

**НАЦІОНАЛЬНА АКАДЕМІЯ УПРАВЛІННЯ
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК
КАФЕДРА ІНТЕЛЕКТУАЛЬНИХ СИСТЕМ**

БАКЛАН І.В.

**ЕКСПЕРТНІ СИСТЕМИ
КУРС ЛЕКЦІЙ**

КИЇВ — 2012

НАЦІОНАЛЬНА АКАДЕМІЯ УПРАВЛІННЯ
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК
КАФЕДРА ІНТЕЛЕКТУАЛЬНИХ СИСТЕМ

Баклан І.В.,
ЕКСПЕРТНІ СИСТЕМИ
КУРС ЛЕКЦІЙ

КИЇВ-2012

ББК 813

УДК 004.8

Б996

Затверджено на Раді факультету комп'ютерних наук від «28» серпня 2012 р. протокол N1/280812.

Рецензенти: доктор технічних наук, професор кафедри математичних методів системного аналізу ННК ІПСА НТУ КІП Бідюк П.І. та кандидат технічних наук, доцент кафедри інформатики КНУТД Голубєв Л.П.

Баклан І.В. Експертні системи. Курс лекцій /Навчальний посібник. - К.: НАУ, 2012. - 132 с.

Дисципліна "Експертні системи" є вступною до цілого циклу навчальних курсів, присвячених проблемам побудови інтелектуальних систем. Основним завданням даної дисципліни є вивчення принципів побудови систем, що використовують експертні знання для розв'язку важко формалізованих завдань, що традиційно ставляться до компетенції людини-експерта. Лекційний курс "Експертні системи" розрахований на вивчення протягом одного семестру. Рекомендується студентам 5 курсу (першого року магістратури) за напрямом «Системні дослідження та кібернетика», спеціальність «Системи та методи прийняття рішень».

© **І.В.Баклан**

© **Національна академія управління**

Зміст

ВСТУП.....	8
Тема 1. Визначення експертної системи.....	9
1.1. Що таке експертний аналіз.....	9
1.2. Основні характеристики експертних систем.....	10
1.3. Основні функції експертних систем.....	11
1.3.1. Придбання знань.....	11
1.3.2. Подання знань.....	12
1.3.3. Керування процесом пошуку розв'язку.....	13
1.3.4. Роз'яснення ухваленого рішення.....	14
Вправи.....	15
Тема 2. Подання знань.....	16
2.1. Знання й дані.....	16
2.2. Моделі подання знань.....	18
2.2.1. Продукційна модель.....	18
2.2.2. Семантичні мережі.....	18
2.2.3. Фрейми.....	19
2.2.4. Формальні логічні моделі.....	20
2.3. Висновок на знаннях.....	20
2.3.2. Методи пошуку в глибину й завширшки.....	22
2.4. Об'єкто-орієнтований підхід, об'єднаний із правилами.....	22
2.5. Робота з нечіткістю.....	23
2.5.1. Основи теорії нечітких множин.....	23
Вправи.....	25
Тема 3. Інженерія знань.....	26
.....	26
3.1.1. Поле знань.....	26
3.1.2. "Піраміда" знань.....	27
3.2. Стратегії одержання знань.....	28
3.3. Теоретико-методичні аспекти добування й структурування знань.....	30
3.3.1. Психологічний аспект.....	30
3.3.2. Контактний шар.....	30
3.3.2.1. Стать й вік.....	31
3.3.2.2. Особистість.....	31
3.3.2.3. Темперамент.....	31
3.3.2.4. Мотивація.....	31
3.3.3. Процедурний шар.....	31
3.3.3.1. Ситуація спілкування.....	31
3.3.3.2. Устаткування.....	32
3.3.3.3. Професійні прийоми аналітика.....	32
3.3.4. Когнітивний шар.....	33
3.3.4.1. Когнітивний стиль.....	33
3.3.4.2. Семантична репрезентативність.....	34
3.3.5. Лінгвістичний аспект.....	34
3.3.5.1. "Загальний код".....	34
3.3.5.2. Понятійна структура.....	35
3.3.5.3. Словник користувача.....	35
3.3.6. Гносеологічний аспект.....	35
3.3.6.1. Внутрішня погодженість.....	36

3.3.6.2. Системність.....	36
3.3.6.3. Об'єктивність.....	37
3.3.6.4. Історизм.....	37
3.3.7. Методологія процесу одержання нового знання.....	37
3.3.7.1. Е_1: опис й узагальнення фактів.....	37
3.3.7.2. Е_2: установлення зв'язків.....	37
3.3.7.3. Е_3: побудова моделі.....	38
3.3.7.4. Е_4: пояснення й проорокування явищ.....	38
3.4. Методи практичного добування знань.....	39
Тема 4. Практикум з інженерії знань.....	41
4.1. Практикум по інженерії знань	41
4.1.1. Текстологічні методи.....	41
4.1.1.1. Розуміння тексту.....	41
4.1.1.2. Значеннєва структура тексту.....	42
4.1.1.3. Алгоритм добування знань із тексту.....	43
4.1.2. Комунікативні методи	43
4.1.2.1. Пасивні методи.....	43
4.1.2.1.1. Спостереження.....	43
4.1.2.1.2. Аналіз протоколів "думок у голос".....	44
4.1.2.1.3. Лекції.....	44
4.1.2.2. Активні індивідуальні методи.....	45
4.1.2.2.1. Анкетування.....	45
4.1.2.2.2. Інтерв'ю.....	45
4.1.2.3. Активні групові методи.....	48
4.1.2.3.1. "Круглий стіл".....	48
4.1.2.3.2. "Мозковий штурм".....	48
4.1.2.3.3. Експертні ігри.....	49
4.1.2.3.4. Ігри з експертом.....	49
4.1.2.3.5. Рольові ігри в групі.....	50
4.1.2.3.6. Ігри із тренажерами.....	50
4.1.2.3.7. Комп'ютерні експертні ігри.....	50
Тема 5. Філософія CLIPS - C Language Integrated Production System.....	52
5.1. Введення до CLIPS.....	52
5.2. Історія створення CLIPS.....	52
5.3. Робота з CLIPS.....	53
5.4. Синтаксис визначень.....	54
Тема 6. Огляд можливостей системи CLIPS.....	55
6.1. Основні елементи мови.....	55
6.1.1. Типи даних.....	55
6.1.2. Функції.....	56
6.1.3. Конструктори.....	57
6.2. Абстракції даних.....	57
6.2.1. Факти.....	57
6.2.1.1. Упорядковані факти.....	58
6.2.1.2. Неупорядковані факти.....	58
6.2.1.3. Ініціалізація фактів.....	59
6.2.2. Об'єкти.....	59
6.2.2.1. Ініціалізація об'єктів.....	59
6.2.3. Глобальні змінні.....	59
6.3. Подання знань.....	60

6.3.1. Евристичні знання.....	60
6.3.2. Процедурні знання.....	60
6.3.2.1. Функції.....	60
6.3.2.1.1. Родові функції.....	60
6.3.2.1.2. Оброблювачі повідомлень.....	61
6.3.2.1.3. Модулі.....	61
6.4. Об'єкто-орієнтовані можливості CLIPS.....	61
6.4.1. Відмінності COOL від інших об'єкто-орієнтованих мов.....	61
6.4.2. Основні можливості ООП.....	61
6.4.3. Запити й набори об'єктів.....	62
Тема 7. CLIPS. Факти.....	63
7.1. Факти в CLIPS.....	63
7.2. Робота з фактами.....	64
7.2.1. Конструктор deftemplate.....	64
7.2.2. Конструктор deffacts.....	67
7.2.3. Функція assert.....	68
7.2.4. Функція retract.....	69
7.2.8. Функція modify.....	70
7.2.7. Функція assert-string.....	71
7.2.8. Функція fact-existp.....	72
7.2.9. Функції для роботи з неупорядкованими фактами.....	72
7.2.10. Функції збереження й завантаження списку фактів.....	73
Тема 8. CLIPS. Правила	75
8.1. Створення правил. Конструктор defrule.....	75
8.2. Основний цикл виконання правил.....	77
8.3. Властивості правил.....	78
8.3.1. Властивість salience.....	78
8.3.2. Властивість auto-focus.....	78
8.4. Стратегія дозволу конфліктів.....	78
8.4.1. Стратегія глибини.....	79
8.4.2. Стратегія широчини.....	79
8.4.3. Стратегія спрощення.....	79
8.4.4. Стратегія ускладнення.....	79
8.4.5. Стратегія LEX.....	79
8.4.6. Стратегія MEA.....	80
8.4.7. Випадкова стратегія.....	80
8.5. Синтаксис LHS правила.....	80
8.5.1. Зразок (pattern PC).....	81
8.5.1.1. Символьні обмеження.....	81
8.5.1.2. Групові символи для простих і складових полів.....	82
8.5.1.3. Змінні, пов'язані із простими й складеними полями.....	83
8.5.1.4. Єднальні обмеження.....	84
8.5.1.5. Предикатні обмеження.....	85
8.5.1.6. Обмеження, що повертають значення.....	86
8.5.1.7. Зіставлення зразків з об'єктами.....	86
8.5.2. Автоматичне додавання і перегруповування умовних елементів.....	86
8.5.2.1. Безумовні правила.....	87
8.5.2.2. Використання елементів test та not перед and.....	87
8.5.2.3. Використання елемента not перед test.....	87
8.5.2.4. Використання елемента not перед or.....	88

8.5.2.5. Зауваження про автоматичне додавання й перегрупування умовних елементів	88
8.6. Команди й функції для роботи із правилами	88
8.6.1. Перегляд і видалення існуючих правил	88
8.6.2. Збереження правил	90
8.6.3. Запуск і зупинка програми	91
8.6.4. Перегляд плану рішення задачі	92
8.6.5. Перегляд даних, здатних активувати правило	92
8.6.6. Адреса зразка	93
8.6.7. Умовний елемент test	94
8.6.8. Умовний елемент or	94
8.6.9. Умовний елемент and	94
8.6.10. Умовний елемент not	95
8.6.11. Умовний елемент exists	96
8.6.12. Умовний елемент forall	97
8.6.13. Умовний елемент logical	97
Додаток 1. Приклади програм в системі CLIPS	100
Д1.1. Приклад простої бази даних	100
Д1.2. База знань з використанням властивості salience	101
Д1.3. Приклад бази знань з використанням порівняння за зразком	102
Д1.4. Приклад бази знань з використанням чисельних виразів	103
Д1.5. Приклад бази даних з введення повідомлень	104
Д1.5. Приклад бази знань з простими фактами	106
Д1.6. Приклад бази знань з використанням відкату	108
Д1.7. Приклад бази знань з можливістю модифікації	109
Література	112

ВСТУП

Мета навчальної дисципліни.

Засвоєння студентами основних теоретичних відомостей та практичних умінь з курсу. Підготувати студента до ефективного використання сучасних експертних систем у подальшій професійній діяльності; допомогти набуту навички практичної роботи з програмними засобами для вирішення задач підтримки прийняття діагностичних та керувальних рішень; навчити студентів вирішенню прикладних задач автоматизації керування, аналізу та прогнозування стану складних об'єктів і процесів на основі експертних систем.

Завдання навчальної дисципліни:

- формувати знання та практичні навички для використання експертних систем для розв'язання задач прийняття рішень;
- забезпечувати єдину методичну базу для взаємодії курсу експертних систем та інших предметних дисциплін;
- давати уяву про стан і перспективу розвитку експертних систем та їхнього програмного забезпечення.

Вимоги до знань та вмінь студентів, які вивчили дисципліну повинні

знати:

- основні поняття та визначення теорії експертних систем;
- класифікацію експертних систем;
- математичні моделі подання знань в експертних системах;
- основні принципи побудови експертних систем;
- методи логічного виведення та пошуку рішень в експертних системах;

вміти:

- обґрунтовувати та аналізувати вибір конкретного типу моделі та методу подання знань для вирішення відповідних практичних задач;
- використовувати сучасні програмні засоби для вирішення задач побудови експертних систем;
- створювати бази знань для побудови та використання експертних систем;
- здійснювати підготовку даних для побудови моделей знань;
- представляти результати роботи із експертними системами;
- аналізувати результати побудови та використання моделей й вирішення практичних задач на основі експертних систем.

У нашому курсу висвітлюються питання теорії й практики розробки експертних систем.

