції, абсолютно не розуміючи їх змісту. Якщо зміняться умови, за яких виконується програма, комп'ютер може виявитися безпомічним.

Інший підхід — **декларативний.** Інтелектуальному виконавцеві (людині чи комп'ютерові) досить сказати, **що** треба робити, тобто лише сформулювати завдання, побудувавши всі взаємозв'язки між об'єктами у предметній області. **Як** це завдання буде виконуватися — повинен визначити сам виконавець.

Запустити космічний корабель так, щоб він приземлився на Марсі, взяв зразки ґрунту та привіз їх назад — задача дуже складна, але вона піддається точній алгоритмізації. Математичні методи дозволяють точно розрахувати траєкторію, по якій повинен рухатися такий корабель.

Послати робота до магазину за пляшкою молока — задача, на декілька порядків більш складна. Не кажучи вже про сам процес спілкування з продавцем, робот повинен вирішити ряд не зовсім формалізованих підзадач. Заздалегідь розрахувати траєкторію руху неможливо, оскільки робот повинен уникнути зіткнень з людьми та автомобілями. Якщо магазин закритий на переоблік, він повинен знайти інший магазин. *Неможливо передбачити всі ситуації, які можуть виникнути, а відтак — алгоритмізувати розв'язок задачі.* Тому виконання такого завдання під силу лише інтелектуальній системі, яка вміє *орієнтуватися в зовнішньому світі, аналізувати поточні ситуації та коригувати, адаптувати свою поведінку на основі такого аналізу*.

2.1.2. Формалізація понять алгоритмічності та декларативності

Спробуємо формалізувати деякі з впроваджених раніше понять. Введемо такі позначення:

q — *первинні інструкції,* записані без будь-яких змін у тому вигляді, в якому їх сформулював автор; аналогічно можна визначити *первинний опис ситуації* як результат її безпосереднього сприйняття;

S — множина факторів, які визначають поточний стан виконавців; розглядатимемо S як об'єднання двох множин: S_1 та S_2, де S_1 — множина *контрольованих факторів,* які відомі авторові процедури і на які він може мати вплив, S_2 — множина *неконтрольованих* факторів;

Z — знання, які має виконавець;

r_A — робочий алгоритм, який формується та реалізується виконавцем при алгоритмічному підході. При цьому, як правило, автоматично формулюється і програма p_A, що відповідає цьому алгоритму;

r_D — робочий алгоритм, який формується та реалізується інтелектуальним виконавцем при декларативному підході. Згідно з фундаментальними тезами Тюринга та Черча (все, що може бути виконано будь-яким виконавцем, може бути промодельоване на машині Тюринга) сам факт виконання завдання свідчить про існування цього алгоритму. Щоправда, цей алгоритм може і не усвідомлюватися виконавцем, а тому не може бути явно сформульований у вигляді алгоритму чи програми (якщо ж програма, що відповідає алгоритму r_D, повинна бути згенерована, йдеться мова про *задачу синтезу програм*).

Тоді можна записати такі співвідношення:

$$r_A = f\left(q, S_1\right),$$

$$r_D = g\left(q, S_1, S_2, Z\right).$$

Принциповим є наступне. Алгоритм r_A формується на основі первинних інструкцій q однозначно. При цьому також можуть відбуватися зміна і поповнення первинних інструкцій, але цей процес має повністю контрольований і детермінований характер. Як приклад можна навести компіляцію програм, написаних мовами високого рівня. Тому автор процедури може бути впевнений у гарантованому результаті (якщо, звичайно, були дотримані певні формальні вимоги до первинних інструкцій і забезпечено належний стан виконавця).

На противагу цьому, за декларативного підходу такої впевненості немає. Інтелектуальний виконавець певним чином поповнює первинні інструкції, але їх авторові не завжди відомо, яким чином це поповнення відбувається. Тому не можна бути впевненим у результаті виконання інструкцій, і ця втрата гарантованості є неминучою платою за відмову від алгоритмічності. Отже, ми маємо справу з узагальненням поняття алгоритму, яке дістало назву **"квазіалгоритм".**

2.2. Квазіалгоритми

Алгоритмом називається чітка однозначна зрозуміла виконавцеві послідовність інструкцій, виконання яких обов'язково приводить до гарантованого результату за скінченний час. На відміну від цього, інструкції квазіалгоритму можуть бути не зовсім чіткими, і результат виконання квазіалгоритмічної процедури не обов'язково є гарантованим.

Можна виділити як мінімум чотири основні *джерела квазіалгоритмічності*:

1. **Дія випадкових чинників**, що не залежать від виконавця. Строго кажучи, з огляду на цей фактор квазіалгоритмом слід вважати будь-який, навіть найбільш формалізований, алгоритм. Алгоритм розрахований на роботу при певних умовах; якщо ці умови зміняться, алгоритм може не призвести до потрібного результату. *Якщо ж не вдається чітко окреслити межі застосування алгоритму або забезпечити виконання необхідних умов для його роботи, ми маємо справу зі справжнім квазіалгоритмом*.

2. **Недостатнє врахування автором алгоритмічної процедури особливостей виконавця.** Це призводить до того, що виконавець неправильно розуміє, що від нього вимагається. Процедура може бути в принципі алгоритмічною, за нормальних умов

призводити до гарантованого результату, але цей результат може бути не передбачений автором і тому бути для нього несподіванкою.

3. **Нечіткість формулювань**, що фігурують в описі процедури. Розглянемо будь-який кулінарний рецепт, наприклад: “розтерти тісто, змішати з дрібно нарізаними яблуками, додати солі і перцю за смаком і смажити до появи рум’яної скоринки”. Це є типовим прикладом нечіткості формулювань, що призводить до невизначеностей. До якої межі слід розтирати тісто? Що означає “дрібно нарізані”? Що означає “за смаком” і т. д. Неінтелектуальна система, орієнтована на чисто алгоритмічне керування, просто не зрозуміє цього опису і тому не зможе його виконати. Інтелектуальна ж система, наприклад, людина, повинна спробувати поповнити і уточнити цей опис на основі наявних знань і досвіду. Сам по собі факт виконання завдання буде свідчити про те, що квазіалгоритмічний первинний опис процедури був зведений до деякого внутрішнього алгоритму, але навряд чи виконавець зможе чітко пояснити і розписати цей алгоритм. Крім того, ці внутрішні алгоритми для кожного виконавця будуть різними.

4**. "Свобода волі",** яку можуть мати високоорганізовані, по-справжньому інтелектуальні системи. Ці системи можуть мати свої цілі, і, якщо дана процедура суперечить цим цілям, вони можуть відмовитися її виконувати або виконати не так, як це від них вимагається. “Свобода волі” полягає у тому, що інтелектуальна система самостійно приймає рішення про свою поведінку і в залежності від цих рішень може виконувати зовнішні розпорядження, не виконувати їх, або ж виконувати так, як вона вважає за потрібне.

2.3. Характеристика інтелектуальних систем з точки зору кібернетики

2.3.1. Означення інтелектуальної системи

Інтелектуальна система може припускати зовнішнє керування, але для неї характерною є **самокерованість**. Система має **певну мету і прагне так планувати свої дії, щоб досягати цієї мети**. Як вхідні стимули системи можна розглядати поточну ситуацію, що сприймається і аналізується системою. Результатом реакції системи стає зміна зовнішньої ситуації, і поведінка системи коригується в залежності від того, бажаною чи небажаною є ця зміна.

Людина має певну суму **знань про світ**, яка дозволяє їй орієнтуватися в життєвих ситуаціях та приймати правильні рішення. Крім того, людина вміє певним чином **використовувати ці знання**. Ці самі риси мають мати системи штучного інтелекту.

Також можна стверджувати, що **здатність до поповнення первинних знань, є** *однією із ключових рис інтелектуальних систем*. Ця властивість інтелектуальних систем називається **здатністю до навчання.** Розрізняють *зовнішнє навчання* та *самонавчання*.

Підсумувавши усі сказане, можна стверджувати, що **інтелектуальною системою** *називається самокерована кібернетична система, яка має певну суму знань про світ і здатна на основі безпосереднього сприйняття і подальшого аналізу поточної ситуації до планування дій, спрямованих на досягнення мети, а також до навчання*.

2.3.2. Типова схема функціонування інтелектуальної системи

Функціонування інтелектуальної системи можна описати як постійне прийняття рішень на основі аналізу поточних ситуацій для досягнення певної мети. Схема функціонування інтелектуальної системи наведена на рис. 2.1.

Виокремлюють наступні *етапи функціонування інтелектуальних систем*:

1. **Безпосереднє сприйняття зовнішньої ситуації.**
   Результатом є первинний опис ситуації.

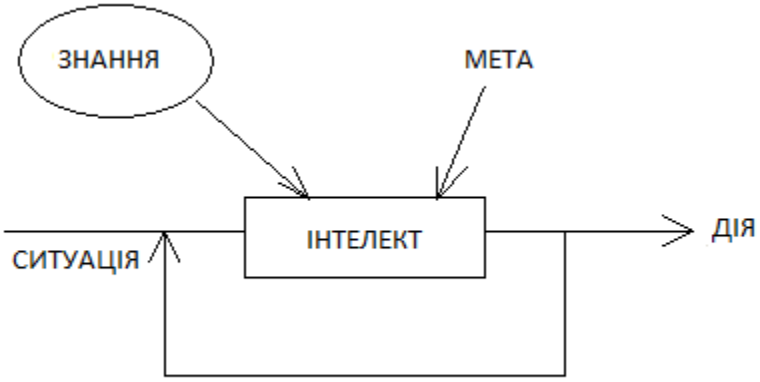


Рис. 2.1. Схема функціонування інтелектуальної системи.

2. **Зіставлення первинного опису зі знаннями системи і поповнення цього опису.**
   Результатом є формування вторинного опису ситуації в термінах знань системи. Цей процес можна розглядати як процес *розуміння ситуації* або як процес перекладу первинного опису на внутрішню мову системи. При цьому можуть змінюватися внутрішній стан системи та її знання.
   Вторинний опис може бути не єдиним, і система має змогу вибирати між різними вторинними описами. Крім того, система в процесі роботи може переходити від одного вторинного опису до іншого. Якщо ми можемо формально задати форми внутрішнього представлення описів ситуацій та операції над ними, то сподіватимемося на певний автоматизований аналіз цих описів.
3. **Планування цілеспрямованих дій та прийняття рішень, аналіз можливих дій та їх наслідків і вибір тих дій, що найкраще узгоджуються з метою системи.**
   Таке рішення зазвичай формулюється певною внутрішньою мовою (свідомо або підсвідомо).
4. **Зворотна інтерпретація прийнятого рішення, тобто формування робочого алгоритму для реагування системи.**
5. **Реалізація реакції системи; наслідками є зміни зовнішньої ситуації і внутрішнього стану системи і т. д.**

Проте не слід вважати, що зазначені етапи є повністю відокремленими у тому розумінні, що наступний етап починається тільки після того, як повністю закінчиться попередній. Навпаки, для функціонування інтелектуальної системи характерним є взаємне проникнення цих етапів. Наприклад, ті чи інші рішення можуть прийматися вже на етапі безпосереднього сприйняття ситуації. Насамперед це рішення про те, на які зовнішні подразники слід звертати увагу, а на які не обов’язково. Зовнішніх подразників так багато, що їх сприйняття повинно бути вибірковим.

3. ПОДАННЯ ЗНАНЬ В ІНТЕЛЕКТУАЛЬНИХ СИСТЕМАХ

3.1. Підходи до подання знань

У філософії знання визначаються як "відображення об'єктивних властивостей та зв'язків світу".

Знання *є інформаційною основою інтелектуальних систем, оскільки саме вони завжди зіставляють зовнішню ситуацію зі своїми знаннями і керуються ними при прийнятті рішень. Не менш важливим є те, що знання — це систематизована інформація, яка може певним чином поповнюватися і на основі якої можна отримувати нову інформацію, тобто нові знання.*

Існує багато підходів до визначення поняття "знання". На сучасному етапі домінуючою парадигмою, що лежить в основі найвідоміших моделей подання знань у системах штучного інтелекту, можна вважати парадигму (певну сукупність ключових принципів), характерну для **символьного підходу**. Цю парадигму можна охарактеризувати як **вербально-дедуктивну**, або **словесно-логічну**, через певні чинники:

- будь-яка інформаційна одиниця задається вербально, тобто у формі, наближеній до словесної, у вигляді набору явно сформульованих тверджень або фактів;
- основним механізмом отримання нової інформації на базі існуючої є **дедукція**, тобто висновок від загального до часткового. Такий дедуктивний підхід є бездоганним з логічного погляду. В його основі лежать транзитивність імплікації (якщо з a випливає b, з b випливає c, то з a випливає c) і дія квантора загальності (якщо деяка властивість P виконується для будь-якого елемента множини M, а $x \in M$, то P виконується для x.

Але вербально дедуктивне задання знань не є повним, оскільки:

- дедуктивний висновок не виступає єдино можливим. Мислення людини багато в чому є рефлекторним, інтуїтивним. Воно, як правило, спирається на підсвідомі процеси. Людина часто робить висновки за аналогією, асоціацією. Ці висновки не завжди вірні, але вони істотно доповнюють процеси дедуктивного мислення. Без підсвідомого мислення стає неможливим (будь-яке відкриття — як правило, підсвідоме породження гіпотези, яка потім перевіряється дедуктивним або експериментальним шляхом);
- далеко не всі знання є вербальними. Так, жодне твердження не зберігається в пам'яті людини явно. Відомо, що в основі діяльності мозку людини лежить передача сигналів між нервовими клітинами. Часто людина не може сформулювати свої знання. Наприклад, будь-хто знає, що таке "стіл". Але якщо попросити людину дати визначення цього поняття, у неї можуть виникнути проблеми. Таким чином, поняттями можна оперувати і не знаючи чіткого їх визначення. Типовими є слова: "Я не можу пояснити чому, але мені здається.