Як інструментальне середовище розробки використається експертна оболонка CLIPS. Суть технології CLIPS полягає в тім, що мова й середовище CLIPS надають користувачам можливість швидко створювати ефективні, компактні й легко керовані експертні системи. При цьому користувач застосовує множину вже готових інструментів (убудований механізм керування базою знань, механізм логічного висновку, менеджери різних об'єктів CLIPS і т.д.) і конструкцій (упорядковані факти, шаблони, правила, функції, родові функції, класи, модулі, обмеження, убудована мова COOL і т.д.).

Тема 1. Визначення експертної системи

День сьогодні з ранку не задався. Ви тільки що встановили нову версію текстового редактора, до якого давно звикли, але після клацання на його ярликові комп'ютер реагує зовсім не так, як хотілося б, - виводиться повідомлення начебто цього:

Call to Undefined Link (Виклик невизначеного зв'язку).

Як і більшість повідомлень про помилки, це допомагає не більше, ніж пророкування долі по стану Марса. Ви застосовуєте надзвичайний захід - видаляєте цілий каталог і переінстальюєте програму, але результат від цього не міняється. Ви починаєте міняти налаштування в різних файлах ініціалізації, але це теж не допомагає.

Нарешті, втомившись від безнадійних спроб, ви набираєте номер сервісної служби підтримки користувачів. І тільки після цього фортуна повертається до вас особою - на допомогу приходить людина, яка знає, про що говорить. Він радить вам викинути з півдюжини застарілих dll-модулів у системному каталозі й знову перевстановити програму. Виконавши його пораду, ви вже через десяток хвилин можете нормально працювати, кров'яний тиск, що недавно підскочив, знову повертається до норми.

Який би рівень експертного аналізу не був потрібний у даній області, ясно, що фахівець із сервісної служби здатний його зробити, а ви - ні. Хоча в ящику стола у вас лежить диплом кандидата технічних наук за фахом "Інформатика", і ви, можливо, прекрасно програмуєте завдання у своїй області, але, не маючи певного досвіду й підготовки, проблему усунення несправності розв'язати не змогли. Таким чином, здатність виконати експертний аналіз - це не тільки питання наявності певних знань і рівня кваліфікації. Для цього потрібно мати й дуже специфічні навички й уміння розібратися в конкретній ситуації в даній предметній області. Таким чином, бути експертом і мати загальна освіта - це далеко не те саме.

Едварда Фейгенбаума називають "батьком експертних систем", як це значиться на обкладинці однієї з його книг "Становлення експертної компанії". Він дійсно стояв у джерел експертної індустрії й створив першу експертну систему DENDRAL в області ідентифікації органічних сполук за допомогою аналізу мас-спектрограм. Далі Фейгенбаум разом із Шортліфом і Бучананом спроектували першу медичну експертну систему MYCIN, при цьому вони зробили відкриття, якому було призначено істотно розширити сферу створення й використання експертних систем. Коли

вони видалили із системи MYCIN базу знань (конкретну медичну інформацію), те залишилася частина, називана "машиною логічного висновку". Було показано, що базу знань можна змінювати й замінити повністю, не порушуючи цілісності системи. Так виникла EMYCIN (Empty MYCIN - порожній MYCIN) або перша експертна оболонка - інструментальне середовище для побудови експертних систем різного призначення. З тих часів (із середини 70-х років XX сторіччя) з'явилося велике число подібних інструментальних систем MicroExpert, GURU, G2, JESS та ін.

1.1. Що таке експертний аналіз

Замисліться над таким питанням: "При виконанні яких умов комп'ютерну програму можна назвати експертом?"

Цілком логічно зажадати, щоб така програма мала знання. Просто здатність виконувати деякий алгоритм, наприклад робити аналіз списку елементів на наявність якої-небудь властивості, явно не відповідає цій вимозі. Це однаковим, що дати першому випадковому перехожому список питань і відповідей і очікувати від нього успішного виконання пошуку й усунення несправностей у системах певного типу. Раніше або пізніше, але він обов'язково зіштовхнеться із ситуацією, не передбаченою в тому списку, яким його забезпечили.

Знання, якими має програма, повинні бути сконцентровані на певну предметну область. Випадковий набір імен, дат і місць подій, сентенцій із класиків і т.п. - це аж ніяк не ті знання, які можуть послужити основою для програми, що претендує на здатність виконати експертний аналіз. Знання припускають певну організацію й інтеграцію - тобто окремі відомості повинні співвідноситися один з одним і утворювати щось начебто ланцюжка, у якому одна ланка "тягне" за собою наступне.

І, нарешті, із цих знань повинне безпосередньо випливати вирішення проблем. Просто продемонструвати свої знання, що стосуються, наприклад, технічного обслуговування комп'ютерів, - це далеко не те ж саме, що привести комп'ютер в "почуття". Точно так само, отримати доступ до оперативної документації - це зовсім не те ж саме, що роздобути у своє розпорядження фахівця (або програму), здатного впоратися з виниклими проблемами.

Тепер спробуємо підсумувати ці міркування в наступному формальному визначенні експертної системи.

Експертна система - це програма для комп'ютера, яка оперує зі знаннями в певній предметній області з метою вироблення рекомендацій або вирішення проблем.

Експертна система може повністю взяти на себе функції, виконання яких звичайно вимагає залучення досліду людини- фахівця, або відігравати роль асистента для людини, що ухвалює розв'язок. Інакше кажучи, система (технічна або соціальна), що вимагає ухвалення рішення, може одержати його безпосередньо від програми або через проміжну ланку - людини, яка спілкується із програмою. Той, хто ухвалює рішення, може бути експертом зі своїми власними правами, і в цьому випадку програма може "виправдати" своє існування, підвищуючи ефективність його роботи. Альтернативний варіант - людей, що працює в співробітництві з такою програмою, може добитися з її допомогою результатів більш високої якості. Загалом кажучи, правильний розподіл функцій між людиною й машиною є однією із ключових умов високої ефективності впровадження експертних систем. Технологія експертних систем є одним з напрямків нової області дослідження, яка одержала найменування штучного інтелекту (Artificial Intelligence - AI). Дослідження в цій області сконцентровані на розробці й впровадженні комп'ютерних програм, здатних емулювати (імітувати, відтворювати) ті області діяльності людини, які вимагають мислення, певного майстерності й накопиченого досвіду. До них належать завдання прийняття рішень, розпізнавання образів і розуміння людської мови. Ця технологія вже успішно застосовується в деяких областях техніки й життя суспільства - органічної хімії, пошуку корисних копалин, медичній діагностиці. Перелік типових завдань, які вирішуються експертними системами, включає:

- добування інформації з первинних даних (таких як сигнали, що надходять від гідролокатора);
- діагностика несправностей (як у технічних системах, так і в людському організмі);
- структурний аналіз складних об'єктів (наприклад, хімічних сполук);
- вибір конфігурації складних багатокомпонентних систем (наприклад, розподілених комп'ютерних систем);
- планування послідовності виконання операцій, що приводять до заданої цілі (наприклад, виконувані промисловими роботами).

Хоча відомі й "звичайні" програми, що спеціалізуються на певних завданнях із представленого переліку (або аналогічних їм у суміжних областях), у наступній главі ми покажемо, у чому полягає суттєва різниця між "звичайним" підходом і запропонованим у сфері штучного інтелекту й чому експертні системи можна виділити в окремий, досить добре помітний клас програм. Чіткого формального визначення експертної системи, яке всіх би задовольнило, не існує - приведені вище теж досить розпливчасте. Але проте існує досить багато важливих ознак, властивих тією чи іншою мірою всім експертним системам.

1.2. Основні характеристики експертних систем

Експертна система відрізняється від інших прикладних програм наявністю наступних ознак.

Моделює не стільки фізичну (або іншу) природу певної проблемної області, скільки **механізм мислення** людини стосовно до розв'язку завдань у цій проблемній області. Це суттєво відрізняє експертні системи від систем математичного моделювання або комп'ютерної анімації. Не можна, звичайно, сказати, що програма повністю відтворює психологічну модель фахівця в цій предметній області (експерта), але важливо, що основна увага все-таки приділяється відтворенню комп'ютерними засобами методики рішення проблем, яка застосовується експертом, - тобто виконанню деякої частини завдань так само (або навіть краще), як це робить експерт.

Система, крім виконання обчислювальних операцій, формує певні міркування й висновки, ґрунтуючись на тих знаннях, якими вона розташовує. Знання в системі представлені, як правило, на деякій спеціальній мові й зберігаються окремо від власне програмного коду, який і формує висновки й міркування. Цей компонент програми прийнято називати **базою знань**.

При розв'язку завдань основними є евристичні й наближені методи, які, на відміну від алгоритмічних, не завжди гарантують успіх. Евристика, по суті, є правилом впливу (rule of thumb), який у машинному виді уявляє деяке знання, придбане людиною в міру накопичення практичного досвіду вирішення аналогічних проблем. Такі методи є приблизними в тому розумінні, що, по-перше, вони не вимагають вичерпної вихідної інформації, і, по-друге, існує певний ступінь упевненості (або непевності) у тому, що запропонований розв'язок є вірним.

Експертні системи відрізняються й від інших видів програм з галузі штучного інтелекту.

Експертні системи мають справу із предметами

реального світу, операції з якими звичайно вимагають наявності значного досвіду, накопиченого людиною. Множина програм з галузі штучного інтелекту є суцільно дослідницькими й основна увага в них приділяється абстрактним математичним проблемам або спрощеним варіантам реальних проблем (іноді їх називають "іграшковими" проблемами), а метою виконання такої програми є "підвищення рівня інтуїції" або відпрацювання методики. Експертні системи мають яскраво виражену практичну спрямованість у науковій або комерційній галузі.

Однією з основних характеристик експертної системи є її продуктивність, тобто швидкість одержання результату і його вірогідність (надійність). Дослідницькі програми штучного інтелекту можуть і не бути дуже швидкими, можна примиритися й з існуванням у них відмов в окремих ситуаціях, оскільки, зрештою, - це інструмент дослідження, а не програмний продукт. А ось експертна система повинна за прийнятний час знайти розв'язок, який був би не гірше, чим те, яке може запропонувати фахівець у цій предметній області.

Експертна система повинна мати здатність пояснити, чому запропонований саме такий розв'язок, і довести його обґрунтованість. Користувач повинен отримати всю інформацію, необхідну йому для того, щоб бути впевненим, що розв'язок прийнятий "не зі стелі". На відміну від цього, дослідницькі програми "спілкуються" тільки зі своїм творцем, який і так (швидше за все) знає, на чому ґрунтується її результат. Експертна система проектує розраховуючи на взаємодію з різними користувачами, для яких її робота повинна бути, по можливості, прозорою.

Найчастіше термін система, заснована на знаннях (knowledge-based system), використовується в якості синоніма терміна експертна система, хоча, строго кажучи, експертна система - це більш широке поняття. Система, заснована на знаннях, - це будь-яка система, процес роботи якої заснований на застосуванні правил відносин до символічної подання знань, а не на використанні алгоритмічних або статистичних методів. Таким чином, програма, здатна міркувати про погоду, буде системою, заснованою на знаннях, навіть у тому випадку, якщо вона не здатна виконати метеорологічну експертизу. А ось щоб мати право називатися метеорологічною експертною системою, програма повинна бути здатна давати прогноз погоди (інше питання - наскільки він буде достовірний).

Підсумовуючи все сказане, відзначимо - експертна система містить знання в певній предметній області, накопичені в результаті практичної діяльності людини (або людства), і використовує їх для вирішення проблем,

специфічних для цієї області. Цим експертні системи відрізняються від інших, "традиційних" систем, у яких перевага віддається більш загальним і менш пов'язаним із предметною областю теоретичним методам, найчастіше математичним. Процес створення експертної системи часто називають інженерією знань (knowledge engineering) і він розглядається в якості "застосування методів штучного інтелекту" (див. [Feigenbaum, 1977]). Далі, у розділах 2 і 3, ми більш пильно розглянемо відмінність між загальноприйнятим у програмуванні підходом до вирішення проблем і тем, який пропонується при проектуванні експертних систем.

У частині, що залишився, цієї глави ми розглянемо наступні питання. Спочатку будуть перераховано чотири базові функції, які реалізуються в експертних системах. Ці функції тут будуть розглянуті дуже поверхово, з єдиною метою дати читачеві загальне уявлення про них і познайомити з відповідною термінологією. Останній розділ глави дасть читачеві виставу про загальну структуру матеріалу в цій книзі й про його розподіл по главах.

1.3. Основні функції експертних систем

Оскільки теорія експертних систем виросла з більш загальної концепції штучного інтелекту, то немає нічого дивного в тому, що проблематика цих областей має багато спільного. На деяких з таких зв'язків акцентується увага в подальших розділах при огляді літератури. Ви також зустрінете в них зазначення на подальші глави цієї книги, в яких та або інша тема розглядається детально.

1.3.1. Придбання знань

Бучанан у такий спосіб сформулював функцію придбання знань [Buchanan et al, 1983]:

"[Придбання знань це] передача потенційного досвіду розв'язку проблеми від деякого джерела знань і перетворення його у вигляд, який дозволяє використовувати ці знання в програмі".

Передача знань виконується в процесі досить тривалих і великих співбесід між фахівцем із проектування експертної системи (будемо надалі називати його інженером по знаннях) і експертом у певній предметній області, здатним досить чітко сформулювати наявний у нього досвід. За існуючими оцінками, таким методом можна сформувати від двох до п'яти "елементів знання" (наприклад, правил впливу) у день. Звичайно, це дуже низька швидкість, а тому багато дослідників розглядають функцію придбання знань у якості одного з головних "вузьких місць" технології експертних систем [Feigenbaum, 1977].

Причин такої низької продуктивності предосить. Нижче перелічені тільки деякі з них.

Фахівці у вузькій області, як правило, користуються власним жаргоном, який важко перевести на звичайний "людський" мову (див. урізання 1.1). Але зміст жаргонного "слівця" аж ніяк не очевидний, а тому потрібно досить багато додаткових питань для уточнення його логічного або математичного значення. Наприклад, фахівці з військової стратегії говорять про "агресивну демонстрацію" іноземної військової моці, але при цьому не можуть пояснити, чому така "агресивна" демонстрація відрізняється від демонстрації, що не несе погрози.

Факти й принципи, що лежать в основі багатьох специфічних галузей знання експерта, не можуть бути чітко сформульовані в термінах математичної теорії або детермінованої моделі, властивості якої добре зрозумілі. Так, експертів у фінансовій галузі може бути відомо, що певні події можуть стати причиною зростання або зниження котирувань на фондовій біржі, але він нічого вам не скаже точно про механізми, які приводять до такого ефекту, або про кількісну оцінку впливу цих факторів. Статистичні моделі можуть допомогти зробити загальний довгочасний прогноз, але, як правило, такі методи не працюють відносно курсів конкретних акцій на коротких часових інтервалах.

Для того щоб розв'язати проблему в певній галузі, експерту недостатньо просто мати суму знань про факти й принципи в цій області. Наприклад, досвідчений фахівець знає, якого роду інформацією потрібно розташовувати для формулювання того або іншого судження, наскільки надійні різні джерела інформації і як можна розчленувати складну проблему на більш прості, які можна вирішувати більш-менш незалежно. Виявити в процесі співбесіди такого роду знання, засновані на особистому досвіді, що й погано піддаються формалізації, значно складніше, чим отримати простий перелік якихось фактів або загальних принципів.

Експертний аналіз навіть у дуже вузькій галузі, виконуваний людиною, дуже часто потрібно помістити в досить великий контекст, який включає й багато речей, що видадуться експерту саме собою, що розуміють, але для стороннього аж ніяк такими, що не являються. Візьмемо для прикладу експерта-юриста, який бере участь у судовому процесі. Дуже важко окреслити кількість і природу знань загального роду, які виявляються залучені в розслідування того або іншої справи.

1.3.2. Подання знань

Подання знань - ще одна функція експертної системи. Теорія подання знань - це окрема область досліджень, тісно пов'язана з філософією

формалізму й когнітивною психологією. Предмет дослідження в цій області - методи асоціативного зберігання інформації, подібні тем, які існують у мозку людини. При цьому основна увага, природно, приділяється логічній, а не біологічній боку процесу, вилучаючи подробиці фізичних перетворень.

Приклад 1.1. Синтаксис і семантика представлення знань про сімейні відносини

Основна частина подання знань, на яку часто навіть не звертають особливої уваги, полягає в тому, що подання повинна якимось способом "стандартизувати" семантичну розмаїття людської мови. Ось кілька речень.

"Степан - батько Бориса".

"Степан - Бориса батько".

"Бориса батько - Степан".

"Батьком Бориса є Степан".

Усі ці фрази виражають ту саму думку (семантично ідентичні). При машинній поданні цієї думки (знання) ми намагаємося знайти більш простий метод зіставлення форми й змісту, чим у звичайній людській мові, тобто добитися того, щоб вирази з однаковим (або схожим) змістом були однаковими й за формою. Наприклад, усі приведені вище фрази можуть бути зведені до виразу в такій формі:

батько (Степан, Борис)

батько (Степан, Василь)

В 70-х роках дослідження в області подання знань розвивалися в напрямках розкриття принципів роботи пам'яті людини, створення теорій добування відомостей з пам'яті, розпізнавання й відновлення. Деякі з досягнутих у теорії результатів привели до створення комп'ютерних програм, які моделювали різні способи зв'язування понять (*концептів*). З'явилися комп'ютерні застосування, які могли деяким чином відшукувати потрібні "елементи" знання на певному етапі розв'язку деякої проблеми. Згодом психологічна вірогідність цих теорій відійшла на другий план, а основне місце, принаймні з погляду проблематики штучного інтелекту, зайняла їхня здатність служити інструментом для роботи з новими інформаційними й керувальними структурами.

Загалом, питання подання знання було й швидше за все залишиться питанням суперечливим. Філософи й психологи найчастіше бувають шоковані безцеремонністю фахівців зі штучного інтелекту, які жваво базикають про людське знання на жаргоні, що уявляє дику суміш термінології, узятій з логіки, логістики, філософії, психології й інформатики. З іншого боку, комп'ютерний формалізм виявився новаторським

засобом постановки, а іноді й пошуку відповідей на важкі запитання, над якими сторіччями билися метафізики.

В області експертних систем подання знань цікавить нас в основному як засіб відшукування методів формального опису великих масивів корисної інформації з метою їх наступної обробки за допомогою символічних обчислень. Формальний опис означає впорядкування в рамках якої-небудь мови, що володіє достатньо чітко формалізованим синтаксисом побудови виразів і такого ж рівня семантикою, що погоджує значення виразу з його формою.

Символічні обчислення позначають виконання нечислових операцій, у яких можуть бути сконструйовані символи й символічні структури для подання різних концептів і відносин між ними.

В області штучного інтелекту ведеться інтенсивна робота зі створення мов подання (representation languages). Під цим терміном розуміються комп'ютерні мови, орієнтовані на організацію описів об'єктів і ідей, на протидію статичним послідовностям інструкцій або зберігання простих елементів даних. Основними критеріями доступу до подання знань є логічна адекватність, евристична потужність і природність, органічність нотації. Ці терміни, швидше за все, потребують пояснень.

Логічна адекватність означає, що подання повинна мати здатність розпізнавати всі відмінності, які ви закладаєте у вихідну сутність. Наприклад, неможливо уявити ідею, що кожен ліки має який-небудь побічний небажаний ефект, якщо тільки не можна буде провести відмінність між призначенням конкретного лікарського препарату і його побічним ефектом (наприклад, аспірин збільшує виразкову хворобу). У більш загальному вигляді вираз, що передає цей ефект, звучить так: "кожен ліки має небажаний побічний ефект, специфічний для цього препарату".

Евристична потужність означає, що поряд з наявністю виразної мови подання повинна існувати деякий засіб використання вистав, сконструйованих і інтерпретованих таким чином, щоб з їхньою допомогою можна було розв'язати проблему. Часто виявляється, що мова, що володіє більшою виразною здатністю в термінах кількості семантичних відмінностей, виявляється й більше складним у керуванні описом взаємозв'язків у процесі розв'язку проблеми. Здатність до виразу в багатьох зі знайдених формалізмів може виявитися досить обмеженою в порівнянні з англійською мовою або навіть стандартною логікою. Часто рівень евристичної потужності розглядається по результату, тобто по тому, наскільки легко виявляється витягти потрібне знання стосовно до конкретної ситуації. Знати, які знання найбільше підходять для розв'язку конкретної проблеми, - це

одне з якостей, яке відрізняє дійсно фахівця, експерта в певній галузі, від новачка або просто начитаний людини.

Природність нотації слід розглядати як якусь чесноту системи, оскільки більшість застосувань, побудованих на базі експертних систем, потребує накопичення великого обсягу знань, а розв'язати таке завдання досить важко, якщо угоди в мові подання занадто складні. Будь-який фахівець скаже вам, що при інших рівних характеристиках краще та система, з якої простіше працювати. Вирази, якими формально описуються знання, повинні бути по можливості простими для написання, а їх зміст повинен бути зрозумілим навіть тому, хто не знає, як же комп'ютер інтерпретує ці вирази. Прикладом може слугувати декларативний програмний код, який сам по собі дає досить чітка подання про процес його виконання навіть тому, хто не має подання про деталі реалізації комп'ютером окремих інструкцій.

За минулі роки було запропоновано чимало угод, придатних для кодування знань на мовному рівні. Серед них відзначимо *правила, що породжують* (production rules) [Davis and King, 1977], *структуровані об'єкти* (structured objects) [Findler, 1979] і *логічні програми* (logic programs) [Kowalski, 1979]. У більшості експертних систем використовується один або трохи з перерахованих формалізмів, а доводи на користь і проти кожного з них дотепер являють собою тему для жвавих дискусій серед теоретиків.

Більшість фрагментів програмного коду, наведених у цій книзі, написано мовою CLIPS, у якій використовується комбінація правил, що породжують, і структурованих об'єктів. У Додатку ви знайдете досить великий опис основних концепцій і програмних конструкцій мови CLIPS, яке супроводжується безліччю прикладів.

1.3.3. Керування процесом пошуку розв'язку

При проектуванні експертної системи серйозна увага повинна бути приділена й тому, як здійснюється доступ до знань і як вони використовуються при пошуку розв'язку [Davis, 1980, а]. Знання про те, які знання потрібні в тієї або іншій конкретній ситуації, і вміння ними розпорядитися - важлива частина процесу функціонування експертної системи. Такі знання дістали найменування мета-знань - тобто знань про знання. Вирішення нетривіальних проблем вимагає й певного рівня планування й керування при виборі, яке питання потрібно задати, який тест виконати, і т. ін.

Використання різних стратегій перебору наявних знань, як правило, виявляє досить істотний вплив на характеристики ефективності

програми. Ці стратегії визначають, яким способом програма відшукує розв'язок проблеми в деякому просторі альтернатив. Як правило, не буває так, щоб дані, якими розташовує програма роботи з базою знань, дозволяли точно "вийти" на ту область у цьому просторі, де має сенс шукати відповідь.

Більшість формалізмів вистави знань може бути використане в різних режимах керування, і розробники експертних систем продовжують експериментувати в цій області. У наступних главах будуть описані системи, які спеціально підібрані таким чином, щоб проілюструвати відмінності в існуючих підходах до розв'язку проблеми керування. У кожній із представлених систем є що-небудь корисне для студентів, що спеціалізуються в галузі розробки й дослідження експертних систем.

Приклад. 1.2. Обслуговування автомобіля

Уявіть собі, що ваш автомобіль із труднощами заводиться, а в дорозі явно відчувається зниження потужності. Самі по собі ці симптоми недостатні для того, щоб ухвалити рішення, де ж шукати джерело несправності - у паливній або електросистемі автомобіля.

Пізнання в обладнанні автомобіля підказують - потрібно ще поекспериментувати, перш ніж звати на допомогу механіка. Можливо, погана паливна суміш, тому придивитися до вихлопу й нагару на свічах. Можливо, збоїть розподільник - подивитися, не ушкоджена чи його кришка. Ці досить специфічні евристики не гарантують, що віднайдеться дійсна причина, але раптом вам посміхнеться фортуна, і ви знайдете несправність без втомлюючої процедури послідовної перевірки всіх систем. Швидше за все, ваших знань досить для того, щоб виконати загальну перевірку, перш ніж займатися досконалим вивченням окремих вузлів. Наприклад, подивитися, чи досить потужна іскра у свічі (якщо це так, то підозри з електросистеми можна зняти), перш ніж перевіряти акумулятор. При відсутності спеціальних евристик, чому більш методично ви будете діяти, тим більше шансів швидко знайти причину несправності. Загальне евристичне правило говорить:

"Спочатку перевір увесь вузол, а вже потім приступай до перевірки його компонентів".