Тому необхідно розвивати інші моделі знань, окрім вербально-дедуктивних Наприклад, у рамках **конекціоністського підходу** можна розглядати моделі на основі однорідного поля знань. **Однорідним полем знань** називається сукупність простих однорідних елементів, які обмінюються між собою інформацією: нові знання народжуються на основі певних процедур, визначених над полем знань.

3.2. Вербально-дедуктивне визначення знань

У рамках вербально-дедуктивної парадигми можна навести таке визначення знань.

Знаннями *інтелектуальної системи називається трійка <F, R, P>, де F — сукупність явних фактів, які зберігаються в пам'яті системи в явному вигляді, R — сукупність правил виведення, які дозволяють на основі відомих знань набувати нових знань, P — сукупність процедур, які визначають, яким чином слід застосовувати правила виведення.*

Базою знань *(БЗ) інтелектуальної системи називатимемо сукупність усіх знань, що зберігаються в пам'яті системи.*

У базах знань прийнято розрізняти *екстенсіональну* та *інтенсіональну* частини.

Екстенсіональною частиною бази знань називається сукупність усіх явних фактів, *інтенсіональною* частиною — сукупність усіх правил виведення та процедур, за допомогою яких з існуючих фактів можна виводити нові твердження.

Приклад. Розглянемо таку базу знань.

Ф1. Заєць їсть траву.
Ф2. Вовк їсть м'ясо.
ФЗ. Заєць є м'ясом.
ПІ. Якщо *А* є м'ясом, а *В* їсть м'ясо, то *В* їсть *А*.
П2. Якщо *А* їсть м'ясо, то *А* є хижаком.
ПЗ. Якщо *А* їсть *В*, то *В* є жертвою.
П4. Якщо хижак вимирає, жертва швидко розмножується.
П5. Якщо жертва вимирає, хижак вимирає.

Даний приклад є фрагментом неформалізованого опису моделі Лотки-Вольтерра, добре відомої в екології. Літерами "**Ф**" позначені явні факти, які відображають елементарні знання. Їх безпосередній аналіз дозволяє експертній системі відповідати на найпростіші запитання, такі, як "Вовк їсть м'ясо?". Для позитивної відповіді досить просто знайти відповідний факт в екстенсіональній частині бази знань.

Принципово іншим є запитання типу "Вовк є хижаком?". Відповідь на нього на основі простого пошуку знайти неможливо: такого явного факту в екстенсіональній частині бази знань немає. Тому для відповіді на це запитання необхідно застосовувати правила виведення, які дозволяють на основі існуючих явних фактів набувати нових знань, тобто нових фактів. Такі правила позначені у нашому прикладі літерою "**П".** Наприклад, для відповіді на поставлене запитання досить знання факту **Ф2** і правила **П2.** Безумовно, до бази знань можна було б відразу включити твердження "Вовк є хижаком". Але легко побачити: якщо, крім вовків, розглядати інших хижаків (ведмедів, лисиць та ін.), то безпосередньо включати до бази знань явні факти, що вони є хижаками, недоцільно. Все, що може бути виведене логічним шляхом, як правило, не варто оголошувати фактами. Надалі називатимемо явні факти просто фактами, якщо це не викликає непорозумінь.

Нарешті, **процедури** визначають, яким саме чином слід застосовувати правила. Часто ці процедури є складовою мови, якою програмується експертна система (це стосується насамперед спеціалізованих мов ШІ, таких як Лісп, Пролог, Пленер).

3.3. Експертні системи

База знань, наведена в п. 4.2, може лягти в основу *експертної системи*, тобто інформаційної системи, яка здійснює дедуктивне виведення на основі наявних знань. Існують різні визначення експертних систем, основна суть яких зводиться до такого.

Експертні системи *— це інтелектуальні програмні засоби, здатні у діалогу з людиною одержувати, накопичувати та коригувати знання із заданої предметної галузі, виводити нові знання, розв'язувати на основі цих знань практичні задачі та пояснювати хід їх розв'язку.*

Експертні системи акумулюють знання експертів — провідних спеціалістів у даній предметній галузі. В основі роботи експертних систем лежить дедуктивне виведення нових тверджень з існуючих. Типове застосування експертних систем — консультації для фахівців середньої кваліфікації і неспеціалістів у тій галузі, для якої вона розроблена.

Створити універсальну експертну систему неможливо. По-перше, це пов'язано з "прокляттям розмірності". По-друге, експертна система повинна акумулювати знання людей-експертів, а "універсальних" експертів не існує і не може існувати. По-третє, жодне логічне виведення не може замінити інтуїцію та досвід експерта.

Можна лише експертні системи, які належать до конкретної предметної галузі. Остання передбачає лімітований набір явищ і понять певної сфери людської діяльності та обмежене коло задач, які вирішуються у цій галузі. Існує чимало експертних систем у таких сферах як медична і технічна діагностика, пошук корисних копалин, юриспруденція, аналіз інвестицій і комерційних ризиків і т. п. У створенні експертних систем повинні брати участь фахівці як мінімум двох категорій:

- **експерт,** що є висококваліфікованим фахівцем у даній предметній області, знання якого потрібно передати експертній системі;
- **інженер знань,** завдання якого — формалізувати знання експерта і *привести* їх до вигляду, придатного для занесення до бази знань.

Серйозна проблема у даному випадку пов'язана з тим, що знання експертів важко формалізувати. Експерт часто не може сформулювати свої знання у явному вигляді, робить правильні висновки, але *не може пояснити як саме він їх робить*. Якщо ж експерт і формулює певні правила виведення, вони далеко не завжди відзначаються точністю та адекватністю. З цього випливає ряд інших труднощів, які так чи інакше виявляють себе на етапі формалізації знань або застосування готових експертних систем.

3.4. Дані та знання

Відомо, що сучасний етап розвитку інформатики характеризується еволюцією моделей даних у напрямі переходу від традиційних (реляційна, ієрархічна, мережева) до моделей знань. Знання, як будь-яку інформацію, можна вважати даними. Але *знання є високоорганізованими даними, для яких характерна певна внутрішня структура та розвинуті зв'язки між різними інформаційними одиницями.* Іншим принциповим аспектом є те, що інформаційні системи, які ґрунтуються на знаннях, повинні мати можливість отримувати нові знання на основі існуючих. У цьому ми вже пересвідчилися на прикладах дедуктивного виведення нових знань з явних фактів.

Екстенсіональна частина БЗ складається з фактів, які запам'ятовуються явно, інтенсіональну — з правил, які дозволяють отримувати нові факти. Наявність розвинутої

інтенсіональної частини є однією з основних рис, які відрізняють знання від традиційних даних. **Інтенсіональним відношенням,** яке описує базу даних, часто називають правило або сукупність правил, яким підпорядковується кожний запис у базі даних.

Приклад. Розглянемо базу даних (БД) деканату, що містить дані про те, які курси прослухав кожний студент. Відомо, що жоден студент не має права слухати курс "Бази даних", якщо він не прослухав курсу "Основи програмування". Нехай у базі даних зберігається інформація про Іванова, Петрова та Сидорова, які прослухали обидва курси.

Екстенсіональна частина:

Прізвище	Дисципліна
Іванов	Бази даних
Петров	Бази даних
Сидоров	Бази даних

Інтенсіональна БД (або її вже можна назвати БЗ) може мати такий вигляд:

Правило: Якщо Прослухав(***X,*** Бази даних), то Прослухав**(*X,*** Основи програмування**).**

Наявність такого правила дозволяє скоротити базу знань: твердження, істинність яких можна встановити за допомогою правил, не обов'язково запам'ятовувати в явному вигляді.

Пошук в інтенсіональній БЗ складається з двох етапів:

- пошук потрібного факту в екстенсіональній частині;
- дедуктивне виведення факту на основі правил інтенсіональної частини.

Реалізовувати інтенсіональні правила можна навіть у рамках реляційної моделі даних шляхом програмування відповідних запитів.

Знання можуть бути **неповними.** Це означає, що для доведення або спростування певного твердження може не вистачати інформації. У багатьох системах логічного виведення прийнято **постулат замкненості світу:** на запит про істинність деякого твердження система відповідає "так" тоді і тільки тоді, коли його можна довести; якщо ж довести неможливо, система відповідає "ні". Водночас *"неможливо довести через нестачу інформації"* і *"доведено, що ні"* — це зовсім не те саме. З огляду на це бажано, щоб експертна система запитувала у користувача про факти, яких не вистачає.

Знання можуть бути **недостовірними.** Наприклад, на виготовлення продукції можуть впливати випадкові чинники (об'єктивна невизначеність) або експерт може бути не зовсім упевненим у деякому факті чи правилі (суб'єктивна невизначеність).

Ненадійність знань і недостовірність наявних фактів обов'язково повинні враховуватися в процесі логічних побудов. Звичайно, можна було б просто відкидати факти та правила виведення, які викликають сумнів, але довелося би відмовитися від цінної інформації. Крім того, в експертних системах часто доводиться мати справу з неточно визначеними поняттями, такими, як *"великий", "маленький"* тощо.

3.5. Зв'язки між інформаційними одиницями

У базі знань можуть зберігатися відомості про різні об'єкти, поняття, процеси і т. п., тобто відповідні описи, які називаються **інформаційними одиницями.**

Опис; який утворює ***інформаційну одиницю,*** *розглядається як деяка абстракція реальної сутності, що існує в дійсному світі.*

Це означає що дана абстракція описує певні риси реальної сутності. Бажано, щоб абстракція описувала найважливіші з них.

У реальному світі все тісно взаємопов'язане. Відповідні зв'язки повинні знайти адекватне відображення в БЗ. Існує велика група зв'язків, за допомогою яких можна описувати просторові відношення ("знизу", "зверху", "поруч", "між" і т. п.); часові відношення ("раніше", "пізніше", "під час", "одночасно" і т. п.), причинно-наслідкові відношення тощо. Ці відношення є спільними для всіх предметних областей. Логічні системи, що ґрунтуються на цих відношеннях, називаються **псевдофізичними логіками.** Використання псевдофізичних логік (гак само, як і більш загальних предикатів, таких, як узагальнення та агрегація) дає можливість інтелектуальній системі поповнювати первинні описи.

Так, якщо інтелектуальна система сприймає текст: "У 1985 році почалася перебудова. У 1991 році була проголошена незалежність України", врахування часових властивостей, що визначається часовою логікою, дає змогу відповідати на запитання типу: "Що було раніше: початок перебудови и чи проголошення незалежності України?"

Основними зв'язками між інформаційними одиницями є **узагальнення** та **агрегація.**

Агрегація дозволяє задавати будову об'єкта та його складових. Наприклад, агрегація дає змогу визначити в базі знань аудиторію як об'єкт, що має вікна, двері і т. п. Вікна, в свою чергу, теж можна розглядати як агрегований об'єкт: вони мають ручки, рами і т. п. Інакше кажучи, відношення *"Має",* яке описує агрегацію, означає, що певний об'єкт містить у своєму складі деякий інший об'єкт. За цим відношенням закріпилася спеціальна назва **Has_Part.** По суті, це відношення "ціле — частина".

Таким саме чином ввести і зворотне відношення (*"Є частиною",* або **Is_Part**).

Узагальнення (відношення *"Є"*) задає ієрархію класів **(екземпляр — клас — надклас).** Так, можна говорити, що об'єкт Вася Петров належить до класу Студент. Відповідно інформаційна одиниця, яка описує Васю Петрова, може бути описана на основі більш загальної інформаційної одиниці Студент (точніше, вона задається як екземпляр інформаційної одиниці Студент). Клас Студент, у свою чергу, є *підкласом* типу Людина і т. п. Надклас об'єднує атрибути, або ознаки, спільні для його підкласів; підкласи можуть успадковувати ті чи інші властивості надкласів.

Із відношенням *"Є"* пов'язаний один з основних відомих механізмів логічного виведення — механізм **виведення за успадкуванням** (інша назва — виведення **за наслідуванням).** Суть цього механізму можна сформулювати так: якщо деяка умова виконується для всього класу, то вона виконується і для кожного представника цього класу, а також для всіх підкласів цього класу (якщо інше не задано явним чином). Інакше кажучи, екземпляри успадковують властивості класів, підкласи успадковують властивості надкласів.

В усіх системах керування базами знань повинна бути забезпечена належна підтримка успадкування. Розглянемо *приклад* [8]. У базі знань міститься така інформація:

Усі птахи літають.
Ластівка є птахом.
Юкко є ластівкою.

Маємо загальний клас *"Птахи",* його підклас *"Ластівки"* та об'єкт *"Юкко",* який є екземпляром класу *"Ластівка".* На підставі цих знань будь-яка система, що підтримує успадкування, повинна зробити висновки, що *Юкко є птахом; всі ластівки літають; Юкко теж літає.*

Насправді, замість єдиного відношення *"Є"* необхідно розглядати цілу систему споріднених, але не ідентичних відношень. Інакше легко припуститися логічних помилок, подібних до таких:

Юкко є ластівкою.

Ластівка є видом, який вивчається натуралістами.

Отже, *Юкко є видом, який вивчається натуралістами*.

Слід розглянути такі відношення:

- **"клас — підклас",** яке є відношенням часткового порядку (звичайно строгого). Для цього відношення виконується властивість транзитивності;
- **"екземпляр — клас".** Екземпляри успадковують властивості своїх класів, але саме відношення "екземпляр — клас" не є транзитивним.

"Ластівка" є екземпляром класу "Види, які вивчаються натуралістами", а "Юкко" є екземпляром класу "Ластівка", але "Юкко" у жодному разі не є екземпляром класу *"Види, які* "Види, які вивчаються натуралістами" ("Юкко" взагалі не є видом).

Також необхідно чітко розрізняти поняття **"клас"** і **"множина", "належність до класу"** та **"належність до множини".** Клас об'єднує однотипні елементи, множина — необов'язково. Відношення "елемент — множина" також не є транзитивним. Складні системи можуть бути описані у вигляді певної канонічної форми, яка являє собою композицію двох ієрархій — **ієрархії класів** та **ієрархії об'єктів.**

Ієрархія класів задається відношенням *узагальнення,* **ієрархія об'єктів** — *відношенням агрегації.*

Дійсно, з одного боку, такі об'єкти, як "ручки", "рами"та інші, утворюють новий об'єкт — "вікно". "Вікна", "двері" тощо — це об'єкти, які утворюють ще один новий об'єкт: "аудиторію". Таким чином, можна розглядати певну ієрархію об'єктів. З іншого ж боку, всі ці об'єкти є екземплярами своїх класів. Так, конкретне вікно належить до класу "Вікна", клас "Вікна" є підкласом класу "Дерев'яні вироби" і т. п.

Відношення "клас — підклас" є характерним для ієрархії класів, відношення "елемент — множина" та "підмножина — множина" — до ієрархії об'єктів, а відношення "екземпляр — клас" об'єднує обидві ієрархії.

3.6. Проблема винятків

З успадкуванням пов'язана дуже серйозна проблема — **проблема винятків.** Вона полягає в тому, що деякі підкласи можуть не успадковувати ті чи інші властивості надкласів. Інакше кажучи, характерні риси класу успадковуються всіма його підкласами, **крім** деяких.

Нехай відомо, що *літають всі птахи, крім пінгвінів* (існують деякі інші види птахів, які не літають, але для наших цілей це не має суттєвого значення). Якби це твердження відразу потрапило до БЗ саме в такому вигляді, особливих проблем не виникало б (хоча і в цьому випадку слід було б передбачити належну обробку винятків).

Але, як було зазначено раніше, експерт не завжди може сформулювати свої знання в явному вигляді. Зокрема, він може не знати або не пам'ятати всіх винятків. Тому він може

спочатку включити до БЗ твердження про те, що *всі птахи літають,* а потім пригадати, що *пінгвіни не літають,* і додати це до БЗ. У результаті ми могли б отримати БЗ, подібну до такої:

Усі птахи літають.
Ластівка є птахом.
Юкко є ластівкою.
Пінгвін є птахом.
Пінгвіни не літають.
Бакс є пінгвіном.

Якби три останні твердження **не були** включені до бази знань, системі просто дійшла б хибного висновку, що *Бакс літає.* Але включення даних відомостей до бази знань ще більше ускладнює ситуацію. **Система знань стає суперечливою: з** одного боку, система повинна дійти висновку, **що** Бакс літає, а з іншого — що Бакс не літає. У даному разі кажуть про **втрату монотонності** дедуктивної системи [8].

Система дедуктивного виведення *називається* ***монотонною,*** *якщо для неї виконується така властивість: якщо з набору тверджень* $(q_1, \ldots, q_n)$ *випливає твердження V, то V випливає і з набору тверджень* $(q_1, \ldots, q_n, r)$.

Інакше кажучи, в монотонній системі додавання нових фактів і правил не впливає на істинність висновків, які могли бути отримані без них.

Для вирішення проблеми винятків часто використовують наступне правило:
у разі виникнення суперечностей підклас успадковує відповідну властивість лише від найближчого попередника, тобто найближчого до нього в ієрархії класів.

3.7. Властивості та моделі знань

У [9] сформульовані такі особливості знань, які відрізняють їх від звичайних даних:

1) **внутрішня інтерпретованість:** кожна інформаційна одиниця повинна мати унікальне ім’я, за яким інформаційна система її знаходить, а також відповідає на запити, в яких це ім’я згадується;
2) **структурованість:** знання повинні мати гнучку структуру; одні інформаційні одиниці можуть включатися до складу інших (відношення типу "частина — ціле", "елемент — клас");
3) **зв’язність*:*** в інформаційній системі повинна бути передбачена можливість встановлення різних типів зв’язків між різними інформаційними одиницями (причинно-наслідкові, просторові та ін.);
4) **семантична метрика*:*** на множині інформаційних одиниць корисно задавати відношення, які характеризують ситуаційну близькість цих одиниць;
5) **активність:** виконання програм в інтелектуальній системі повинно ініціюватися поточним станом бази знань.

Часто виокремлюють інші властивості знань, наприклад **шкальованість,** яка означає, що формально неоднакові поняття насправді відображаються на одній і тій самій **шкалі понять,** різні точки якої відповідають інтенсивності того самого фактора. Так, температура може бути високою або низькою, і це породжує такі поняття, як "холодно", "тепло", "гаряче" тощо.

Моделлю знань *називається фіксована система формалізмів (понять і правил), відповідно до яких інтелектуальна система подає знання в своїй пам'яті та здійснює операції над ними.*

Моделі задання знань необхідні:

- для створення спеціальних мов описів знань і маніпулювання ними;
- для формалізації процедур зіставлення нових знань з уже існуючими;
- для формалізації механізмів логічного виведення.

Як уже зазначалося раніше, найрозвинутішими на сучасному етапі є моделі задання знань, які ґрунтуються на вербально-дедуктивній парадигмі. Найвідомішими з них є чотири класи моделей:

- семантичні мережі;
- фреймові;
- логічні;
- продукційні.

Вербально-дедуктивні моделі задання знань мають багато спільних рис, Отже, можна вважати, що всі вони мають єдину концептуальну основ).

3.8. Неоднорідність знань. Області і рівні знань

Для успішного здійснення операцій зі знаннями необхідно відокремлювати в БЗ певні фрагменти, які називаються **областями знань.** Ці області знань повинні бути відносно незалежними між собою, що означає таке:

- зміни в одній області знань не повинні приводити до суттєвих змін у інших областях;
- вирішення складної задачі можна, як правило, звести до підзадач таким чином, що для вирішення кожної з цих підзадач достатньо знань з однієї області.

Такий розподіл є важливим для полегшення проектування і використання БЗ. Зокрема, різні області знань можуть проектуватися незалежно одна від одної.

Виходячи з постановки задачі можна по-різному розділяти знанні на області. Так, для експертних систем, які ведуть діалог з користувачем мовою, наближеною до природної, можна виокремити такі області знань [7]:

- **предметна область,** яка містить знання про конкретну предмет, галузь, в якій працює експертна система;
- **область мови,** яка містить знання про мову, якою ведеться діалог;
- **область системи,** яка містить знання експертної системи про власні можливості;
- **область користувача,** яка містить знання про користувача. Наявність їх дозволяє враховувати індивідуальні особливості кожного користувача. Наприклад, пояснення користувачеві можуть надавати залежно від рівня його підготовленості;
- **область діалогу,** яка містить знання про мету діалогу, а також про форми та методи його організації.

Знання можуть різнитися за **рівнем задання** і **рівнем детальності. За рівнями задання** розрізняють знання нульового рівня (конкретні і абстрактні) і знання вищих рівнів — знання про знання **(метазнання).**

4. МЕРЕЖЕВІ ТА ФРЕЙМОВІ МОДЕЛІ ЗНАНЬ

4.1. Семантичні мережі

В основі мережевих моделей лежить конструкція, що називається **семантичною мережею**.

Під семантичною мережею розуміється мережа, в якій поєднані зв'язки різних типів. **Семантичну мережу** можна неформально описати у вигляді графа, вершини якої, як правило, позначають об'єкти предметної області, а дуги відповідають зв'язкам між ними.

Формально ці моделі можна задати у вигляді $<I, C_1, C_2, ..., C_n, \Gamma>$. Тут I — множина інформаційних одиниць, $C_1, C_2, ..., C_n$ — типи зв'язків між інформаційними одиницями, Γ — відображення, що задає зв'язки між інформаційними одиницями.

Мережні моделі тісно пов'язані з **моделями "сутність-зв'язок"**. Під сутністю розуміється об'єкт довільної природи. Об'єкт, що існує в реальному світі, називається **П-сутністю**. Його представлення в базі знань називається **М-сутністю**. У базі знань відображаються також зв'язки між сутностями.

Виділяють **класифікуючі** мережі, **функціональні** мережі та **сценарії**.

Класифікуючі мережі дозволяють задавати відношення ієрархії між інформаційними одиницями.

Функціональні мережі характеризуються наявністю функціональних відношень, які дозволяють описувати процедури обчислень одних інформаційних одиниць через інші.

У *сценаріях* використовуються відношення типу "причина-наслідок", "дія", "засіб дії" та ін.

Запропоновано ряд формалізацій для представлення знань у вигляді семантичних мереж. Історично першою була модель Квілліана.

Базовим є поняття **концептуального об'єкту**, який графічно зображується **концептуальним графом**. У моделі Куілліана П-сутність описується концептуальним об'єктом, причому до опису включаються клас, до якого належить П-сутність, властивості сутності та його прототип (приклад). Як приклад можна навести концептуальний граф для чайника.

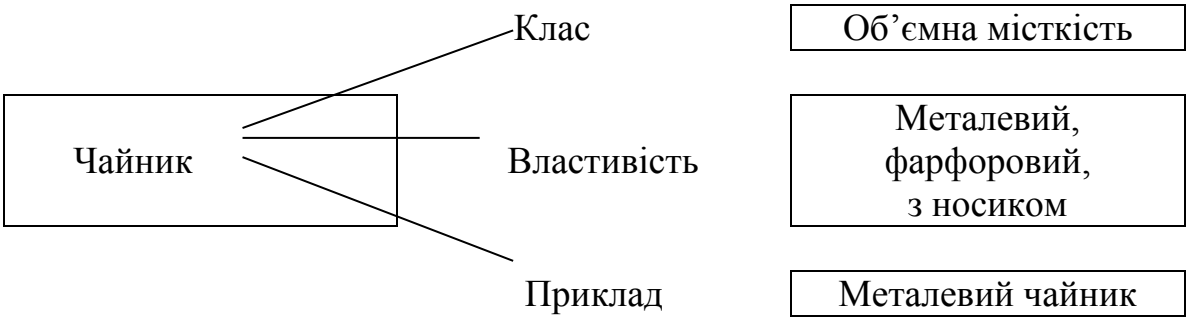


Рис. 4.1. Приклад концептуального графа

Більш загально, у вигляді концептуального графа можна представляти будь-яку логічну формулу або словесне висловлювання. Такий концептуальний граф можна вважати простою семантичною мережею, що відповідає одному твердженню. Так, наприклад, висловлювання **"Жак надсилає книгу Марі"** може бути представлене у вигляді предиката ПОСИЛКА(ЖАК_2, МАРІ_4, КНИГА_22), а концептуальний граф матиме вигляд:

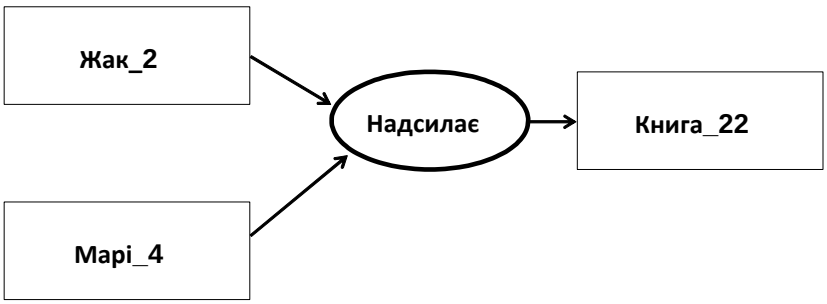


Рис. 4.2. Приклад простої семантичної мережі, що відповідає одному фактові

Цифри потрібні для того, щоб дати кожній сутності своє індивідуальне ім’я, яке не співпадає ні з якими іншими.

Прямокутники концептуального графа призначені для представлення аргументів, а кола — для представлення самих предикатів. Круг і прямокутник з’єднуються дугою, якщо вони є відповідно ім’ям та аргументом того самого предиката.

Для спрощення вигляду концептуального графа можна не виділяти імена предикатів у вигляді окремих кругів, а писати ці імена прямо на дугах.

Часто буває корисним представити предикат, що відповідає деякій складній фразі природної мови, у вигляді кон’юнкції бінарних предикатів. У нашому прикладі предикат ПОСИЛКА(ЖАК_2, МАРІ_4, КНИГА_22) можна переписати у вигляді

Відправник(Посилка_8, Жак_2)
Отримувач(Посилка_8, Марі_4)
Що(Посилка_8, Книга_22)
Is_a(Посилка_8, Посилки).

Відповідний концептуальний граф матиме вигляд:

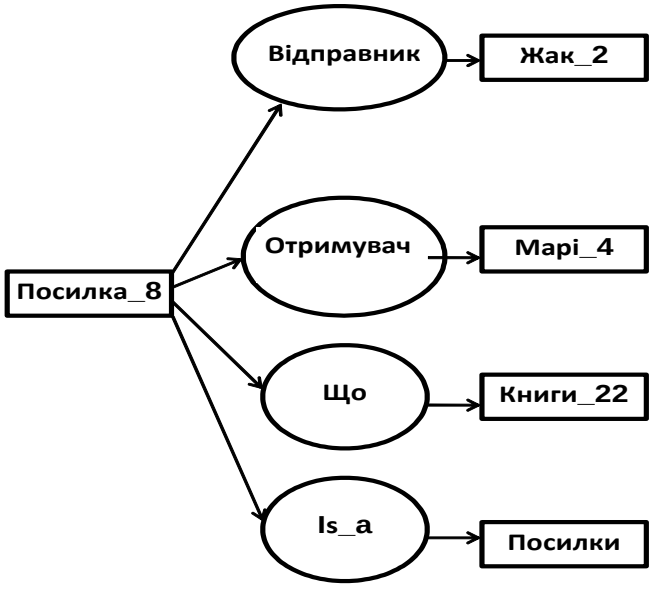


Рис. 4.3. Концептуальний граф у бінарній формі

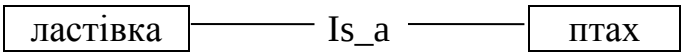
Предикати **Is_a, Has_Part** та **Is_Part** відіграють винятково важливу роль та *означають належність деякої сутності до певного типу / агрегату*.