Це правило можна вважати частиною режиму керування - систематичної стратегії застосування наявних знань. Інше евристичне правило можна сформулювати, наприклад, так:

"Спочатку міняй більш дешеві деталі, а вже потім берися за більш дорогі".

У деяких випадках ці дві евристики можуть

суперечити один одному, так що потрібно заздалегідь вибрати, яка з них має пріоритет у випадку, якщо обидві включені в той самий режим керування.

1.3.4. Роз'яснення ухваленого рішення

Питання про те, як допомогти користувачу зрозуміти структуру й функції деякого складного компонента програми, зв'язаний із порівняно новою галуззю взаємодії людини й машини, яка з'явилася на перетинанні таких областей, як штучний інтелект, промислова технологія, фізіологія й ергономіка. На сьогодні внесок у цю галузь дослідників, що займаються експертними системами, полягає в розробці методів подання інформації про поведінку програми в процесі формування ланцюжка логічних висновків при пошуку розв'язку.

Подання інформації про поведінку експертної системи важливо з багатьох причин.

Користувачі, що працюють із системою, потребують підтвердження того, що в кожному конкретному випадку висновок, до якого прийшла програма, в основному коректно.

Інженери, що мають справу з формуванням бази знань, повинні переконатися, що сформульовані ними знання застосовані правильно, у тому числі й у випадку, коли існує прототип.

Експертам у предметній області бажане простежити хід міркувань і спосіб використання тих відомостей, які з їхніх слів були введені в базу знань. Це дозволить судити, наскільки коректно вони застосовуються в даній ситуації.

Програмістам, які супроводжують, налагоджують і модернізують систему, потрібно мати у своєму розпорядженні інструмент, що дозволяє заглянути в "її нутро" на рівні більш високому, чим виклик окремих мовних процедур.

Менеджер системи, що використовує експертну технологію, який зрештою відповідає за наслідки розв'язку, прийнятого програмою, також потребує підтвердження, що ці розв'язки досить обґрунтовані.

Здатність системи пояснити методику ухвалення рішення іноді називають *прозорістю системи*. Під цим розуміється, наскільки просто персоналу з'ясувати, що робить програма й чому. Цю характеристику системи слід розглядати в сукупності з режимом керування, про який ішла мова в попередньому розділі, оскільки послідовність етапів ухвалення рішення тісно пов'язана із заданою стратегією поведінки.

Відсутність достатньої прозорості поведінки системи не дозволить експертові вплинути на її продуктивність або дати пораду, як можна її

підвищити. Простежування й оцінка поведінки системи - завдання досить складне й для її розв'язку необхідні спільні зусилля експерта й фахівця з комп'ютерних наук.

Вправи

1. Чим експертні системи відрізняються від звичайних програмних застосунків і типових програм штучного інтелекту? чи Може програма, що не використовує методи штучного інтелекту, мати такі ж властивості?

2. У чому різниця між експертною системою й системою, заснованою на знаннях?

3. Чи є експертною системою програма прогнозування погоди в Київській області, яка виводить повідомлення такого роду: "Завтра погода не буде відрізнятися від сьогоднішньої"? Припустимо, що вона представляє сьогоднішню погоду в символьному виді, легко модифікується й здатна до розширення, прекрасно працює й може пояснити, чому вона прийшла до певного висновку, вивівши приблизно таке повідомлення: "Добові

зміни кліматичних умов у цю пору року малоймовірні".

4. Чи є експертною системою програма, яка формує прогноз погоди на певну дату (скажемо, 16 червня), узявши середні температуру повітря, кількість опадів, що випали, і кількість сонячних годин 16 червня за всі роки, починаючи з 1900?

5. Чи є система пошуку в мережі World Wide Web експертною? Якщо ні, то яких властивостей їй не вистачає для того, щоб кваліфікувати її як експертну систему пошуку потрібної Web-Сторінки?

6. Чому завдання придбання знань є вузьким місцем у проектуванні експертних систем? Які розв'язки пропонуються для усунення такої ситуації?

7. Поясніть зауваження про логічну й евристичну адекватність, яке ставиться до мови подання знань.

Тема 2. Подання знань

2.1. Знання й дані

Якщо у вас є проблема або задача, яку не можна вирішити самостійно - ви звертаєтеся до знаючих людей, або до експертів, до тих, хто має ЗНАННЯ. Термін "системи, засновані на знаннях" (knowledge-based systems) з'явився в 1976 році одночасно з першими системами, що акумулюють досвід і знання експертів. Це були експертні системи (expert systems) MYCIN й DENDRAL [Shortliffe, 1976; Shortliffe Feigenbaum, Buchanan, 1978] для медицини й хімії. Вони ставили діагноз при інфекційних захворюваннях крові й розшифровували дані мас-спектрографічного аналізу.

Експертні системи з'явилися в рамках досліджень по штучному інтелекті (ШІ) (artificial intelligence) у той період, коли ця наука переживала серйозну кризу, і був потрібний істотний прорив у розвитку практичних додатків. Цей прорив відбувся, коли на зміну пошукам універсального алгоритму мислення й рішення задач дослідникам прийшла ідея моделювати конкретні знання фахівців-експертів. Так у США з'явилися перші комерційні системи, засновані на знаннях, або експертні системи (ЕС). Ці системи по праву стали першими інтелектуальними системами, і дотепер єдиним критерієм інтелектуальності є наявність механізмів роботи зі знаннями.

Так з'явився новий підхід до рішення задач штучного інтелекту — *подання знань*.

При вивченні інтелектуальних систем традиційно виникає питання - що ж таке знання й чим вони відрізняються від звичайних даних, десятиліттями оброблюваних на комп'ютерах. Можна запропонувати кілька робочих визначень, у рамках яких це стає очевидним.

Визначення 2.1

Дані — це інформація, отримана в результаті спостережень або вимірів окремих властивостей (атрибутів), що характеризують об'єкти, процеси та явища предметної області.

Інакше, дані - це конкретні факти, такі як температура повітря, висота будинку, прізвище співробітника, адреса сайту й ін.

При обробці на ЕОМ дані трансформуються, умовно проходячи наступні етапи:

- D1 - дані як результат вимірів і спостережень;
- D2 - дані на матеріальних носіях інформації (таблиці, протоколи, довідники);

- D3 - моделі (структури) даних у вигляді діаграм, графіків, функцій;
- D4 - дані в комп'ютері мовою описи даних;
- D5 - бази даних на машинних носіях інформації.

Знання ж засновані на даних, отриманих емпіричним шляхом. Вони являють собою результат досвіду й розумової діяльності людини, спрямованої на узагальнення цього досвіду, отриманого в результаті практичної діяльності.

Так, якщо озброїти людини даними про те, що в нього висока температура (результат спостереження або виміру), те цей факт не дозволить йому вирішити задачу видужання. А якщо досвідчений лікар поділиться знаннями про те, що температуру можна знизити жарознижуючими препаратами й рясним питвом, те це істотно наблизить рішення задачі видужання, хоча насправді потрібні додаткові дані й більше глибокі знання.

Визначення 2.2

Знання — це зв'язку й закономірності предметної області (принципи, моделі, закони), отримані в результаті практичної діяльності й професійного досвіду, що дозволяє фахівцям ставити й вирішувати задачі в даній області.

При обробці на ЕОМ знання трансформуються аналогічно даним:

- Z1 - знання в пам'яті людини як результат аналізу досвіду й мислення;
- Z2 - матеріальні носії знань (спеціальна література, підручники, методичні посібники);
- Z3 - поле знань - умовний опис основних об'єктів предметної області, їхніх атрибутів і закономірностей, їх єдналих;
- Z4 — знання, описані на мовах подання знань (продукційні мови, семантичні мережі, фрейми — див. далі);
- Z5 - база знань на машинних носіях інформації.

Часто використовується й таке визначення знань:

Знання — це добре структуровані дані, або дані про дані, або метадані.

Ключовим етапом при роботі зі знаннями є формування поля знань (третій етап Z3), ця нетривіальна задача включає виявлення й визначення об'єктів і понять предметної області, їхніх властивостей і зв'язків між ними, а також подання їх

у наочній й інтуїтивно зрозумілій формі. Цей термін уперше був уведений при практичній розробці експертної системи по психодіагностиці АВТАНТЕСТ і тепер широко використовується розробниками ЕС.

Без ретельного пророблення поля знань не може бути мови про створення бази знань.

Істотним для розуміння природи знань є способи визначення понять. Один із широко застосовуваних способів заснований на ідеї інтенціонала й екстенціонала.

Визначення 2.3

Інтенціонал поняття — це визначення його через співвіднесення з поняттям більш високого рівня абстракції із вказівкою специфічних властивостей.

Наприклад, інтенціонал поняття "МЕБЛІ": "предмети, призначені для забезпечення комфортного проживання людини й захищаючі будинок".

Визначення 2.4

Екстенціонал — це визначення поняття через перерахування його конкретних прикладів, тобто понять більше низького рівня абстракції.

Екстенціонал поняття "МЕБЛІ": "Шафа, диван, стіл, стілець і т.д."

Інтенціонали формують знання про об'єкти, у те час як екстенціонал поєднує дані. Разом вони формують елементи поля знань конкретної предметної області.

Для зберігання даних використовуються бази даних (для них характерні великий об'єм і відносно невелика питома вартість інформації), для зберігання знань - бази знань (невеликого об'єму, але винятково дорогі інформаційні масиви).

База знань — основа будь-якої інтелектуальної системи, де знання описані на деякій мові подання знань, наближеній до природного.

Знання можна розділити на:

- глибинні;
- поверхневі.

Поверхневі — знання про видимі взаємозв'язки між окремими подіями й фактами в предметній області.

Глибинні — абстракції, аналогії, схеми, що відображують структуру й природу процесів, що протікають у предметній області. Ці знання пояснюють явища й можуть використатися для прогнозування поведінки об'єктів.

Поверхневі знання

"Якщо ввести правильний пароль, на екрані комп'ютера з'явиться зображення робочого стола".

Глибинні знання

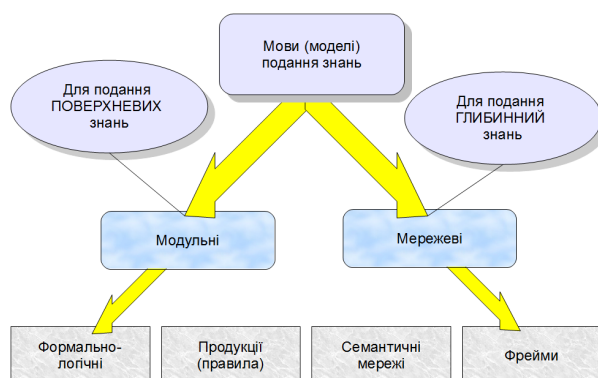
"Розуміння принципів роботи операційної системи й знання на рівні кваліфікованого системного адміністратора".

Сучасні експертні системи працюють, в основному, з поверхневими знаннями. Це пов'язане з тим, що на даний момент немає універсальних методик, що дозволяють виявляти глибинні структури знань і працювати з ними.



Рис 2.1. Піраміда Н'юела.

Крім того, у підручниках по ШІ знання традиційно ділять на процедурні й декларативні. Історично первинними були процедурні знання, тобто знання, "розчинені" в алгоритмах. Вони управляли даними. Для їхньої зміни було потрібно змінювати текст програм. Однак з розвитком інформатики й програмного забезпечення все більша частина знань зосереджувалася в структурах даних (таблиці, списки, абстрактні типи даних), тобто збільшувалася роль декларативних знань.



Сьогодні знання придбали чисто декларативну форму, тобто знаннями вважаються пропозиції, записані на мовах подання знань, наближених до природної мови й зрозумілих неспеціалістам.

Один з піонерів ШІ Алан Н'юел проілюстрував

еволюцію засобів спілкування людини з комп'ютером як перехід від машинних кодів через символічні мови програмування до мов подання знань (рис. 2.1).

2.2. Моделі подання знань

2.2.1. Продукційна модель

МПЗ, засновані на правилах (rule-based), є найпоширенішими й більше 80% ЕС використовують саме їх.

Визначення 1.5

Продукційна модель або модель, заснована на правилах, дозволяє представити знання у вигляді пропозицій типу "Якщо (умова), те (дія)".