Семантична мережа — *це композиція концептуальних графів, об'єднаних за тими чи іншими правилами.*

Багато методів логічного виведення на семантичних мережах ґрунтуються на зв'язку **Is_a** та понятті **наслідування**, що тісно з ним пов'язане. Сутність наслідує атрибути типу, до якого вона належить. Якщо для типу визначені конкретні значення атрибутів, сутність, що належить до цього типу, наслідує їх за замовченням.

Зв'язок Part_Of показує відношення **"ціле–частина".** Цей зв'язок показує відношення між екземплярами класу.

Як приклад розглянемо речення, яке задає факт "всі ластівки — птахи". Цьому реченню може відповідати така семантична мережа:



Якщо далі присвоїти ластівці ім'я "Юкко", тоді мережу можна розширити наступним чином:

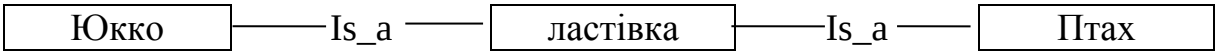


Рис. 4.4. Ілюстрація наслідування на семантичній мережі

Тоді з семантичної мережі можна вивести речення: Юкко — птах.

4.2. Фрейми

Фреймові моделі базуються на теорії фреймів, розробленій у 1975 р. М.Мінським.

Слово "фрейм" походить від англійського слова "frame", яке перекладається як "рамка", "каркас". Мінський запропонував гіпотезу, згідно з якою знання групуються в модулі, і назвав фреймами ці модулі. Фрейм можна інтуїтивно уявляти собі як каркас, на якому тримаються знання, або як рамку, на основі якої описуються різноманітні об'єкти та ситуації. Мінський писав, що коли людина потрапляє в нову ситуацію, вона **співставляє** цю ситуацію з тими фреймами, які зберігаються у неї в пам'яті.

Фрейм *можна визначити як структуру даних, що призначена для опису типових ситуацій або типових понять.*

М. Мінський визначив ***фрейм*** *як мінімальний опис деякої сутності, такий, що подальше скорочення цього опису призводить до втрати цієї сутності.*

Розглянемо, наприклад, поняття **"Студент",** яке описується відповідним фреймом. Кожний студент може бути охарактеризований такими характеристиками, як прізвище, ім'я та по-батькові, факультет, курс. Ці характеристики у фреймовій моделі зображуються **слотами** [7].

На основі фрейму "**Студент**" може бути описаний будь-який конкретний студент. Цей опис відбувається шляхом заповнення слотів конкретними значеннями. Можна сказати, що ***екземпляри понять*** *утворюються в результаті конкретизації* ***фреймів понять.***

Для фреймових моделей характерна **ієрархія понять**. У нашому прикладі фрейм "**Студент**" наслідує слоти від більш загального фрейму "**Людина**". Тому, якщо фрейм "**Людина**" описаний, відповідні слоти в описі фрейму "**Студент**" можна не задавати. Якщо ж якісь слоти уже заповнені конкретними значеннями, то ці значення можуть успадковуватися фреймами-нащадками.

За допомогою фреймів можуть описуватися і більш складні об’єкти. Розглянемо, наприклад, опис аудиторії на основі фрейму "**Аудиторія**":

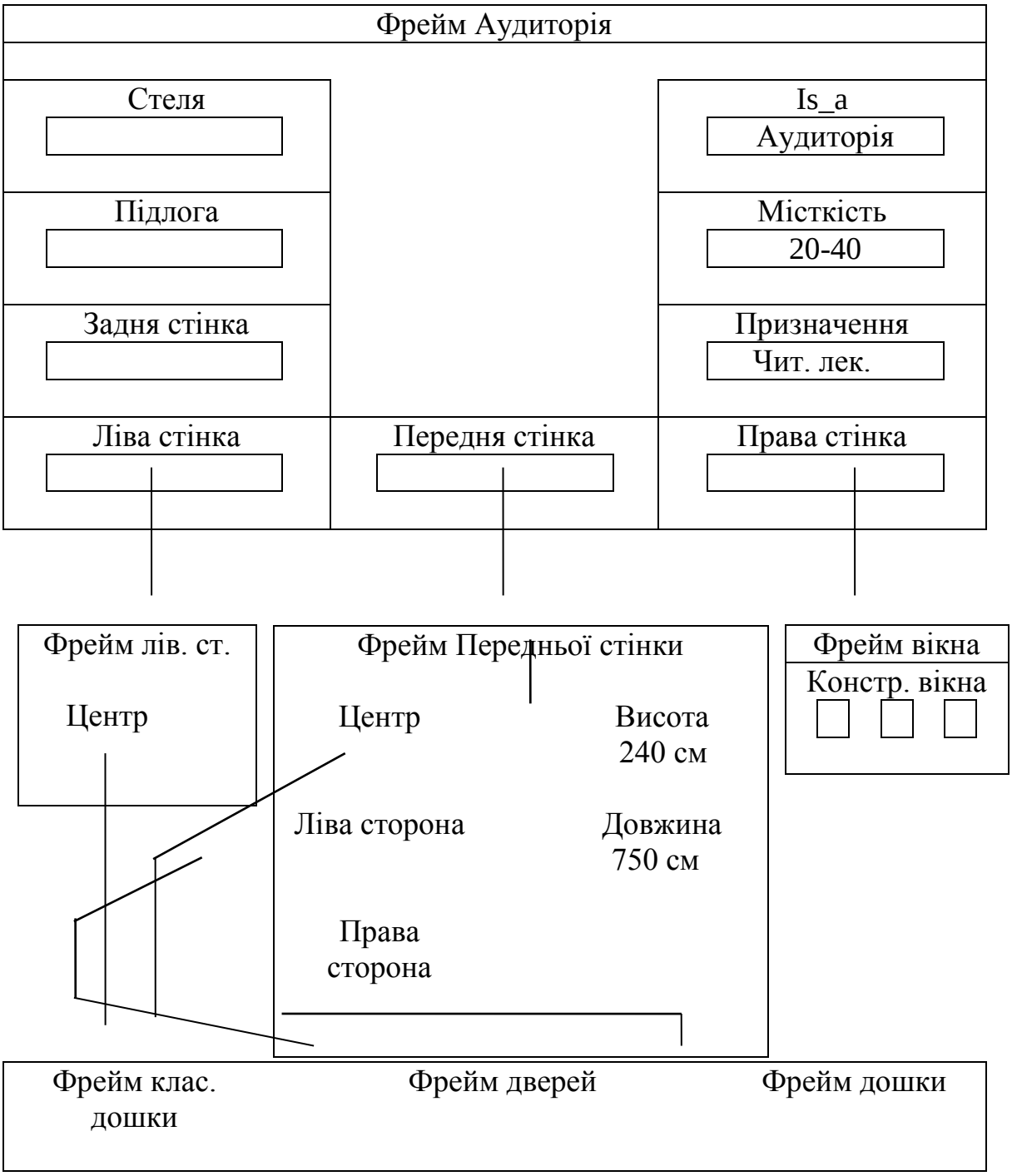


Рис. 4.5. Фрейм аудиторії

На рис. 4.5 фактично зображена мережа частково конкретизованих фреймів, пов'язаних між собою зв'язком **Has_Part**. Взагалі, фрейми можуть об'єднуватися в мережі, і тому їх часто розглядають в одному контексті з семантичними мережами [1].

Один і той самий предмет може описуватися різними фреймами, які відповідають різним умовам спостерігання цього предмета. Риси, спільні для цих різних фреймів, описуються **базовим фреймом**. Можна вважати, що фрейми, які відповідають різним умовам спостереження, успадковують основні характеристики базового фрейму. Зрозуміло, що зберігати ці похідні фрейми в пам'яті не обов'язково.

При пошуку фреймів, які найбільш підходять, велике значення мають мережі подібностей, відмінностей та ін. [10].

Існують різні моделі зображення знань за допомогою фреймів. Фрейм може мати, наприклад, таку структуру:

Ім'я фрейму	Вказівник наслідування	Визначення типу даних	Значення слоту	Демони
Слот 1				
Слот 2				
……				
Слот *n*				

Рис. 4.6. Структура даних фрейму

Основні елементи цієї структури:

1) Ім'я фрейму. Цей ідентифікатор є єдиним в даній фреймовій системі.

Кожний фрейм складається з довільного числа слотів (декілька з них системні, решта — користувача). Традиційно перелік основних системних слотів такий:

a) слот Is_a, що вказує на фрейм-предок даного фрейму;

b) слот-вказівник фреймів нащадків, який є списком вказівників цих фреймів;

c) слот для введення імені користувача, дати визначення, дати зміни, тексту коментаря;

d) інші слоти.

Кожний слот в свою чергу задається фіксованою структурою даних.

2) Ім'я слоту — ідентифікатор, унікальний в фреймі. Деякі імена слотів мають смислове навантаження:

a) Is_a(визначає тип відношення);

b) DESCENDANTS (вказівник прямого дочірнього фрейму);

c) DEFINED_BY (користувач, який визначає фрейм);

d) DEFINED_ON (дата визначення фрейму);

e) MODIFIED_ON (дата модифікації фрейму);

f) COMMENT — коментар;

g) Has_Part, RELATIONS — системні і використовуються при редагуванні бази знань і керуванні виведенням.

3) Вказівники наслідування використовуються тільки в фреймових системах ієрархічного типу, що ґрунтуються на відношеннях "Is_a". Вони вказують, яку інформацію про атрибути слотів фрейму вищого рівня наслідують слоти з таким же ім'ям в фреймі

нижчого рівня.

Типові вказівники наслідування:

a) UNIQUE (U: унікальний) — показує, що кожний фрейм може мати слоти з різними значеннями;

b) SAME (S: такий же) — всі слоти повинні мати однакові значення;

c) RANGE (R: встановлення меж) — значення слотів нижчого рівня повинні бути в межах, вказаних значеннями слотів фрейма верхнього рівня;

d) OVERRIDE (O: ігнорувати) — при відсутності вказівки значення слотів фрейму верхнього рівня стає значенням слота фрейму нижчого рівня, але у випадку визначення нового значення, значення слотів нижчого рівня вказуються в якості значень слотів. O виконує одночасно функції вказівників U і S.

4) Визначення типу даних — вказується, що слот має числове значення, або виступає вказівником іншого фрейму.

5) Значення слоту. Поле для введення значення слоту. Значення слоту повинно співпадати з вказаним типом даних цього слоту, і , крім того, *повинна виконуватись умова наслідування*.

6) Демон. Тут дається визначення **демонів** типу **if_needed, if_added, if_removed.** *Демоном* називається процедура, яка *автоматично запускається при виконанні деякої умови*. Демони запускаються при зверненні до відповідного слоту. Наприклад, демон **if_needed** запускається, якщо в момент звернення до слоту його значення не було встановлено, **if_added** — при підстановці значення в слот, **if_removed** — при стиранні значення слоту.

Демон можна вважати різновидом *приєднаної процедури*, які також можна включати до опису фреймів. Приєднана процедура запускається за повідомленням, переданим з іншого фрейму.

5. ЛОГІЧНІ МОДЕЛІ. ЛОГІЧНЕ ПРОГРАМУВАННЯ

5.1. Логічна модель знань

Логічна модель задання знань базується на формалізмах логіки предикатів першого порядку. Позитивною характеристикою логічних моделей є однозначність теоретичного обґрунтування і можливість реалізації системи формально точних визначень і висновків; недолік — формальний процедурний стиль мислення. Людська логіка — це інтелектуальна модель з нечіткою структурою. В цьому полягає її відмінність від строгої логіки.

Логічною моделлю називається формальна система, що задається четвіркою <*T*, *P*, *A*, *B*> [1]**.** Тут *T* — це множина базових елементів, *P* **—** множина синтаксичних правил; *A* **—** множина аксіом, *B* **—** множина правил виведення.

5.2. Основні поняття мови Пролог

Пролог є **мовою логічного програмування**. Парадигма логічного програмування означає [2], що *для виконання програм використовується механізм доведення теорем*, який дозволяє з'ясувати, чи містяться суперечності у деякому наборі логічних формул. Сама програма розглядається як набір логічних формул разом з теоремою, яка повинна бути доведена.

Пролог — це мова програмування, *призначена для обробки символьної нечислової інформації*. Особливо добре Пролог пристосований для вирішення завдань, в яких фігурують об'єкти і зв'язки між ними [6]**.**

Програмування на мові Пролог включає наступні етапи:

- опис *фактів* про об'єкти та відношення між ними;
- визначення *правил* про об'єкти та відношення між ними;
- формулювання *питань* (запитів) про об'єкти та відношення між ними;

5.2.1. Факти

Припустимо, потрібно повідомити системі Пролог факт: "Джону подобається Мері". Цей факт містить два об'єкта та відношення "подобається". У мові Пролог використовується наступна форма запису фактів:

```
likes(john, mary).
```

Потрібно дотримуватися наступних правил:

Імена усіх відношень та об'єктів мають починатися з малої літери. Спочатку записується ім'я відношення. Потім у дужках через кому записуються імена об'єктів.

Кожний факт має завершуватися крапкою.

Порядок імен об'єктів відіграє значення.

Розглянемо наступний набір фактів:

```
likes(john, mary). %John likes mary
likes(john, fish).
likes(mary, book).
likes(john, book).
valuable(gold).
```

```
female(jane).
owns(john, gold).
father(john, mary).
gives(john, book, mary).
```

Кожний раз використання деякого імені має на увазі деякий індивідуальний об'єкт, який має інтерпретуватися однаково у всій базі знань.

Ім'я відношення, яке розташовується перед дужками, називається предикатом, а кількість його аргументів — місністю (арністю) предиката.

Сукупність фактів у Пролозі називається *базою даних*.

5.2.2. Запити

Після визначення сукупності фактів можна звертатися до системи Пролог з питаннями про задані відношення. Ці питання записуються так само, як факти, але перед ними ставиться спеціальний знак "?–".

Розглянемо питання

```
?- likes(mary, book).
```

Сенс запитання полягає у перевірці того, чи має Мері задану конкретну книгу (ми не запитуємо, чи має вона усі книги, чи які-небудь книги).

Система Пролог *сприймає запити як цілі, які потрібно досягти*.

Звертання до Прологу із запитом ініціює процедуру пошуку у базі даних, завантажених попередньо у систему. Система Пролог шукає в базі даних факти, зіставні з фактом у питанні. Два факти зіставні (або відповідають один іншому), якщо їх предикати однакові та їх відповідні аргументи співпадають. У разі успіху повертається значення `Yes.`

5.2.3. Змінні

Складніші запити можна отримати використовуючи змінні. Для знаходження усього, що подобається Джону, можна сформулювати питання "Подобається Джону *X*?" Це питання записуються у Пролозі так:

```
?– likes(john, X).
```

Це питання інтерпретується як "Існує що-небудь, що подобається Джону?".

Для позначення змінних у Пролозі використовуються ідентифікатори, які починаються із великої літери.

У процесі пошуку факти у БД переглядаються послідовно у порядку їх введення. *При цьому неконкретизована змінна вважається зіставною з будь-яким аргументом предиката, розташованим у відповідній позиції*. У процесі пошуку *знаходяться усі значення X, які дозволяють досягти мету*. Після знаходження зіставного факту змінна X конкретизується і в базі даних спеціальним маркером позначається місце, у якому відбулося співставлення. Це робиться для того, щоб мати можливість знайти наступні значення X, які можуть привести до задоволення мети. Лексична область визначення змінної — одне речення.

Якщо значення деяких змінних не використовується у програмі, то замість них можна використовувати анонімні змінні, які записуються як символ підкреслювання:

```
gives(john, _, mary).
```

Кожний раз при появі символу підкреслювання він позначає нову анонімну змінну.

Питання до системи може містити кілька цілей. Для того щоб перевірити, чи подобаються Джон та Мері один одному, треба зробити запит

```
likes(john, mary), likes(mary, john).
```

Якщо *цілі відокремлені комою*, то вважається, що перед системою ставиться завдання досягти *кон'юнкцію цілей*, причому запит опрацьовується зліва направо. Кожна ціль має свій маркер, які дають змогу здійснювати backtracking (пошук з поверненням).

Якщо цілі відокремлюються двокрапкою, то система шукає диз'юнкцію цілей.

5.2.4. Визначення відношень за допомогою правил

У Пролозі для того, щоб вказати, що деякі відношення визначаться на основі інших відношень, використовуються правила. Правило — це деяке *твердження про об'єкти та про зв'язки між ними*. Між фактами та правилами існує важлива відмінність. Факти є логічними твердженнями, які є безумовно істинними. З іншого боку правила — твердження, які є вірними при виконанні деяких умов.

У Пролозі правила складаються із висновку (заголовку або голови) та тіла (умови), які відокремлюються за допомогою символу :—. Символ ':—' читається *якщо*. Тіло складається із списку цілей, відокремлених комами, які розглядаються як знаки кон'юнкції. Факти та правила, які задають предикат, називаються *твердженнями*.

Припустимо, ми хочемо сформулювати твердження про те, що Джону подобаються усі жінки.

```
likes(john, X) :- female(X).
```

Кожне входження змінної у правило позначає один і той самий об'єкт. При формулюванні правил можна використовувати кон'юнкцію чи диз'юнкцію фактів інших правил.

```
likes(john, X) :- female(X), young(X).
likes(john, X) :- female(X); food(X).
```

Приклад. Розглянемо правило: "Людина може вкрасти предмет, якщо ця людина злодій, їй подобається цей предмет та предмет є цінним".

```
canSteal(Person, Object):- thief(Person),
        likes(Person, Object), valuable(Object).
```

Фрази Пролога відносяться до трьох категорій: факти, правила та питання. Факти — це правила з порожнім тілом, питання — правила без голови.

5.2.5. Рекурсивні правила

Розглянемо двомісний предикат `hasPart` для опису відношення агрегації (позначення того факту, що складовою частиною першого аргументу є другий аргумент). Опишемо за допомогою цього предикату наступні факти:

```
hasPart(car, wheel).
hasPart(car, engine).
```

```
hasPart(bike, wheel).
hasPart(car, door).
hasPart(door, handle).
hasPart(bike, engine).
hasPart(window, glass).
hasPart(door, glass).
```

Тоді можна визначити відношення contains для перевірки належності об'єктів до складу інших об'єктів:

```
contains(Aggregate, Detail) :- hasPart(Aggregate, Detail).
contains(Aggregate, Detail) :- hasPart(Aggregate, Part),
                               contains(Part, Detail).
```

Завдання для самостійної роботи. Написати правила для

1) Двомісного відношення `inSameAggregate(X, Y)` (X та Y входять до складу того самого агрегату).

2) Одномісного відношення `inDifferentAggregate(X)` (X входить до складу різних агрегатів).

5.3. Об'єкти даних у Пролозі

З точки зору синтаксису усі об'єкти даних у Пролозі є термами [6]. Об'єкти даних поділяються на *прості об'єкти* та *структури*. *Прості об'єкти* діляться на *константи* та *змінні*. *Константи* бувають двох типів: *атоми* та *числа*.

Структура — це єдиний об'єкт, який складається із сукупності інших об'єктів, які називаються компонентами. Для з'єднання компонентів структури використовуються *функтори*.

Приклад описання структур:

```
likes(mary, book('White fang', 'Jack London')).
segment(point(2,3), point(5,6)).
segment(point(2,3), point(1,1)).
segment(point(1,6), point(1,1)).
segment(point(1,1), point(5,6)).
vertical(segment(point(X, _), point(X, _))).
```

Завдання для самостійної роботи. Написати правила для

1) відношення перпендикулярності двох відрізків;
2) перевірки належності трикутника до одного із наступних типів прямокутні, гострокутні, тупокутні.

5.4. Основні операції Прологу

5.4.1. Рівність і встановлення відповідності

У Пролозі визначений предикат рівність "=", який використовується для узгодження термів. У випадку узгодження із базою даних цілі

```
? – X = Y.
```

Пролог робить спробу зробити зіставлення (встановити відповідність між X та Y). Цільове твердження вважаться доведеним, якщо така відповідність існує. При узгодженні з базою даних цілей вигляду X = Y, де X та Y — довільні терми, в яких можуть входити неконкретизовані змінні, використовуються наступні правила:

- якщо X неконкретизована змінна, а Y — конкретизована (має певне значення), то X та Y узгоджуються. Крім того, змінні X також стає конкретизованою — вона набуває того самого значення, що і Y.
- Дві однакові константи (цілі числа або атоми) завжди рівні (узгоджуються) між собою.
- Дві структури рівні, якщо вони мають однаковий функтор і однакову кількість аргументів, причому відповідні аргументи рівні. Наприклад, наступна ціль буде успішно досягнута і змінна X буде конкретизована значенням bicycle.

```
rides(student, bicycle) = rides(student, X).
```

Цільове твердження `X = Y` *є завжди вірним, якщо один із аргументів неконкретизований*.

Предикат `"\="` відповідає відношенню "не рівне". Ціль `X \= Y` досягається тоді і тільки тоді, коли не можна довести твердження `X = Y`.

5.4.2. Арифметичні операції

У Пролозі для виконання арифметичних операцій треба зробити окрему вказівку. Наприклад, наступний запит

```
?- X = 1 + 2.
```

не приведе до присвоювання змінній X значення 3.

Для обчислення значень виразів використовується інфіксний оператор `is`. Його правий аргумент — терм, значення якого треба обчислити. Наприклад:

```
?- X is 1 + 2.
```

Результат обчислень перевіряється на відповідність із лівим аргументом. Якщо він неконкретизований, то змінна конкретизується результатом обчислень. Основні операції:

```
+,  –,  *,  /,  //, **,  mod.
```

5.4.3. Операції порівняння

Операції порівняння так само як операції **is** також змушують систему Пролог виконати арифметичні обчислення (у своїх обох аргументах!).

Для перевірки нерівності "менше або рівне" використовується символ "=<".

`X =:= Y`— операція перевірки рівності значень X та Y.

`X =\= Y` — операція перевірки нерівності значень X та Y.

Операція перевірки рівності "=:=" відрізняється від операції "=", оскільки *вона не спричиняє конкретизацію змінних*, а лише *приводить до обчислення значень виразів для обох аргументів* та їх порівняння.

Для *перевірки ідентичності* двох термів використовується операція "==", яка приймає істинне значення, якщо обидва операнди ідентичні (неконкретизована змінна рівна сама собі та усім змінним, зчепленим із нею попередніми операціями "="). Операція "==" не викликають попереднє обчислення значень своїх аргументів.

Для перевірки на "неідентичність" використовується відношення "`\==`".

Для лексикографічного порівняння термів використовуються операції `@<`, `@>`, `@=<` та `@>=`.

5.4.4. Заперечення як недосягнення мети

Унарний предикат not визначений таким чином, що виклик `not(goal)` повертає істинне значення, якщо ціль Goal є недосяжною. Наприклад

```
animal(dog).
animal(cobra).
animal(rabbit).
snake(cobra).
likes(mary, X):-
    animal(X), not(snake(X)).
```

У стандарті Пролога для заперечення передбачено позначення "\+":

```
\+snake(X).
```

Приклад. Предикат `likesToOnlyJohn` задовольняють лише ті аргументи, які подобаються лише Джону.

```
likesExceptJohn(X) :- likes(Y, X), Y \== john.
likesToOnlyJohn(X) :- likes(john,X),
                      not(likesExceptJohn(X)).
```

При використанні оператора not потрібно враховувати гіпотезу про замкненість світу, яка приймається в системах ШІ. Наприклад, ввівши запит

```
?- not(human(mary)).
```

скоріше за все отримаємо відповідь `yes`. Цю відповідь потрібно сприймати не як твердження про те, що Мері не є людиною. Вона лише вказує на те, що у системі нема достатньої інформації для того щоб визначити є Мері людиною чи ні.

Розглянемо наступні факти та правила:

```
goodQuality(samsung).
goodQuality(apple).
expensive(apple).
moderate(X) :- not(expensive(X)).
```

На запит

```
?- goodQuality(X), moderate(X).
```

система Пролог знайде відповідь `X = samsung`. Якщо ж переставити цілі:

```
?- moderate(X), goodQuality(X).
```

то отримується відповідь `no`.

Відмінність полягає у тому, що у першому запиті перед виконанням цілі `moderate(X)` змінна X вже є конкретизованою, а у другому — ні. Запит `not(expensive(X))` інтерпретується наступним чином:

`not`(існує такий X, що досягається ціль `expensive(X)`)

Це рівносильно твердженню

Для всіх X твердження expensive(X) є хибним.

Завдання для самостійної роботи. Написати правило для відношення `biLikes(X)`: об'єкт X подобається рівно двом особам.

5.5. Списки

5.5.1. Поняття списку

Список — послідовність, яка складається із довільної кількості елементів.

У Пролозі непорожній список складається із двох компонентів:

1) перший елемент (голова списку);

2) решта елементів (хвіст).

Хвіст списку сам є списком.

Для позначення порожнього списку використовується спеціальний атом Пролога [].

Наприклад, для списку `ann, tennis, tom, skiing` головою є `ann`, а хвостом — список `tennis, torn, skiing`.

Список як і усі інші структуровані об'єкти мають ієрархічну (деревоподібну) структуру, наведену на рис. 5.1.

Для описів списків використовують функтор "." із двома аргументами:

```
.(Head, Tail)
```

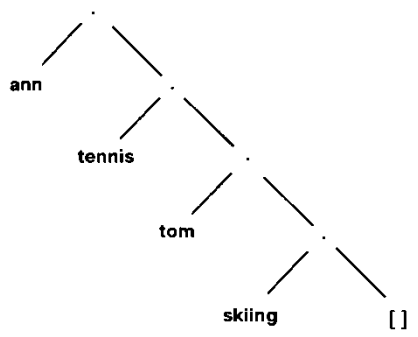


Рис. 5.1. Деревоподібне зображення списку

Наведений у прикладі список записується у вигляді наступного терма:

```
.(ann, .(tennis, .(tom, .(skiing, []))))
```

Для скорочення запису використовується наступна форма запису:

```
[ann, tennis, torn, skiing]
```

Крім того, передбачена ще одна форма запису — вертикальна риска для відокремлення голови списку від хвоста. Наприклад

```
[a, b, c] = [a | [b,c]] = [a, b | [c]] = [a, b, c | []]
```

5.5.2. Деякі операції із списками

1. Перевірка належності елемента до списку.

```
item(X, [X|_]).

item(X, [_|L]) :- item(X, L).
```

2. Добавлення елемента у початок списку.

```
addFront(X, L, [X|L]).
```

3. Конкатенація списків.

```
conc([], L, L).
conc([X|L1], L2, [X|L3]) :- conc(L1, L2, L3).
```

4. Знаходження останнього елемента списку (2 способи).

```
last([X], X).
last([_|L], Y) :- last(L, Y).
last2(L, X) :- conc(_, [X], L).
```

5. Видалення елемента із списку.

```
del(X, [X|L], L).
del(X, [Y|L], [Y|L1]) :- del(X, L, L1).
```

6. Підсписок

```
subList(L1, L2) :- conc(_, L1, L3), conc(L3, _, L2).
```

7. Знаходження найбільшого елемента числового списку

```
max([X|L], X) :- max(L, MaxValue), X > MaxValue.
max([X|L], MaxValue) :- max(L, MaxValue), X =< MaxValue.
```

Для роботи із списками у Swi-Prolog можна використовувати вбудовані предикати `member(?Elem, ?List)`, `append(?List1, ?List2, ?List1AndList2)` та `select(?Elem, ?List1, ?List2)`.