Під "умовою" (антецедентом) розуміється деяка пропозиція-зразок, по якому здійснюється пошук у базі знань, а під "дією" (консеквентом) - дії, виконувані при успішному результаті пошуку (вони можуть бути проміжними, що виступають далі як умови, і термінальну або цільову, завершальну роботу системи).

Найчастіше висновок на такій базі знань буває прямий (від даних до пошуку мети) або зворотний (від мети для її підтвердження — до даних). Дані — це вихідні факти, що зберігаються в базі фактів, на підставі яких запускається машина висновку або інтерпретатор правил, що перебирає правила із продукційної бази знань.

Продукційна, модель так часто застосовується в промислових експертних системах, оскільки залучає розробників своєю наочністю, високої модульності, легкістю внесення доповнень і змін і простотою механізму логічного висновку.

Є велика кількість програмних засобів, що реалізують продукційний підхід (наприклад, мови високого рівня CLIPS й OPS 5; "оболонки" або "порожні" ЕС - EXSYS Professional і Карра, інструментальні системи KEE, ARTS, PIES), а також промислових ЕС на його основі (наприклад, ЕС, створених засобами G2.

2.2.2. Семантичні мережі

Термін "семантична" означає "змістовна", а сама семантика — це наука, що встановлює відносини між символами й об'єктами, які вони позначають, тобто наука, що визначає зміст знаків. Модель на основі семантичних мереж була запропонована американським психологом Куїліаном. Основною її перевагою є те, що вона більше інших відповідає сучасним поданням про організації довгострокової пам'яті людини.

Визначення 1.6

Семантична мережа — це орієнтований граф, вершини якого — поняття, а дуги — відносини між ними.

Як поняття звичайно виступають абстрактні або конкретні об'єкти, а відносини це зв'язку типу: "це" ("АКО — A-Kind-Of, "is" або "елемент класу"), "має частиною" ("has part"), "належить", "любить".

Можливо запропонувати кілька класифікацій семантичних мереж, пов'язаних з типами відносин між поняттями.

По кількості типів відносин:

- однорідні (з єдиним типом відносин);
- неоднорідні (з різними типами відносин).
- По типах відносин:
 - бінарні (у які відносини зв'язують два об'єкти);
 - N-арні (у які є спеціальні відносини, що зв'язують більше двох понять).

Найбільше часто в семантичних мережах використовуються наступні відносини:

- елемент класу (троянда *це* квітка);
- атрибутивні зв'язки /мати властивість (пам'ять *має властивість* — об'єм);
- значення властивості (кольори *має значення* — жовтий);
- приклад елемента класу (троянда, *наприклад* — чайна);
- зв'язку типу "*частина-ціле*" (велосипед *включає* кермо);
- функціональні зв'язки (обумовлені звичайно дієсловами "робить", "впливає"...);
- кількісні (*більше, менше, дорівнює*...);
- просторові (*далеко від, близько від, за, під, над*...);
- часові (*раніше, пізніше, протягом*...);
- логічні зв'язки (*і, або, не*) і ін.

Мінімальний склад відносин у семантичній мережі такий:

- елемент класу або АКО;
- атрибутивні зв'язки /мати властивість;
- значення властивості.

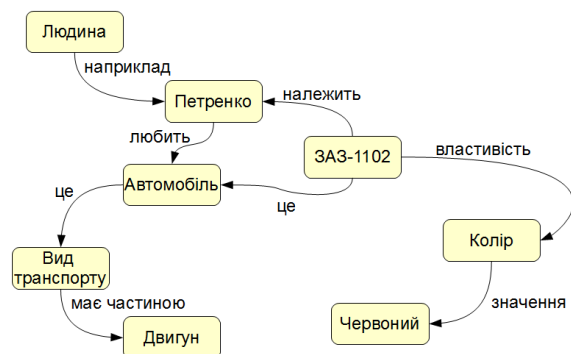
Недоліком цієї моделі є складність організації процедури організації висновку на семантичній мережі.

Ця проблема зводиться до нетривіальної задачі пошуку фрагмента мережі, що відповідає деякої підмережі, що відбиває поставлений запит до бази.

На мал. 2.3 зображений приклад семантичної

мережі. Як вершини отут виступають поняття "людина", "Петренко", "ЗА3-1102", "автомобіль", "вид транспорту" й "двигун".

Для реалізації семантичних мереж існують спеціальні мережні мови, наприклад, NET, мова реалізації систем SIMER + MIR й ін. Широко відомі експертні системи, що використовують семантичні мережі як мова подання знань - PROSPECTOR, CASNET, TORUS.



2.2.3. Фрейми

Термін *фрейм* (від англ. *frame* — "каркас" або "рамка") був запропонований Марвіном Мінським, одним з піонерів ШІ, в 70-і роки для позначення структури знань для сприйняття просторових сцен. Ця модель, як і семантична мережа, має глибоке психологічне обґрунтування.

Визначення 1.7

Фрейм — це абстрактний образ для подання стереотипу об'єкта, поняття або ситуації.

Інтуїтивно зрозуміло, що під абстрактним образом розуміється деяка узагальнена й спрощена модель або структура. Наприклад, проголошення вголос слова "кімната" породжує в почутому образ кімнати: "житлове приміщення із чотирма стінами, підлогою, стелею, вікнами й дверима, площею 6—20 м²". Із цього опису нічого не можна забрати (наприклад, забравши вікна, ми одержимо вже прикомірок, а не кімнату), але в ньому є "дірки" або "слоти" — це незаповнені значення деяких атрибутів — наприклад, кількість вікон, кольори стін, висота стелі, покриття підлоги й ін.

У теорії фреймів такий образ кімнати називається фреймом кімнати. Фреймом також називається й формалізована модель для відображення образу.

Розрізняють фрейми-зразки або прототипи, що зберігаються в базі знань, і фрейми-екземпляри, які створюються для відображення реальних фактичних ситуацій на основі даних, що надходять. Модель фрейму є досить універсальною, оскільки дозволяє відобразити все різноманіття знань про світ через:

- *фрейму-структури*, що використовуються для позначення об'єктів і понять (позиція, застава, вексель);
- *фреймів-ролі* (менеджер, касир, клієнт);
- *фрейми-сценарії* (банкрутство, збори акціонерів, святкування іменин);
- *фреймів-ситуації* (тривога, аварія, робочий режим пристрою) і ін.

Традиційно структура фрейму може бути представлена як список властивостей: (ІМ'Я ФРЕЙМУ:

(ім'я 1-го слоту: значення 1-го слоту) ,
 (ім'я 2-го слоту: значення 2-го слоту) ,

 (ім'я N-го слоту: значення N-го слоту)) .

Той же запис можна представити у вигляді таблиці (див. табл. 2.1), доповнивши її двома стовпцями.

Таблиця 2.1. Структура фрейму

Ім'я фрейму			
Ім'я слоту	Значення слоту	Спосіб одержання значення	Приєднана процедура

У таблиці додаткові стовпці (3-й й 4-й) призначені для опису способу одержання слотом його значення й можливого приєднання до того або іншого слоту спеціальних процедур, що допускається в теорії фреймів. Як значення слоту може виступати ім'я іншого фрейму, так утворюються мережі фреймів.

Існує кілька способів одержання слотом значень у фреймі-екземплярі:

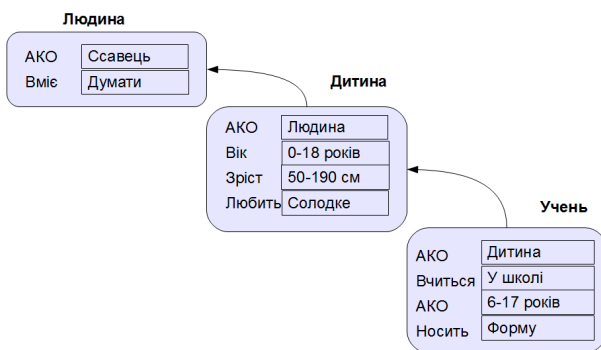
- за замовчуванням від фрейму-зразка (Default-значення);
- через спадкування властивостей від фрейму, зазначеного в слоті АКО;
- по формулі, зазначеної в слоті;
- через приєднану процедуру;
- явно з діалогу з користувачем;
- з бази даних.

Найважливішою властивістю теорії фреймів є

запозичення з теорії семантичних мереж - так називане спадкування властивостей. І у фреймах, і в семантичних мережах спадкування відбувається по АКО-св'язям (A-Kind-Of = це). Слот АКО вказує на фрейм більше високого рівня ієрархії, звідки неявно успадковуються, тобто переносяться, значення аналогічних слотів.

Наприклад, у мережі фреймів на мал. 1.4 поняття "учень" успадковує властивості фреймів "дитина" й "людина", які перебувають на більше високому рівні ієрархії. На питання "чи люблять учні солодке?" треба відповідь "так", тому що цією властивістю володіють всі діти, що зазначено у фреймі "дитина". Спадкування властивостей може бути частковим: вік для учнів не успадковується із фрейму "дитина", оскільки зазначено явно у своєму власному фреймі.

Основною перевагою фреймів як моделі подання знань є те, що вона відбиває концептуальну основу організації пам'яті людини [Шенк, Хантер, 1987], а також її гнучкість і наочність.



Спеціальні мови подання знань у мережах фреймів FRL (Frame Representation Language) [Байдун, Бунін, 1990], KRL (Knowledge Representation Language) [Уотермен, 1989], фреймова "оболонка" Карра [Стрельников, Борисов, 1997] й інші програмні засоби дозволяють ефективно будувати промислові ЕС. Широко відомі такі фрейм-орієнтовані експертні системи, як ANALYST, МОДИС, TRISTAN, ALTERID [Ковригин, Перфильев, 1988; Николов, 1988; Sisodia, Warkentin, 1992].

2.2.4. Формальні логічні моделі

Традиційно в поданні знань виділяють *формальні логічні моделі*, засновані на класичному вирахованні предикатів 1-го порядку, коли предметна область або задача описується у вигляді набору аксіом. Реальне вираховування предикатів 1-го порядку в промислових експертних системах практично не використовується. Ця логічна модель застосовна в основному в дослідницьких "іграшкових" системах, тому що пред'являє дуже високі вимоги й обмеження до предметної області. У промислових же експертних системах використовуються різні її модифікації й розширення, виклад яких виходить за рамки цього

циклу лекцій.

2.3. Висновок на знаннях

Найбільше поширення одержала продукційна модель подання знань. При її використанні база знань складається з набору правил, а програма, що управляє перебором правил, називається машиною висновку.

Визначення 1.8

Машина висновку (інтерпретатор правил) — це програма, що імітує логічний висновок експерта, що користується даною продукційною базою знань для інтерпретації даних, що надійшли в систему.

Обычно вона виконує дві функції:

- перегляд існуючих даних (фактів) з робочої пам'яті (бази даних) і правил з бази знань і додавання (у міру можливості) у робочу пам'ять нових фактів;
- визначення порядку перегляду й застосування правил. Цей механізм управляє процесом консультації, зберігаючи для користувача інформацію про отримані висновки, і запитує в нього інформацію, коли для спрацьовування чергового правила в робочій пам'яті виявляється недостатньо даних [Осуга, Сазки, 1990].

У переважній більшості систем, заснованих на знаннях, механізм висновку являє собою невелику по об'єму програму й включає два компоненти - один реалізує безпосередньо висновок, інший - управляє цим процесом.

Дія *компонента висновку* засновано на застосуванні правила, називаного *modus ponens*: "Якщо відомо, що щиро твердження А, і існує правило виду "ЯКЩО А, ТО В", тоді твердження В також істинно".

Таким чином, правила спрацьовують, коли перебувають факти, що задовольняють їхній лівій частині: якщо щиро посилку, то повинне бути істинно й висновок.

Компонент висновку повинен функціонувати навіть при недоволі інформації. Отримане рішення може й не бути точним, однак система не повинна зупинятися через те, що відсутня яка-небудь частина вхідної інформації.

Керуючий компонент визначає порядок застосування правил і виконує чотири функції:

2. *Зіставлення* — зразок правила зіставляється з наявними фактами.

3. *Вибір* — якщо в конкретній ситуації можуть бути застосовані відразу кілька правил, то з них вибирається одне, найбільш підходяще за заданим критерієм (дозвіл конфлікту).

4. *Спрацьовування* — якщо зразок правила при

зіставленні збігся з будь-якими фактами з робочої пам'яті, те правило спрацює.

5. *Дія* — робоча пам'ять піддається зміні шляхом додавання в неї висновку правила, що спрацювало. Якщо в правій частині правила втримується вказівка на яку-небудь дію, то воно виконується (як, наприклад, у системах забезпечення безпеки інформації).