Завдання для самостійної роботи. Написати правила для предикатів `oddLength` — перевірка того, що список має парну довжину, `shift` — зсув уліво на задану кількість позицій, `reverse` — зміна порядку елементів на протилежний, `middle` — знаходження

середнього елементу списку, `median` — знаходження медіани числового списку, `permutation` — перестановка, `subset` — підмножина, `delDuplicate` — видалення повторних входжень елементів.

5.6. Керування перебором з поверненням

5.6.1. Відтинання

Нехай потрібно обчислити двохступінчасту функцію, графік якої має вигляд

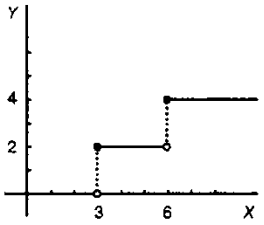


Рис. 5.2.

Залежність між x та $f(x)$ можна задати таким чином:

$$f(x)=\begin{cases}0, & x<3,\\ 2, & 3\le x<6,\\ 4, & 6\le x.\end{cases}$$

Для обчислення можна використати наступні правила Пролога:

```
f(X, 0) :- X < 3.          % Правило 1
f(X, 2) :- 3 =< X, X < 6.    % Правило 2
f(X, 4) :- 6 =< X.           % Правило 3
```

Проаналізуємо виконання наступного запиту:

```
?- f(1, Y), 2 < Y.
```

При спробі досягнення першої мети, `f(1, Y)`, змінна `Y` конкретизується значенням 0. Тому друга ціль приймає вигляд 2 < 0. Дана ціль недосяжна, а отже, а тому і весь запит завершується неуспіхом. Трасування виконання запиту наведене на рис. 5.3.

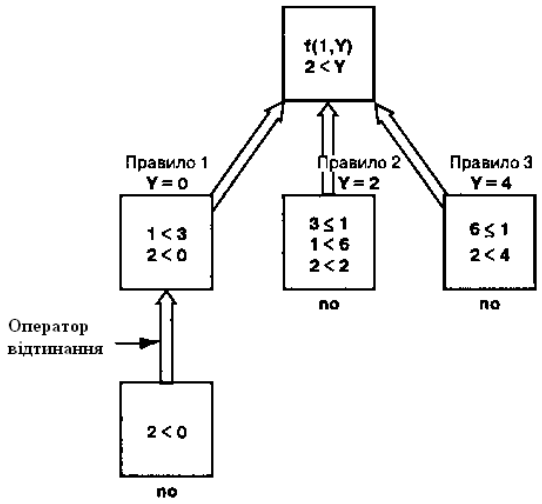


Рис. 5.3

Оскільки правила 1-3 є попарно несумісними, то не може бути досягнуто ціль більше одного разу. Але система Пролог не має про це інформацію. У прикладі, наведеному на рис. 5.3, ціль правила один буде досягнуто у точці, позначеній як "Оператор відтинання". Тому потрібно повідомити системі Пролог про те, що не потрібно виконувати безперспективний перебір інших варіантів. Це завдання вирішується із використанням оператора відтинання, який записується як "!" і вставляється між цілями як своєрідна *псевдоціль*. Після використання оператора відтинання, програма стає такою:

```
f(X, 0) :- X < 3, !.
f(X, 2) :- 3 =< X, X < 6, !.
f(X, 4) :- 6 =< X.
```

Тепер символ ! запобігає перебору з поверненням. Отримана версія програми є більш ефективною, оскільки пошук цілі по середній та правій гілкам вже не ведеться.

Припустимо, що зроблений наступний запит:

```
?- f(7, Y).
Y = 4
```

При пошуку розв’язку була виконана послідовна перевірка усіх трьох правил. При цьому після виконання першої перевірки 7 < 3 перевірка умови 3 =< 7 є зайвою. Те саме стосується цілей 7 < 6 та 6 =< 7. Тому відповідні зайві перевірки можуть бути усунуті. Це дає змогу отримати наступну версію програми.

```
f(X, 0) :- X < 3, !.
f(X, 2) :- X < 6, !.
f(X, 4).
```

Слід зазначити, що видалення операторів відтинань у останній програмі приводить до нової програми, яка вже не є еквівалентною до попередніх, оскільки здатна знаходити кілька розв’язків.

Опис механізму відтинань:

Назвемо батьківською ціль, яка узгоджується із головою правила, яке містить оператор відтинання. Якщо у якості цілі виявлений оператор відтинань, то ціль негайно досягається та при цьому у системі відбувається негайна фіксація результатів усіх виборів, зроблених з моменту виклику батьківської цілі до моменту, у який зустрівся оператор !. Усі альтернативи, які залишилися між батьківською ціллю та оператором відтинання, відкидаються.

Для прикладу розглянемо правило

```
н :- bl, b2, . ..,bm, !, ... , bn.
```

яке викликається ціллю `g`, яка узгоджується із ціллю `h`. До моменту виконання оператора ! система вже знайшла деякий розв’язок, який відповідає цілям `bl, b2, ..., bm`. Після виконання оператора відтинання ці поточні значення заморожуються, а усі можливі альтернативи відкидаються. Крім того, ціль `g` стає фіксованою. Усі спроби узгодити ціль `g` з головою якого-небудь іншого правила не проводяться.

Застосуємо правила механізму відтинань для аналізу наступного прикладу:

```
c :- p, q, r, !, s, t, u.
c :- v.
a :- b, c, d.
?- a.
```

Перебір з поверненням можливий для списку цілей `p, q`, але після досягнення ! усі альтернативи у цьому списку відкидаються. Альтернативне правило `c :- v` відносно `c` також відкидається. Але перебір з поверненням у межах списку цілей `s, t, u` усе ще можливий. Ціль `c` входить у правило `a`. Оператор відтинання впливає лише на ціль `c`. З позиції цілі `a` він залишається "непоміченим". Таким чином автоматичним перебір з поверненням у списку цілей `b, c, d` є активним, незважаючи на оператор відтинання. Вплив оператора відтинання на виконання програми проілюстровано на рис. 5.4.

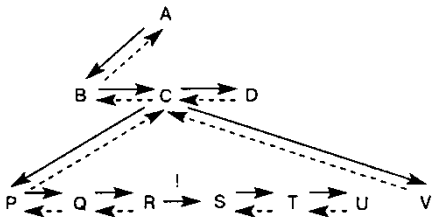


Рис. 5.4.

Ілюстрація прикладу:

```
c(X) :- p(X), !, s(X).
c(X) :- v(X).
a(X) :- b(X), c(X), d(X).
p(1).
p(2).
s(2).
v(3).
b(2).
b(3).
d(3).
```

Відповіддю на запит `a(X)` буде `X = 3`.

Якщо змінити правило `a(X)` на

```
a(X, Y) :- b(X), c(Y), d(Y).
```

то ціль `a(X, Y)` є недосяжною, ціль `a(X, X)` — досяжною.

5.6.2. Приклади використання оператора відтинання

1. Обчислення максимуму двох чисел.
   Правило обчислення максимуму двох чисел

```
max(X, Y, X) :- X >= Y.
```

```
max(X, Y, Y) :- X < Y.
```

з використанням оператора відтинання може бути переписано так

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y).
```

Модифіковане правило є коректним лише у випадку його використання у цілях `max(X, Y, Max)` із неконкретизованою змінною Max. Інакше можливою є помилка. Так, наприклад, ціль `max(3, 1, 1)` є досяжною. Вказане обмеження можна подолати із використанням наступної форми max:

```
max(X, Y, Max) :- X >= Y, !, Max = X; Max = Y.
```

2. Класифікація з розбиттям по категоріям.
Розглянемо класифікацію учасників турніру з тенісу. Результати зберігаються у вигляді фактів:

```
beat(tom, jim).
beat(ann, tom).
beat(pat, jim).
```

Необхідно визначити відношення `class(Player, Category)` для розбивки гравців по трьом категоріям:
`winner` — переміг у всіх зустрічах,
`fighter` — має як перемоги так і поразки,
`looser` — програв усі зустрічі.
Відповідне словесне формулювання має вигляд:
Якщо X кого-небудь переміг та X був ким-небудь переможений,
то X — `fighter`,
у протилежному випадку, якщо X когось переміг,
то X — `winner`,
в протилежному випадку, якщо X був ким-небудь переможений,
то X — `looser`.
Це розбиття можна описати з використанням відтинання наступним чином

```
class(X, fighter) :- beat(X, _), beat(_, X), !.
class(X, winner) :- beat(X, _), !.
class(X, looser) :- beat(_, X).
```

Виклик `class` є безпечним, якщо другий параметр неконкретизований.

Завдання. Написати процедуру `split(Numbers, Positives, Negatives)` розбиття списку на відповідні підсписки.

5.6.3. Недосяжна ціль fail

Твердження "щось не є істинним" можна виразити з використанням цілі `fail`, спроба досягнення якої завжди є невдалою.

Розглянемо запис на Пролозі твердження "Мері подобаються усі тварини крім змій".

Для цього можна переформулювати твердження таким чином:

Якщо X — змія, то твердження "Мері подобається X" не є істинним.

В протилежному випадку, якщо X — тварина, то Мері подобається X.

Останнє твердження на Пролозі запишеться так:

```
likes(mary, X) :- snake(X), !, fail.
likes(mary, X) :- animal(X).
```

Компактніше це правило можна записати так:

```
likes(mary, X) :- snake(X), !, fail; animal(X).
```

Це саме відношення можна записати із використанням оператора not.

```
likes(mary, X) :- animal(X), not(snake(X)).
```

5.7. Додаткові вбудовані предикати Прологу

5.7.1. Перевірка типу термів

Предикат `number(X)` приймає істинне значення, якщо `X` — число або змінна, значенням якої є число. Цей предикат призначений для захисту цілі `Z is X + Y` від недопустимих аргументів.

Інші подібні вбудовані предикати:

`var(X)` — перевірка того, чи є `X` неконкретизованою змінною.

`nonvar(X)` — виконується успішно, якщо `X` — не змінна або уже конкретизована змінна.

`atom(X)` — перевірка того, чи є `X` атомом.

`integer(X), float(X), number(X)` — перевірка належності `X` до відповідних числових множин.

`atomic(X)` — приймає істинне значення, якщо `X` є числом або атомом.

`compound(X)` — приймає істинне значення, якщо `X` є структурою.

Приклад. Написати правило для заміни усіх неконкретизованих елементів списку на найбільший серед його числових елементів.

```
max([X|L], MaxValue) :-
    not(number(X)), !,
    max(L, MaxValue).
max([X|L], MaxValue) :-
    max(L, MaxValue),
    MaxValue > X, !.
max([X|_], X).
replaceVar(InList, OutList) :- max(InList, MaxValue),
    repVar(InList, MaxValue, OutList).
repVar([], _, []).
repVar([X|L1],Value,[Value|L2]):- var(X), !,
```

```
    repVar(L1,Value,L2).
repVar([X|L1], Value, [X|L2]) :- repVar(L1,Value, L2).
```

5.7.2. Створення та декомпозиція термів

1. Предикат =..

Предикат =.. записується як інфіксний оператор і читається "юнів" (univ). Ціль

```
Term =.. L
```

є істинною, якщо L — список, що містить головний функтор Term, за яким йдуть його параметри (компоненти). Приклади:

```
?- f(a, b) =.. L.
L = [f, a, b]
?- T =.. [rectangle, 3, 5].
T = rectangle(3, 5)
?- Z =.. [p, X, f(X,Y)].
Z = p(X, f(X,Y))
```

Розглянемо програму маніпулювання геометричними об’єктами, які зберігаються у програмі у вигляді термів, функтори яких позначають типи фігур, а параметри задають розмір фігури, як показано нижче.

```
square(Side)
rectangle(Side1, Side2)
triangle(Sidel, Side2, Side3)
circle(R)
```

Однією з можливих операцій над фігурами може бути зміна їх розміру за допомогою відношення

```
resize(Figure, Factor, ResizedFigure),
```

де `Factor` — коефіцієнт, який вказує у скільки разів треба збільшити розмір. Традиційний спосіб опису відношення `resize` наведений нижче:

```
resize(square(A), F, square(Al)) :- Al is F*A.
resize(circle(R), F, circle(Rl)) :- Rl is F*R1.
resize(rectangle(A,B), F, rectangle(Al,Bl)):- Al is F*A,
    Bl is F*B.
...
```

Такий підхід є громіздким для великої кількості фігур. Цю незручність можна усунути з використанням предикату =.. наступним чином:

```
resize(Fig, F, Fig1) :-
    Fig =.. [Type | Parameters],
```

```
    multiplylist(Parameters, F, Parameters1),
    Fig1 =.. [Type | Parameters1].
multiplyList([], _, []).
multiplyList([X | L], F, [X1 | L1]) :-
    X1 is F*X, multiplyList(L, F, L1).
```

Предикат =.. можна використовувати для програмного формування цілей, які потім можна виконувати із використанням call. Наступний предикат call2 можна використовувати для виклику двохмісних предикатів із заданими аргументами:

```
call2(Predicate, X, Y) :- Goal =.. [Predicate, X, Y],
    call(Goal).
```

Приклад виклику предиката call2:

```
?- call2(likes, john, X).
X = mary ;
X = meat ;
X = fish.
```

2. Предикати `functor` та `arg`

Предикат

```
functor(Term, F, N)
```

є істинним, якщо `F` — головний функтор терма `Term`, арність якого рівна `N`.

Ціль

```
arg(N, Term, A)
```

є істинною, якщо `A` — `N`-й параметр у термі `Term` за умови, що параметри термів-структур нумеруються зліва направо, починаючи з 1.

Приклад:

```
?- functor(t(f(X), X, t), Fun, Arity).
Fun = t
Arity = 3
?- arg(2, f(X, t(a), t(b)), Y).
Y = t(a)
?- functor(D, date, 3),
arg(1, D, 29),
arg(2, D, June),
arg(3, D, 1982) .
D = date( 29, June, 1982)
```

З використанням предикатів `functor` та `arg` можна моделювати масиви. Ціль

```
functor( A, f, 100)
```

створює структуру із 100 елементами:

```
A = f (_,_,_,... )
```

Аналогом присвоєння A[60] = 1, яке характерне для процедурних мов, може бути наступна ціль Пролога:

```
arg(60, A, 1)
```

У результаті виконання цієї цілі, структура A конкретизується таким чином:

```
A = f(_,...,_,1,_,...,_) % 60-й елемент рівний 1
```

Аналогом "процедурного" оператора `X = A[60]` буде

```
arg(60, A, X)
```

Проте у Пролозі відсутній простий аналог процедурного оновлення елементів масиву типу `A[5] = A[5]+1`.

5.7.3. Операції з базою даних

Базою даних у Пролозі вважається сукупність усіх визначених у Пролог-програмі фактів та правил. У мові Пролог передбачена можливість модифікації цієї бази під час виконання програм. Для цього призначені предикати `asserta, assertz, retract`.

Ціль

```
asserta(C)
```

завжди досягається і крім того заносить фразу (факт або правило) `C` у початок бази даних. Ціль `assertz(C)` аналогічна за винятком того, що `C` заноситься у кінець бази.

Ціль

```
retract(C)
```

виконує протилежну дію: видаляє фразу Пролога, яка узгоджується із `C`.

Згідно до стандарту `asserta, assertz, retract` можна застосовувати лише до предикатів, які описані як "динамічні". Динамічні предикати оголошуються за допомогою директиви `:-dynamic` із вказівкою їх арності. Наприклад, директива

```
:- dynamic parent/2, male/1, female/1
```

описує динамічний двохмісний предикат `parent` та одномісні предикати `male, female`.

Предикат `retract` є недетермінованим: за допомогою єдиної цілі `retract` можна видалити цілий набір фраз Пролога.

Приклад. Нехай у Пролог-програмі визначені факти

```
fast(ann).
slow(tom).
slow(pat).
```

До цієї програми можна додати нове правило `faster:`

```
?- asserta((faster(X,Y) :- fast(X), slow(Y))).
yes
?- faster(A, B).
A = ann
B = tom
?- retract(slow(X)).
X = tom;
X = pat;
no
?- faster(ann, _).
no
```

Розглянемо приклад застосування предикатів `assertz` та `retract` для знаходження списку унікальних імен батьків, які є першими аргументами відношення `parent`.

```
:-dynamic name/1.
getParents(_) :- parent(X, _), not(name(X)),
        assertz(name(X)), fail.
getParents(List) :- extractNames(List).
extractNames([X|Names]) :- retract(name(X)), !,
        extractNames(Names).
extractNames([]).
```

Розглянемо приклад формування таблиці множення:

```
:- dynamic product/3.
maketable :-
    L = [0,1,2,3,4,5,6,7,8,9] ,
    select(X, L, _), % Вибрати 1-й множник
    select(Y, L, _), % Вибрати 2-й множник
    Z is X*Y,
    assert(product(X,Y,Z)),
    fail.
```

Для формування таблиці треба виконати запит

```
?- maketable.
```

Після цього можна, наприклад, шукати числа, добуток яких рівний 24:

```
?- product(X, Y, 24).
X = 3,
```

```
Y = 8 ;
X = 4,
Y = 6 ;
X = 6,
Y = 4;
.....
```

Розглянемо приклад знаходження списку дітей для кожного із батьків:

```
extract(X, L):- retract(name(X-L)), !.
extract(_, []).
childList(_) :- parent(X, Y), extract(X, L),
        assertz(name(X-[Y|L])), fail.
childList(List) :- extractNames(List).
?- childList(L).
L = [tom-[ann, liz, pat, jane], jim-[mary], jane-[mary],
pat-[steve, pam]].
```

Для видалення усіх фактів і правил, які узгоджуються із Goal, використовується предикат

```
retractall(Goal)
```

Приклад застосування: `?- retractall(name(_)).`
Перевагою предикату `retractall` над `retracta` є те, що його можна використовувати для видалення динамічних правил.

5.7.4. Генерація списків. Предикати bagof, setof, findall

Ціль `bagof(X, P, L)` генерує список `L`, який складається з усіх об'єктів `X`, таких, що ціль `P` досягається.
Наприклад, для бази даних про родинні зв'язки можна знайти дітей Тома так:

```
?- bagof(X, parent(tom, X), L).
L = [jane, pat, liz, ann].
```

Якщо ім'я батька не вказано, то для кожного з батьків буде сформовано список дітей:

```
bagof(X, parent(Y, X), L).
Y = jane,
L = [mary] ;
Y = jim,
L = [mary] ;
Y = pat,
L = [pam, steve] ;
```

```
Y = tom,
L = [jane, pat, liz, ann].
```

Можна також вказати, що потрібно сформувати один список без групування. Для цього використовується оператор "^". Наприклад, якщо потрібно отримати список усіх батьків, то можна вказати пошуковій системі Пролог, що ім'я дитини не має значення:

```
?- bagof(Y, X^parent(Y, X), L).
L = [tom, tom, tom, tom, jim, pat, jane, pat].
```

Якщо один і той самий об'єкт знаходиться кілька разів, то він дублюється у списку. Якщо відсутній розв'язок, то ціль `bagof` не досягається.

Предикат `setof(X, P, L)` аналогічний до `bagof` за винятком того, що всі дублікати видаляються і результуючий список впорядковується (відповідно до вбудованого предиката `@<`). Наприклад

```
?- setof(Y, X^parent(Y, X), L).
L = [jane, jim, pat, tom].
```

Предикат `findall(X, P, L)` також продукує список об'єктів, які відповідають предикату `P`. Він відрізняється від `bagof` тим, що у список збираються усі об'єкти `X` незалежно від значення інших змінних. Наприклад

```
?- findall(Y, parent(Y, _), L).
L = [tom, tom, tom, tom, jim, pat, jane, pat].
```

Тобто `findall` еквівалентний до `bagof` з параметром ^. Якщо жоден об'єкт `X` не відповідає предикату `P`, то створюється порожній список і ціль досягається.

5.7.5. Предикати maplist та forall

Виклик предиката `maplist(:Goal, ?List)` завершується успіхом, якщо ціль `Goal` може бути успішно застосована для усіх елементів списку `List`.

Приклад. Нехай у програмі визначений предикат

```
positive(X) :- X > 0.
```

Тоді для перевірки того, чи є додатними усі елементи списку, можна використати предикат `maplist` наступним чином:

```
?- maplist(positive, [1, 3, 2, 5]).
true.
```

Виклик предиката `maplist(:Goal, ?List1, ?List2)` завершується успіхом, якщо ціль `Goal` може бути успішно застосована для усіх пар відповідних елементів списків `List1` та `List2`.

Приклад. Нехай у програмі визначений предикат

```
inc(X, Y) :- number(X), !, Y is X + 1.
inc(X, X).
```

Тоді для збільшення на 1 усіх числових елементів списку можна використати предикат `maplist` наступним чином:

```
?- maplist(inc, [2, a, -1, 5], L).
L = [3, a, 0, 6].
```

Виклик предиката `maplist(:Goal, ?List1, ?List2, ?List3)` завершується успіхом, якщо ціль `Goal` може бути успішно застосована для усіх трійок відповідних елементів списків `List1, List2` та `List3`.

Наприклад, для знаходження поелементних сум двох списків можна використати ціль вигляду:

```
maplist(plus, [1,2], [3,4], L).
L = [4, 6].
```

Предикат `forall(:Cond, :Action)` перевіряє ціль `:Action` для усіх результатів виклику цілі `:Cond`.

Приклад. Розглянемо предикат знаходження суми елементів числового списку:

```
:-dynamic sum/1.
sumList(L, Sum) :- assertz(sum(0)),
   forall(select(X, L, _),(retract(sum(S)), T is S + X, assert(sum(T)))),
   retract(sum(Sum)).
```

Результат роботи:

```
?- sumList([2,3,7], S).
S = 12.
```

5.8. Застосування мови Пролог для розв'язування задач штучного інтелекту

5.8.1. Задача про Ханойську вежу

Задача про Ханойську вежу є класичним прикладом використання декомпозиційного методу розв'язування задач. На рис. 5.5 наведено задачу у випадку $n = 3$.

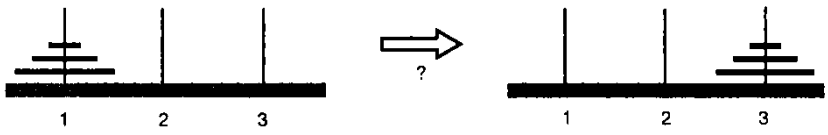


Рис. 5.5. Задача про Ханойську вежу

На вісь 1 нанизані n різних дисків у порядку спадання їх розмірів. Потрібно перенести усі диски на вісь 3, використовуючи при потребі вісь 2 у якості допоміжної осі. Дозволяється одночасно переносити лише один диск. При цьому не можна класти більший диск на менший.

Розв'язок задачі можна отримати за наступним алгоритмом:

1. Перенести верхні $n-1$ диски з першого стержня на другий, використовуючи третій, як допоміжний.

2. Перекласти найбільший стержень з першої на третю вісь.
3. Перенести $n-1$ диски з другого стержня на третій, використовуючи перший, як допоміжний.

Реалізація на Пролозі:

```
hanoiTower(N):-
        move(N, left, middle, right).

writeList([]) :- nl, !.
writeList([H|T]) :- write(H), write(' '), writeList(T).

showMove(From, To, Number):-
   writeList(['move disk', Number, 'from', From, 'to', To]).

move(1, From, _, To):-
        showMove(From, To, 1), !.
move(N, From, Additional, To):-
        N1 is N - 1,
        move(N1, From, To, Additional),
        showMove(From, To, N),
        move(N1, Additional, From, To).
```

5.8.2. Задача про пошук у лабіринті

Потрібно знайти шлях, який веде від заданого початкового положення до виходу із лабіринту. Розглянемо приклад плану будинку, наведеного на рис. 5.6. Потрібно знайти вихід із заданої кімнати, наприклад g, назовні.

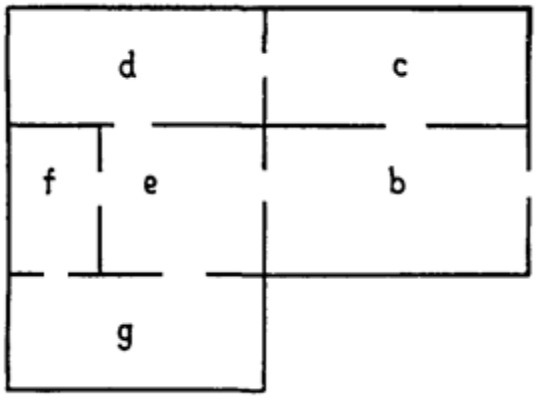


Рис. 5.6. План будинку

Задамо фактами двомісне відношення door, яке вказує на кімнати, між якими є двері (door(b, outside) позначає зовнішні двері).

```
door(b, outside).
door(b, c).
door(b, e).
door(d, c).
door(d, e).
door(f, e).
door(g, e).
door(f, g).
```

При розробці Пролог-програми потрібно врахувати, що двері є симетричними. Крім того для зручності можна вважати outside додатковою кімнатою.

Розглянемо програму із використанням списків. Ключовим предикатом є предикат

path(Start, Finish, Banned, Visited),

який дає змогу знайти шлях від початкової до кінцевої кімнати, який не проходить через жодну із кімнат двічі. Для уникнення зациклювань використовується список "заборонених" кімнат BannedRooms. Знайдений шлях записується у список VisitedRooms.