Інтерпретатор продукцій працює циклічно. У кожному циклі він переглядає всі правила, щоб виявити тих, посилки яких збігаються з відомими на даний момент фактами з робочої пам'яті. Після вибору правило спрацює, його висновок заноситься в робочу пам'ять, і потім цикл повторюється спочатку.

В одному циклі може спрацювати тільки одне правило. Якщо кілька правил успішно зіставлені з фактами, то інтерпретатор робить вибір за певним критерієм єдиного правила, що спрацює в даному циклі. Цикл роботи інтерпретатора схематично представлений на мал. 2.5.

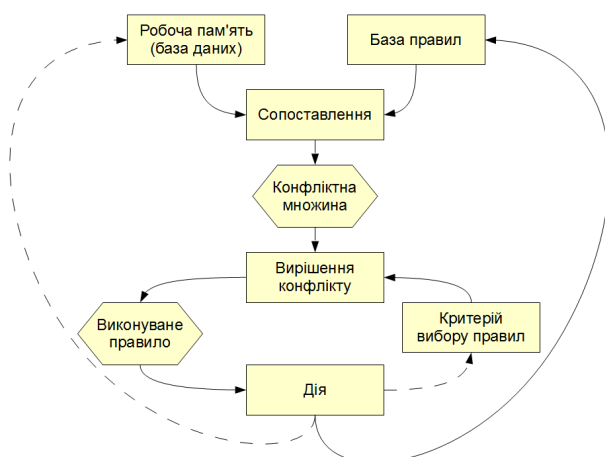


Рис. 2.5. Цикл роботи інтерпретатора

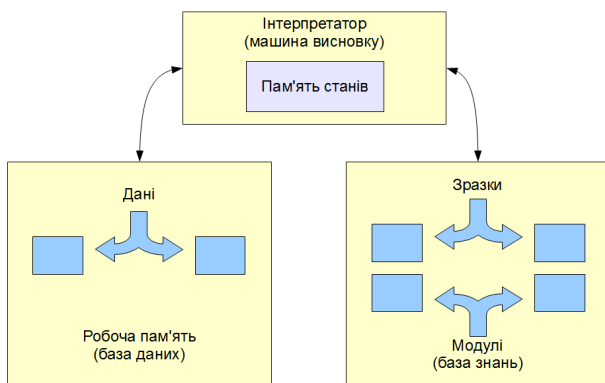


Рис. 2.6. Схема функціонування інтерпретатора

Інформація з робочої пам'яті послідовно зіставляється з посилками правил для виявлення успішного зіставлення. Сукупність відібраних правил становить так названу *конфліктну множину*. Для дозволу конфлікту інтерпретатор має критерій, за допомогою якого він вибирає єдине правило, після

чого воно спрацює. Це виражається в занесенні фактів, що утворюють висновок правила, у робочу пам'ять або в зміні критерію вибору конфліктуючих правил. Якщо ж на закінчення правила входить назва якої-небудь дії, то воно виконується.

Робота машини висновку залежить тільки від стану робочої пам'яті й від складу бази знань. На практиці звичайно враховується історія роботи, тобто поводження механізму висновку в попередніх циклах. Інформація про поводження механізму висновку запам'ятовується в пам'яті станів (мал. 2.6). Звичайно пам'ять станів містить протокол системи.

У системах із *прямим висновком* по відомих фактах відшукується висновок, що із цих фактів треба (див. мал. 1.7, ліва частина). Якщо такий висновок вдається знайти, то воно заноситься в робочу пам'ять. Прямий висновок часто називають висновком, керованим даними, або висновком, керованим антецедентами.

Існують системи, у яких висновок ґрунтується на сполученні згаданих вище методів - зворотного й обмеженого прямого. Такий комбінований метод одержав назву циклічного.

Нехай є фрагмент бази знань із двох правил:

- П1: Якщо "відпочинок - улітку" й "людина - активна", то "їхати в гори".
- П2: Якщо "любить сонце", те "відпочинок улітку".

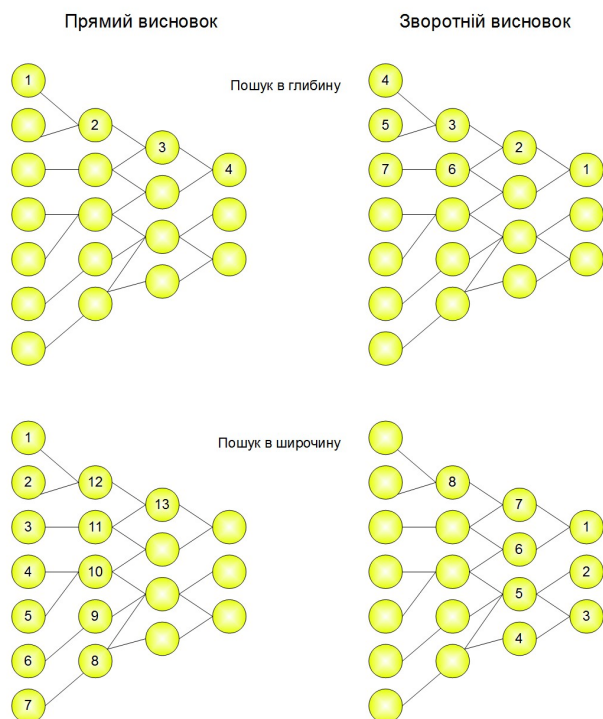


Рис. 1.7. Стратегії висновку

Припустимо, у систему надійшли факти - "людина активний" й "любить сонце".

ПРЯМИЙ ВИСНОВОК— виходячи з фактичних даних, одержати рекомендацію.

- 1-й прохід.

- *Крок 1.* Пробуємо /7/, не працює (не вистачає даних "відпочинок - улітку").
- *Крок 2.* Пробуємо /72, працює, у базу надходить факт "відпочинок - улітку".

- 2-й прохід.

- *Крок 3.* Пробуємо Я/, працює, активізується мета "їхати в гори", що і виступає як рада, що дає ЕС.

ЗВОРОТНИЙ ВИСНОВОК— підтвердити обрану мету за допомогою наявних правил і даних.

- 1-й прохід.

- *Крок 1.* Ціль — "їхати в гори": пробуємо *П1* — даних "відпочинок — улітку" ні, вони стають новою метою й шукається правило, де вона в лівій частини.
- *Крок 2.* Ціль "відпочинок — улітку": правило *П2* підтверджує мета й акти - візує її.

- 2-й прохід.

- *Крок 3.* Пробуємо *П1*, підтверджується шукана мета.

2.3.2. Методи пошуку в глибину й завширшки

У системах, база знань яких нараховує сотні правил, бажаним є використання стратегії керування висновком, що дозволяє мінімізувати час пошуку рішення й тим самим підвищити ефективність висновку. До числа таких стратегій ставляться: пошук у глибину, пошук завширшки, розбивку на підзадачі й альфа-бета-алгоритм.

При пошуку в глибину в якості чергової підцілі вибирається та, котра відповідає наступний, більше детальному рівню опису задачі. Наприклад, що діагностує система, зробивши на основі відомих симптомів припущення про наявність певного захворювання, буде продовжувати запитувати уточнювальні ознаки й симптоми цієї хвороби доти, поки повністю не спростує висунуту гіпотезу.

При пошуку завширшки, навпроти, система спочатку проаналізує всі симптоми, що перебувають на одному рівні простору станів, навіть якщо вони ставляться до різних захворювань, і лише потім перейде до симптомів наступного рівня детальності.

Розбивка на підзадачі має на увазі виділення підзадач, рішення яких розглядається як досягнення проміжних цілей на шляху до кінцевої мети. Прикладом, що підтверджує ефективність розбивки на підзадачі, є пошук несправностей у комп'ютері -

спочатку виявляється підсистема, що відмовила (живлення, пам'ять і т.д.), що значно звужує простір пошуку. Якщо вдається правильно зрозуміти сутність задачі й оптимально розбити її на систему ієрархічно зв'язаних цілей-підцілей, то можна домогтися того, що шлях до її рішення в просторі пошуку буде мінімальний.

Альфа-бета-алгоритм дозволяє зменшити простір станів шляхом видалення галузей, не перспективних для успішного пошуку. Тому проглядаються тільки ті вершини, у які можна потрапити в результаті наступного кроку, після чого безперспективні напрямки виключаються. Альфа-бета-алгоритм знайшов широке застосування в основному в системах, орієнтованих на різні ігри, наприклад, у шахових програмах.

2.4. Об'єкто-орієнтований підхід, об'єднаний із правилами

Найбільш розвиненим способом подання знань в ЕС є об'єкто-орієнтований підхід, що є розвитком фреймового подання. У його основі лежать поняття об'єкт і клас. У предметної області, що цікавить розробника, як об'єкти можуть розглядатися конкретні предмети, а також абстрактні або реальні сутності. Об'єкт має індивідуальність і поведінням, має атрибути, значення яких визначають його стан. Так, наприклад, конкретний покупець, роблячи замовлення, може виявитися в стані, коли грошей на його рахунок не вистачає для оплати, а його поведіння в цьому випадку полягає у зверненні до банку за кредитом.

Кожен об'єкт є представником деякого класу однотипних об'єктів. Клас визначає загальні властивості для всіх його об'єктів. До таких властивостей ставляться склад і структура даних, що описують атрибути класу й відповідних об'єктів, і сукупність методів - процедур, що визначають взаємодію об'єктів цього класу із зовнішнім середовищем.

Об'єкти й класи мають характерні властивості, які активно використовуються при об'єкто-орієнтованому підході й багато в чому визначають його переваги. До цих властивостей ставляться:

Інкапсуляція – приховання інформації. При об'єкто-орієнтованому програмуванні є можливість заборонити доступ до атрибутів об'єктів, доступ можливий тільки через його методи. Внутрішня структура об'єкта в цьому випадку схована від користувача, об'єкти можна вважати самостійними сутностями, відділеними від зовнішнього миру. Для того щоб об'єкт зробив деяку дію, йому ззовні необхідно послати повідомлення, що ініціює виконання потрібного методу. Інкапсуляція дозволяє змінювати реалізацію будь-якого класу об'єктів без побоювання, що це викличе небажані побічні ефекти в програмній системі. Тим самим спрощується

процес виправлення помилок і модифікації програм.

Спадкування – можливість створювати із класів нові класи за принципом «від загального до частки». Спадкування дозволяє новим класам, при збереженні всіх властивостей клас-батьків, додавати свої риси, що відбивають їхню індивідуальність. З погляду програміста новий клас повинен містити тільки коди й дані для нових або методів, що змінюються. Повідомлення, обробка яких не забезпечується власними методами класу, передається класу-батькові. Спадкування дозволяє створювати ієрархії класів й є ефективним засобом внесення змін і доповнень у програмні системи.

Поліморфізм – здатність об'єктів вибирати метод на основі типів даних, прийнятих у повідомленні. Кожен об'єкт може реагувати по-своєму на те саме повідомлення. Поліморфізм дозволяє спростити вихідні тексти програм, забезпечує їхній розвиток за рахунок введення нових методів обробки.

Отже, об'єкто-орієнтований підхід полягає в поданні системи у вигляді сукупності класів й об'єктів предметного середовища. При цьому ієрархічний характер складної системи відбивається у вигляді ієрархії класів, а її функціонування розглядається як взаємодія об'єктів, з якими асоціюється, наприклад, продукційні правила. Асоціювання продукційних правил ЕС з ієрархією класів здійснюється за рахунок використання загальних правил, як префікс яких звичайно використовується посилання на клас, до якого дане правило застосовне. У процедурній інтерпретації наявність префікса, що зв'язує продукцію із класом, викликає необхідність перебору всіх екземплярів зазначеного в префіксі класу і його підкласів і перевірки істинності умови для атрибутів кожного з екземплярів.

Застосування об'єкто-орієнтованого підходу в системах інженерії знань виводить на перший план можливість природної декомпозиції задачі на сукупність підзадач, що представляють досить автономними агентами, що працюють зі знаннями. На сьогоднішній день це єдина практична можливість роботи в умовах експонентного росту складності (кількість взаємозв'язків), характерного для систем, що використовують знання. Так практично всі інструментальні засоби для створення динамічних ЕС підтримують об'єкто-орієнтований підхід, об'єднаний із правилами.