```
path(Start, Start, _, []) :- !.
path(Start, Finish, Banned, [Start|Visited]) :-
    (door(Start, Room); door(Room, Start)),
    \+member(Room, Banned),
    path(Room, Finish, [Start| Banned], Visited).
```

Основний предикат знаходження розв’язку:

```
exitSearch(Start, Visited) :-
    path(Start, outside, [], Visited).
```

Приклад запиту:

```
?- exitSearch(g, R).
R = [g, e, b] ;
R = [g, e, d, c, b] ;
R = [g, f, e, b] ;
R = [g, f, e, d, c, b] ;
false.
```

6. ПРОДУКЦІЙНІ МОДЕЛІ

6.1. Загальна характеристика продукційних моделей

Продукційною називається *модель знань*, в якій база знань складається із *продукцій,* тобто правил "Якщо *A*, тоді *B*" [6, 9]. Термін "продукція" був введений американським математиком Е. Постом. Як і логічні, продукційні моделі насамперед концентруються на дедуктивному виведенні, але є менш формалізованими і тому більш наочними і зручними. На сучасному етапі найбільш вживаним формалізмом для задання знань в експертних системах є саме продукційні моделі.

Приклад 1. Розглянемо інформаційну експерту систему для зоологічного парку, яка повинна на основі заданого опису деякої тварини відповісти на питання, що це за тварина. Продукції такої системи можуть мати вигляд:

Якщо тварина є хижаком, має жовто-коричневий колір та темні смуги, то це — тигр.

Якщо тварина є птахом, не літає, плаває, має чорно-білий колір, то це — пінгвін.

Наведені продукції описують імплікації (з істинності *A* випливає істинність *B*).

Приклад 2. Нехай ідеться про систему керування деяким технічним процесом. Продукції можуть мати такий вигляд:

Якщо температура перевищує 200 градусів, ввімкнути систему охолодження.

Продукції задають *умовні операції* (якщо справедлива умова *A*, виконати дію *B*).

Приклад 3. Типовими продукціями є правила підстановки (замінити *A* на *B*). Прикладом можуть послугувати правила підстановки формальних граматик.

База знань, організована як сукупність продукцій, разом з механізмом керування продукціями, утворює продукційну систему.

Продукційною моделлю називається представлення знань у вигляді *сукупності продукцій. Продукція* визначається як вираз вигляду*:*

$$(i)\ Q;\ P;\ A \Rightarrow B;\ N,$$

де (*i*) — ім’я продукції, за допомогою якого вона виділяється серед усієї множини продукцій;

Q — сфера застосування продукції; предметна область, до якої вона відноситься;

P — умова застосування продукції;

$A \Rightarrow B$ — *ядро продукції;* інтерпретація залежить від конкретної ситуації; часто ядро інтерпретується як *"якщо маєш A, зроби B";*

N — *післяумова продукції,* постулює процедури, які необхідно виконати після реалізації ядра.

Як приклад можна навести такі продукції:

(М-1) Магазин; у покупця є гроші; якщо покупець хоче купити річ, він платить в касу; змінити базу даних про товари.

Як приклад можна навести таку продукцію:

(БС-29) Банківська сфера; клієнт має рахунок у банку на достатню і якщо клієнт бажає зняти певну суму грошей, то він повинен написати ордер; внести зміни до бази даних.

Перспективним видається поєднання продукційних моделей із фреймовими і мережевими. Наприклад, самі знання можна описувати на основі семантичних мереж, а операції над ними задавати як продукційні, що зумовлюють заміну одного фрагмента мережі на інший. З іншого боку самі продукції можна включати до семантичних мереж і фреймів Методи класу можна розглядати як продукції — особливо у випадку, якщо вони є демонами, що запускаються при виконанні певних умов. Як типові продукції можна розглядати зв'язки функціональної мережі, які задають алгоритми обчислення одних величин через інші.

Популярність продукційних моделей зумовлена такими рисами [9]:

1. Більшість людських знань може бути записана у вигляді продукцій.
2. Модульність; продукції, за рідким винятком, є незалежними, і внесення або вилучення окремих продукцій, як правило, не приводить до змін в інших продукціях.
3. У разі необхідності продукційні системи можуть реалізувати будь-які алгоритми.
4. Продукції можуть бути порівняно легко розподілені за сферами застосування.
5. Перспективним є об'єднання продукційних систем і мережевих завдань.
6. Продукції можуть працювати паралельно та асинхронно, тому їх зручно реалізовувати на основі багатопроцесорних комплексів.

До основних проблем, пов'язаних з використанням продукційних систем, відносять складність перевірки несуперечливості та коректності функціонування.

6.2. Продукції та мережі виведення

Системи продукцій часто буває зручно зображати у вигляді графів, які називаються *мережами виведення*.

Мережі виведення будуються так. Вершини графу відповідають умовам і діям, що входять до лівих і правих частин продукцій. Якщо в продукційні системі є продукція $A \Rightarrow B$, то від вершини A до вершини B йде орієнтована дуга. Якщо ж є продукція A, $B \Rightarrow C$, то дуги (AC) та (BC) пов'язані відношенням "I" (кон'юнкції)

Так, для продукційної системи

$A \Rightarrow C$

$B \Rightarrow C$

$B, F \Rightarrow L$

$F \Rightarrow Q$

$D, C \Rightarrow G$

мережа виведення має такий вигляд:

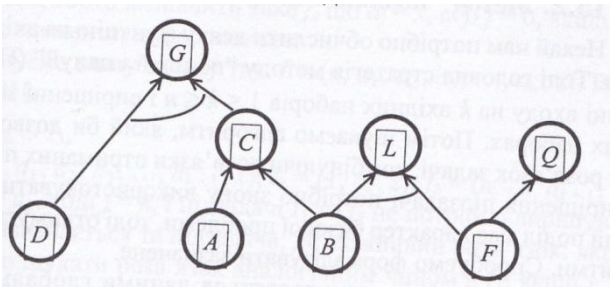


Рис. 6.1. Продукційна мережа виведення

На основі аналізу мереж виведення стає наочнішим і очевиднішим те, що відповідь на запит користувача можна отримати шляхом зведення задач до підзадач. Так, щоб вирішити задачу *G*, необхідно вирішити і задачу *D*, і задачу *C*, розв'язування якої в свою чергу, потребує вирішення або *A*, або *B*. Таким чином, мережа виведення для кожної задачі є структурою, близькою до традиційних І-АБО-графів.

6.3. Типова схема роботи експертної системи па базі продукцій

На рис. 6.2. показана узагальнена схема типової експертної системи, побудованої на основі продукційних правил:

Рис. 6.2. Типова схема роботи продукційної системи

До робочої пам'яті заносяться факти, що задаються користувачем, а також його запити. Логічний блок зіставляє цю інформацію з правилами, які зберігаються в базі знань, і застосовує ті правила, які можуть бути зіставлені із вмістом робочої пам'яті. Наприклад, якщо в базі знань є продукції $A \Rightarrow B$ та $B \Rightarrow C$, то вони можуть бути виконані, коли користувач ввів дані про істинність факту *A*.

6.3. Пряме та зворотне виведення

Існують дві основні стратегії логічного виведення в продукційних системах: пряме та зворотне виведення.

Пряме виведення на основі наявних правил передбачає аналіз:

1) наслідків з фактів,

2) наслідків з наслідків і т. д.

Процес продовжується, поки не буде встановлено істинність або хибність запиту користувача.

Наприклад, нехай є продукції $A \Rightarrow B$ та $B \Rightarrow C$; користувач повідомив про істинність факту *A* і спитав про істинність *C*. Ланцюжок прямого виведення матиме вигляд: з *A* виводиться *B*, з *B* виводиться *C*, тому *C* — істинне.

Пряме виведення не вважається ефективним через те, що можна отримати чимало безперспективних продукцій, і, відповідно, породжується дуже багато зайвих проміжних результатів. Експертні системи на основі прямого виведення використовуються, як правило, при плануванні і прогнозуванні.

При зворотному виведенні логічний блок починає роботу із запиту користувача, тобто з твердження, яке перевіряється. Розглядається одне з правил, на основі яких можна вивести це твердження, після чого перевіряється істинність лівої частини цього правила. Процес повторюється до тих пір, доки він не дійде до фактів, які вважаються істинними.

У нашому прикладі ланцюжок зворотного виведення матиме вигляд: "*C* виводиться з *B*, *B* виводиться з *A*, *A* істинне, отже, і *C* істинне".

Як і у випадку прямого виведення, зворотне виведення може вимагати повторень у разі невдалої спроби, і тому є стратегією перебору.

Зворотне виведення тісно пов'язано з резолюцією, зокрема лінійною, і механізмом виконання програм Прологу.

6.4. Типові дисципліни виконання продукцій

Передусім ядра продукцій прийнято поділяти на детерміновані та недетерміновані. У детермінованих продукціях при виконанні лівої частини завжди виконується і права. У недетермінованих продукціях права частина при виконанні лівої виконується необов'язково. Можливі, наприклад такі інтерпретації недетермінованих продукцій: "Якщо *A*, то можливе *B*"; або "Якщо *A*, то *B* з певною вірогідністю".

Існують різні режими керування виконанням готових продукцій, тобто продукцій, ліві частини яких істинні, і, відповідно, ці продукції можуть бути виконані негайно. Можна виокремити такі режими:

режим негайного виконання;

режим формування конфліктного набору.

У *режимі негайного виконання*, перша знайдена продукція *виконується невідкладно*.

У *режимі формування конфліктного набору* знайдені готові продукції не виконуються відразу, а включаються до *конфліктного набору* — списку продукцій, готових до виконання. Лише після завершення формування конфліктного набору відбувається вирішення конфлікту, тобто вибір зі списку готових продукцій якоїсь однієї. Інша назва конфліктного набору "фронт готових продукцій".

Розглянемо деякі проблеми, пов'язані з конфліктними наборами. Нехай маємо продукції:

1) $A \Rightarrow B$

2) $A \Rightarrow C$

і *A* істинне.

Тоді обидві продукції готові до виконання і утворюють конфліктний

Якщо ядра продукцій інтерпретуються як імплікації, проблема є менш гострою. Продукційна система є чисто декларативною. Після застосування 1), тобто виведення *B*, *A* залишається істинним, і ми завжди можемо використати продукцію 2). Таким чином, порядок застосування продукційних правил може впливати лише на час роботи системи, але не на істинність висновків.

Зовсім інша ситуація виникає, якщо ядра продукцій інтерпретуються як "виконати таку-то дію". Тоді після виконання 1) ситуація може змінитися: *A* може перестати бути істинним, і тоді продукція 2) може стати недосяжною. Тому, якщо насправді треба було вибирати *C*, вибір *B* стає помилкою, яку не завжди можна виправити.

Існують різні стратегії вирішення конфліктів, а також обмеження при включенні готових продукцій до конфліктного набору.

Розрізняють також централізоване і децентралізоване керування продукціями.

За централізованого керування продукційна система має центр керування, який вирішує, коли має спрацювати та чи інша продукція.

За децентралізованого керування кожна продукція може самостійно прийняти рішення про свій запуск. Один з типових механізмів, який застосовується при цьому, є механізм "класної дошки". Таку назву має спеціальна область пам’яті, доступна для різних продукцій. На ній продукції що працюють паралельно, знаходять інформацію, що може ініціювати запуск. Туди ж вони пишуть інформацію, що може виявитись корисною для інших продукцій.

6.5. Основні стратегії вирішення конфліктів у продукційних системах

Очевидно, що найзагальнішим способом керування системами продукцій є комплексне планування її роботи, тобто прийняті рішень на основі аналізу всіх можливих дій та їх наслідків з погляду мети, що стоїть перед системою. Але зрозуміло, що вказаний принцип рідко може бути застосований у повному обсязі, насамперед через експоненційне зростання кількості можливих варіантів розвитку подій.

Існують стратегії керування вирішенням конфліктів, мета яких — намагатися уникнути експоненційного вибуху. Подібні стратегії здебільшого можуть бути охарактеризовані як евристичні. В [9] описані деякі стратегії керування виконанням продукцій.

Принцип "стосу книг". Ґрунтується на ідеї, що найкориснішою є продукція, яка використовується найчастіше (так само, як у стосі книг ті, що найчастіше користуються попитом, як правило, опиняються зверху. Відповідно, з фронту готових продукцій вибирається продукція з найбільшою частотою використання. У [1] відмічено, що керуватися принципом "стосу книг" доцільно, якщо продукції відносно незалежні одна від одної, наприклад, коли кожна з них являє собою правило "ситуація $A \Rightarrow$ дія B".

Принцип "найдовшої умови". З фронту готових продукцій вибирається та, для якої стала справедливою найдовша умова виконання. Підставою для цього принципу є міркування про те, що часткові правила, застосовні до вузького класу ситуацій, є важливішими, ніж загальні правила для широкого класу ситуацій. Наприклад, нехай є дві продукції:

(1) Якщо A — птах, то A літає.

(2) Якщо A — птах і A — пінгвін, то A не літає.

Якщо відомо, що A — пінгвін, то за принципом найдовшої умови слід вибрати (2).

У [9] відзначається, що принцип найдовшої умови доцільно застосовувати, якщо продукції прив’язані до типових ситуацій, зв’язаних відношенням "часткове — загальне". До цього можна додати, що цей принцип заслуговує на особливу увагу при аналізі винятків. Так, у нашому прикладі правило (2) є винятком з правила (1); якби ми були впевнені, що (1) не має винятків, ми могли б застосувати це загальне правило у будь-якій ситуації.

ЛІТЕРАТУРА

1. Глибовець М.М., Олецький О.В. Системи штучного інтелекту. — К.: КМ Академія, 2002. — 366 с.
2. Рассел С., Норвиг П. Искусственный интеллект. Современный поход. — М.: Вильямс, 2006. — 1408 с.
3. Люгер Дж. Искусственный интеллект. Стратегии и методы решения сложных проблем. — М.: Вильямс, 2003. — 864 с.
4. Смолин Д.В. Введение в искусственный интеллект: конспект лекций. — М.: ФИЗМАТЛИТ, 2004. — 208 с.
5. Братко И. Алгоритмы искусственного интеллекта на языке Пролог. — М.: Вильямс, 2004. — 640 с.
6. Девятков В.В. Системы искусственного интеллекта. — М.: Изд-во МГТУ им. Баумана, 2001. — 352 с.
7. Субботін С.О. Подання й обробка знань у системах штучного інтелекту та підтримки прийняття рішень. Запоріжжя: ЗНТУ, 2008. — 341 с.
8. Представление и использование знаний / Под ред. Уэно Х., Исидзука М. — М.: Мир, 1989. — 220 с.
9. Искусственный интеллект: Справочник: В 3-х т. — М.: Радио и связь, 1990.
10. Нильсон Н. Принципы искусственного интеллекта. — М.: Радио и связь, 1985. — 376 с.
11. Руденко О. Г., Бодянський Є. В. Штучні нейронні мережі: Навчальний посібник. — Харків: ТОВ "Компанія СМІТ", 2006. — 404 с.