2.5. Робота з нечіткістю

При формалізації знань існує проблема, що утрудняє використання традиційного математичного апарата. Це проблема опису понять, що оперують якісними характеристиками об'єктів (багато, мало,

сильний, дуже сильний і т.п.). Ці характеристики звичайно розмиті й не можуть бути однозначно інтерпретовані, однак містять важливу інформацію (наприклад, "одним з можливих ознак грипу є висока температура").

Крім того, у задачах, розв'язуваних інтелектуальними системами, часто доводиться користуватися неточними знаннями, які не можуть бути інтерпретовані як повністю щирі або помилкові (логічні true/false або 0/1). Існують знання, вірогідність яких виражається деякою проміжною цифрою, наприклад 0,7.

Як, не руйнуючи властивості розмитості й неточності, представляти подібні знання формально? Для дозволу таких проблем на початку 70-х років ХХ століття американський математик Лотфи Заде запропонував формальний апарат нечіткої (fuzzy) алгебри й нечіткої логіки [Заде, 1972]. Пізніше цей напрямок одержав широке поширення [Орловський, 1981; Аверкин й ін., 1986; Яшин, 1990] і поклато початок однієї з галузей ІІІ за назвою м'які обчислення (soft computing).

Л. Заде ввів одне з головних понять у нечіткій логіці – поняття лінгвістичної змінної.

Лінгвістична змінна (ЛПЗ) — це змінна, значення якої визначається набором вербальних (тобто словесних) характеристик деякої властивості.

Наприклад, ЛПЗ "ріст" визначається через набір {карликовий, низький, середній, високий, дуже високий}.

2.5.1. Основи теорії нечітких множин

Значення лінгвістичної змінної (ЛПЗ) визначаються через так називані нечіткі множини (НМ), які у свою чергу визначені на деякому базовому наборі значень або базовій числовій шкалі, що має розмірність. Кожне значення ЛПЗ визначається як нечітка множина (наприклад, НМ "низький ріст").

Нечітка множина визначається через деяку базову шкалу В і функцію приналежності НМ — $\mu(x)$,

В Д приймаючого значення на інтервалі [0; 1]. Таким чином, нечітка множина В — це сукупність пар виду $(x, \mu(x))$. Часто зустрічається й такий запис:

$$B = \sum_{i=1}^n \frac{x_i}{\mu(x_i)}.$$

де x_i — i -те значення базової шкали.

Функція приналежності визначає суб'єктивний ступінь упевненості експерта в тім, що дане конкретне значення базової шкали відповідає обумовленому НМ. Цю функцію не варто плутати з імовірністю, що носить об'єктивний характер і підкоряється іншим математичним залежностям.

Наприклад, для двох експертів визначення НМ "висока" для ЛП "ціна автомобіля" в умовних одиницях може істотно відрізнитися залежно від їх соціального й фінансового становища.

$$\text{"Висока_ціна_автомобіля_1"} = \{50000/1 + 25000/0.8 + 10000/0.6 + 5000/0.4\}$$

$$\text{"Висока_ціна_автомобіля_2"} = \{25000/1 + 10000/0.8 + 5000/0.7 + 3000/0.4\}$$

Нехай перед нами коштує задача інтерпретації значень ЛП "вік", таких як "молодий" вік, "похилий" вік або "перехідний" вік. Визначимо "вік" як ЛП (мал. 1.8). Тоді "молодий", "похилий", "перехідний" будуть значеннями цієї лінгвістичної змінної. Більш повно, базовий набір значень ЛП "вік" наступний:

$V = \{\text{дитячий, дитячий, юний, молодий, зрілий, похилий, старечий}\}$.

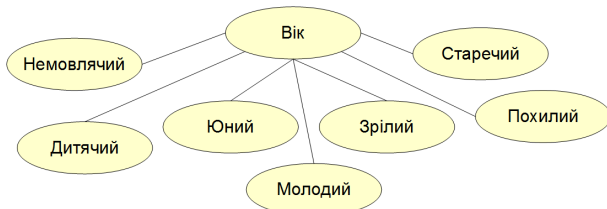


Рис. 1.8. Лінгвістична змінна "вік" і нечіткі множини, що визначають її значення

Для ЛП "вік" базова шкала - це числова шкала від 0 до 120, що позначає кількість прожитого років, а функція приналежності визначає, наскільки ми впевнені в тому, що дана кількість років можна віднести до даної категорії віку. На мал. 1.9 відбито, як ті самі значення базової шкали можуть брати участь у визначенні різних НМ.

Наприклад, визначити значення НМ "дитячий" можна так:

$$\text{немовлячий} = \left\{ \frac{0,5}{1} + \frac{1}{0,9} + \frac{2}{0,8} + \frac{3}{0,7} + \frac{4}{0,6} + \frac{5}{0,3} + \frac{10}{0,1} \right\}$$

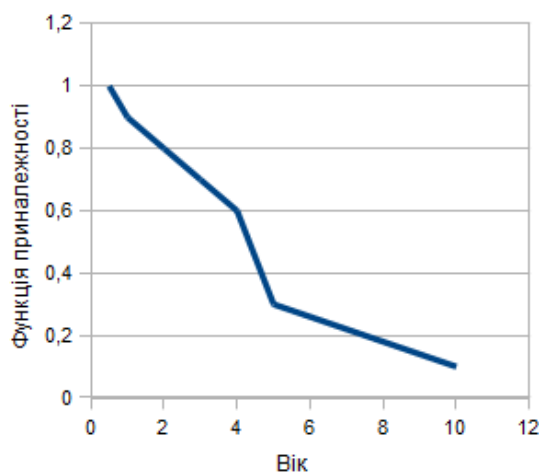
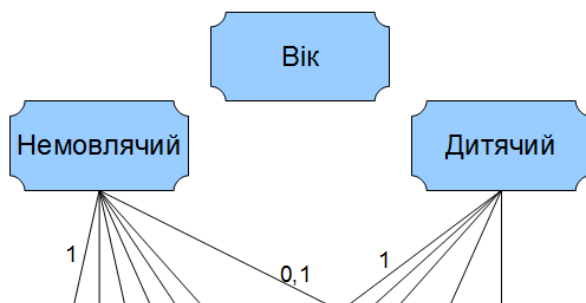


Рис.2.9. Формування нечітких множин

Рис.2.10. Графік функції приналежності нечіткій множині "немовлячий вік".

Рис. 2.10 ілюструє оцінку НМ якимсь усередненим експертом, що дитини до напівроку з високим ступенем упевненості відносять до дитин ($\mu = 1$). Діти до чотирьох років зараховуються до дитин теж, але з меншим ступенем упевненості ($0,5 < \mu < 0,9$), а в десять років дитини називають так тільки в дуже рідких випадках - приміром, для дев'яностолітньої бабусі й 15 років може вважатися дитинством. Таким чином, нечіткі множини дозволяють при визначенні поняття враховувати суб'єктивні думки окремих індивідуумів.

2.5.2. Операції з нечіткими знаннями

Для операцій з нечіткими знаннями, вираженими за допомогою лінгвістичних змінних, існує багато різних способів. Ці способи є в основному евристичними.

Ми не будемо зупинятися на цьому питанні докладно, укажемо лише для прикладу визначення декількох операцій. Наприклад, операція "АБО" часто задається так :

(так називана логіка Заде)

або так:

(імовірнісний підхід).

Посилення йди ослаблення лінгвістичних понять досягається введенням спеціальних квантифікаторів. Наприклад, якщо поняття "старечий вік" визначається як

$$\left\{ \frac{60}{0,6} + \frac{70}{0,8} + \frac{80}{0,9} + \frac{90}{1} \right\}$$

то поняття "дуже старечий вік" розпізнається як

$$\text{con}(A) = A^2 = \sum_{x \in X} \frac{x^2}{x}$$

т. е. дуже старечий вік визначиться так:

$$\left\{ \frac{60}{0,36} + \frac{70}{0,64} + \frac{80}{0,81} + \frac{90}{1} \right\}$$

Для висновку на нечітких множинах використовуються спеціальні відносини й операції над ними.

Одним з перших застосувань теорії НМ

стало використання коефіцієнтів упевненості для висновку рекомендацій медичної системи MYCIN. Цей метод використає кілька евристичних прийомів. Він став прикладом обробки нечітких знань, що вплинули на наступні системи.

У цей час у більшість інструментальних засобів розробки систем, заснованих на знаннях, включені елементи роботи із НМ,

крім того, розроблені спеціальні програмні засоби реалізації так називаного нечіткого висновку, наприклад "оболонка" FuzzyCLIPS.

Вправи

Тема 3. Інженерія знань

3.1. Визначення й структура інженерії знань

Основні труднощі в розробці експертних систем пов'язані із проблемою добування й структурування знань. Саме ці питання досліджує наука за назвою — *інженерія знань* (knowledge engineering).

Визначення 3.1

Інженерія знань — напрямок досліджень і розробок в області інтелектуальних систем, що ставить метою розробку моделей, методів і систем для одержання, структурування й формалізації знань фахівців з метою проектування баз знань.

Основні напрямки досліджень інженерії знань представлені на мал. 3.1.



Рис. 3.1. Структура інженерії знань

3.1.1. Поле знань

Дана лекція цілком присвячена теоретичним проблемам інженерії знань, іншими словами — проектуванню баз знань. Центральним поняттям на стадіях одержання й структурування є так називане *поле знань*, що вже згадувалося в розд. 2.1.1.

Визначення 3.2

Поле знань — це умовний неформальний опис основних понять і взаємозв'язків між поняттями предметної області, виявлених із системи знань експерта, у вигляді графа, діаграми, таблиці або тексту.

Поле знань формується на третій стадії розробки ЕС — стадії структурування.

Поле знань, як перший крок від структурування до формалізації, представляє модель знань про предметну область у тім виді, у якому її зумів

виразити аналітик на деякому "своєму" мові.

Узагальнено синтаксичну структуру поля знань можна представити як

де

- I — структура вхідних даних, підлягаючих обробці й інтерпретації в експертній системі;
- $ПРО$ — структура вихідних даних, тобто результату роботи системи;
- M — операціональна модель предметної області, на підставі якої відбувається модифікація I в O .

Включення компонентів I й O в P_z обумовлене тим, що складові й структура цих інтерфейсних компонентів неявно присутні в моделі репрезентації в пам'яті експерта. Операціональна модель M може бути представлена як сукупність концептуальної структури, що відбиває понятійну структуру предметної області, і функціональної структури, що моделює схему міркувань експерта.

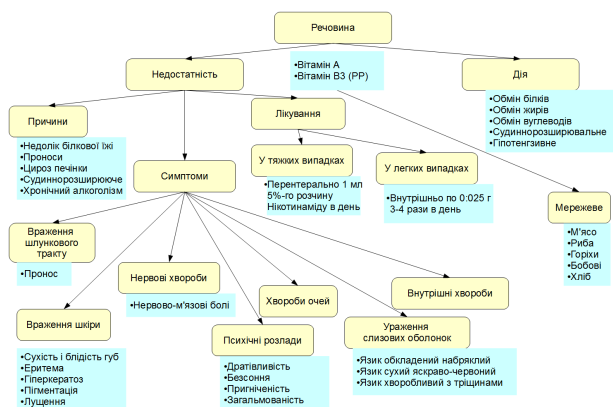
виступає як статична, незмінна складова у той час як представляє динамічну, змінювану складову.

Формування засноване на виявленні понятійної структури предметної області. Далі описаний досить універсальний алгоритм проведення концептуального аналізу на основі модифікації парадигми структурного аналізу і побудови ієрархії понять (так називана "піраміда знань"). Приклад й представлений на мал. 3.2 й 3.3.

В останні роки концептуальну структуру називають *онтологією предметної області*, вона включає впорядковані поняття предметної області (ПО) A і моделює основні функціональні зв'язки RA або відносини між поняттями, що утворюють. Крім онтології розуміння задачі відбиває модель або стратегія ухвалення рішення в обраній ПО. Таким чином, утворить стратегічну складову M , часто вона має форму простої таблиці рішень, як на рис. 3.4.

Схему, що відображає відносини між реальною

дійсністю й полем знань, можна представити так, як показано на рис. 3.4.



2.2. Концептуальна складова поля знань

Ознака 1	Ознака 2	Ознака 3	...Діагноз	Лікування
Ураження шкіри	Ураження очей	Нервові розлади	Брак речовини	Продукти
Сухість губ або лущення	Підупадання зору	Безсоння	Вітамін А	Вершкове масло або морква
		Дратливість	Вітамін В3	М'ясо або риба

Рис.3.3. Функціональна складова поля знань

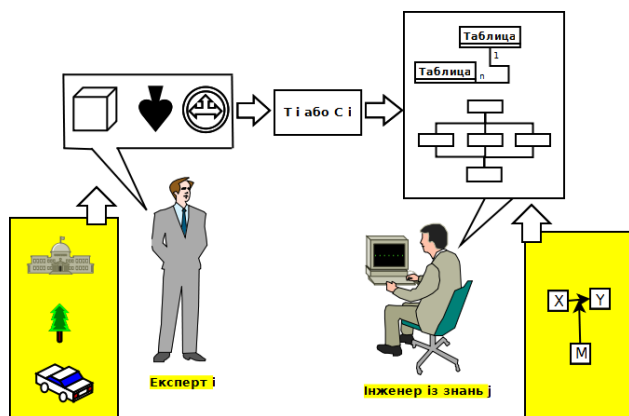


Рис. 3.4. "Зіпсований телефон" при формуванні поля знань

Поле Pz_g^{ij} — це результат, отриманий "після 4-ї трансляції" (якщо говорити мовою інформатики).

1-я трансляція (I_i) — це сприйняття й інтерпретація дійсності *Про* предметну область g i -м експертом. У результаті I_i у пам'яті експерта утвориться модель Mgi як семантична репрезентація дійсності і його особистого досвіду по роботі з нею.

2-я трансляція (Vi_j) — це вербалізація досвіду i -го експерта, коли він намагається пояснити свої міркування Si і передати свої знання Zi , інженерові по знаннях. У результаті Vi , утвориться або текст Ti , або мовне повідомлення Ci

3-я трансляція (Ij) — це сприйняття й інтерпретація повідомлень Ti або Ci j -м інженером по знаннях. У результаті в пам'яті інженера по знаннях утвориться модель миру Mgi .

4-я трансляція (Kj) — це кодування й вербалізація моделі Mgi у формі поля знань Pz_g^{ij} .

Найбільше ця схема нагадує дитячу гру в "зіпсований телефон"; перед інженером по знаннях стоїть найскладніша задача — домогтися максимальної відповідності Mgi й Pz_g^{ij} . На жаль, Pz_g не є відбиттям дійсності O_g , тому що знання — річ суцільно авторизована, суб'єктивна. Так варто було б на кожній ЕС ставити чіткий ярлик $i-j$, тобто "база знань експерта i у розумінні інженера по знаннях j ". Варто замінити, наприклад, інженера по знаннях Петрова на Сидорова, і вийде зовсім інша картина.

Приведемо приклад впливу суб'єктивних поглядів експерта на Mgi й Vi . Реальність (O_g): два чоловіки прибігають на вокзал за 2 хвилини до відходу поїзда. У каси - черга. В автоматичних касах вільно, але ні в того, ні в іншого немає дріб'язку. Наступний поїзд через 40 хвилин. Обоє спізняються на важливу зустріч.

- **Інтерпретація 1-го експерта ($I1$):** не можна приходити на вокзал менш чим за 10 хвилин.
- **Інтерпретація 2-го експерта ($I2$):** треба завжди мати дріб'язок у кишені.
- **Вербалізація 1-го експерта ($V1$):** спізнився до потрібного поїзда, тому що не розрахував час.
- **Вербалізація 2-го експерта ($V2$):** спізнився, тому що на вокзалі плутанина, у касах юрба.

Наступної трансляції ще більше будуть спотворювати й видозмінювати модель, але тепер уже з урахуванням суб'єктивного сприйняття інженерів по знаннях.

Таким чином, якщо вважати поле знань змістовний (семантичний) моделлю предметної області, то ця модель двічі суб'єктивна. І якщо модель Mgi (див. мал. 2.4) — це усичене відображення O_g , то саме Pz — лише відблиск Mgi через призму Vi й Mgj .

3.1.2. "Піраміда" знань

Ієрархічність понятійної структури свідомості підкреслюється в роботах багатьох психологів. Поле знань можна стратифікувати, тобто

розглядати на різних рівнях абстракції понять. В "піраміді знань" кожен наступний рівень служить для сходження на новий щабель узагальнення й поглиблення знань у предметній області. Таким чином, можлива наявність декількох рівнів понятійної структури S_k

Представляється доцільним зв'язати це із глибиною професійного досвіду (наприклад, як у системі АВТАНТЕСТ) або з рівнем ієрархії в структурних сходах організації (рис. 3.5).

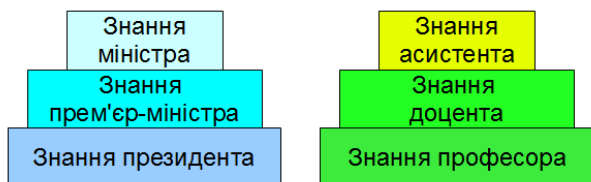


Рис. 3.5. Піраміди знань

Природно, що й стратегії прийняття рішень, тобто функціональні структури S_f на різних рівнях, будуть істотно відрізнятися.

3.2. Стратегії одержання знань

При формуванні поля знань ключовим питанням є сам процес одержання знань, коли відбувається перенос компетентності експертів на інженерів по знаннях. Для назви цього процесу в літературі по ЕС одержали поширення кілька термінів: придбання, видобуток, добування, одержання, виявлення, формування знань. В англійській спеціальній літературі в основному використовуються два терміни: *acquisition* (придбання) і *elicitation* (виявлення, добування, установлення).

Термін "придбання" трактується або дуже широко — тоді він включає весь процес передачі знань від експерта до бази знань ЕС, або вже як спосіб автоматизованої побудови бази знань за допомогою діалогу експерта й спеціальної програми (при цьому структура поля знань заздалегідь заставляється в програму). В обох випадках термін "придбання" не стосується самого таїнства екстрагування структури знань із потоку інформації про предметну область. Цей процес описується поняттям "добування".

Визначення 3.3

Добування знань (knowledge elicitation) — це процес взаємодії аналітика із джерелом знань, у результаті якого стає явним процес міркувань фахівця при ухваленні рішення й структура його подань про предметну область.

За всіх часів більшість розробників ЕС відзначало, що процес добування знань залишається самим "вузьким" місцем при побудові промислових ЕС. При цьому їм доводиться практично самостійно розробляти методи добування,

зіштовхуючись із наступними труднощами :

1. Організаційні непогодженості;
2. Невдалий метод добування, що не збігає зі структурою знань у даній області;
3. Неадекватна модель (мова) для подання знань.

Можна додати до цього :

- невміння налагодити контакт із експертом;
- термінологічний різнобіж;
- відсутність цілісної системи знань у результаті добування тільки "фрагментів";
- спрощення "картини миру" експерта й ін.

Процес добування знань - це тривала й трудомістка процедура, у якій інженерові по знаннях, збройному спеціальними знаннями по когнітивній психології, системному аналізу, математичній логіці й ін., необхідно відтворити модель предметної області, якою користуються експерти для ухвалення рішення. Часто починаючи розробники ЕС, бажаючи спростити цю процедуру, намагаються підмінити інженера по знаннях самим експертом. З багатьох причин це небажано.

По-перше, більша частина знань експерта - це результат численних нашарувань, щаблів досвіду. І часто знаючи, що з А треба В, експерт не віддає собі звіту, що ланцюжок його міркувань був набагато довше, наприклад $A \rightarrow D \rightarrow Z \rightarrow V$, або $A \rightarrow Q \rightarrow R \rightarrow V$.

По-друге, як було відомо ще Платону, мислення діалогічно. І тому діалог інженера по знаннях й експерта - найбільш природна форма вивчення лабіринтів пам'яті експерта, у яких зберігаються знання, частиною носіями невірбальний характер, тобто виражені не у формі слів, а у формі наочних образів, наприклад. І саме в процесі пояснення інженерові по знаннях експерт на ці розмиті асоціативні образи надягає чіткі словесні ярлики, тобто вербалізує знання.

По-третє, експертові сутужніше створити модель предметної області внаслідок глибини й обсягу інформації, який він володіє. Ще в ситуаційному керуванні було виявлено, що об'єкти реального миру зв'язані більш ніж 200 типами відносин (тимчасового, просторового, причинно-наслідкового, типу "частина-ціле" й ін.). Ці відносини й зв'язки предметної області утворюють складну систему, з якої виділити "кістяк" або головну структуру іноді доступніші аналітикові, що володіє до того ж системною методологією.

Термін "придбання" у межах нашого купує залишений за автоматизованими системами прямого спілкування з експертом. Вони дійсно безпосередньо здобувають уже готові фрагменти знань у відповідності зі структурами, закладеними розробниками систем.

Більшість цих інструментальних засобів спеціально орієнтовано на конкретні ЕС з жорстко позначеною предметною областю й моделлю подання знань, тобто не є універсальними.

Визначення 3.4

Придбання знань (knowledge acquisition) — процес заповнення бази знань експертом з використанням спеціалізованих програмних засобів.

Наприклад, система TEIRESIAS, що стала прародителькою всіх інструментів для придбання знань, призначена для поповнення бази знань системи MYCIN або її дочірніх галузей, побудованих на "оболонці" EMYCIN в області медичної діагностики з використанням продукційної моделі подання знань.

Термін формування знань (machine learning) традиційно закріпився за надзвичайно перспективною й активно, що розвивається областю, інженерії знань, що займається розробкою моделей, методів й алгоритмів навчання. Вона включає індуктивні моделі формування знань й автоматичного породження гіпотез, наприклад метод на основі навчальних вибірок, навчання за аналогією й іншими методами. Ці моделі дозволяють виявити причинно-наслідкові емпіричні залежності в базах даних з неповною інформацією, що містять структуровані числові й символічні об'єкти (часто в умовах неповноти інформації).

Визначення 3.5

Формування знань (machine learning) — процес аналізу даних і виявлення схованих закономірностей з використанням спеціального математичного апарата й програмних засобів.

Традиційно до задач формування знань або машинного навчання ставляться задачі прогнозування, ідентифікації (синтезу) функцій, розшифровки мов, індуктивного висновку й синтезу з додатковою інформацією. У широкому сенсі до навчання за прикладами можна віднести й методи навчання розпізнаванню образів.

Для того щоб ці методи стали елементами технології інтелектуальних систем, необхідно вирішити ряд задач :

- забезпечити механізм сполучення незалежно створених баз даних, що мають різні схеми, з базами знань інтелектуальних систем;
- установити відповідність між набором полів бази даних і множиною елементів декларативного компонента бази знань;
- виконати перетворення результату роботи алгоритму навчання в спосіб подання, підтримуваний програмними засобами інтелектуальної системи.

Крім перерахованих існують також й інші стратегії одержання знань, наприклад, у випадку навчання на прикладах (case-based reasoning), коли джерело знань - це множина прикладів предметної області. Навчання на основі прикладів (прецедентів) включає настроювання алгоритму розпізнавання на задачу за допомогою пред'явлення прикладів, класифікація яких відома.

Навчання на прикладах тісно пов'язане з машинним навчанням. Розходження полягає в тім, що результат навчання в розглянутому тут випадку повинен бути інтерпретований у деякій моделі, у якій, можливо, уже втримуються факти й закономірності предметної області, і перетворений у спосіб подання, що допускає використання результату навчання в базі знань, для моделювання міркувань, для роботи механізму пояснення й т.д., тобто робить результат навчання елементом відповідної технології.

Наприклад, у системі INDUCE породжується несуперечливий опис деякого класу об'єктів по множинах прикладів і контрприкладів даного класу. Як мова подання використовується мова змінно-значної логіки першого порядку (варіант мови багатозначної логіки першого порядку).

Останнім часом широке поширення одержали терміни data mining й knowledge discovery, що означають, по суті, той же процес формування знань і пошук закономірностей, здійснюваний на більших вибірках даних, що звичайно перебувають у *сховищах даних* (data warehouse).

Таким чином, можна виділити три основних стратегії проведення стадії одержання знань при розробці ЕС (рис. 3.6):

- с використанням ЕОМ при наявності підходящого програмного інструментарію — *придбання знань*;
- с використанням програм навчання при наявності репрезентативної (тобто досить представницької) вибірки прикладів прийняття рішень у предметній області й відповідних пакетів прикладних програм — *формування знань*;
- без використання обчислювальної техніки шляхом безпосереднього контакту інженера по знаннях і джерела знань (будь те експерт, спеціальна література або інші джерела) — *добування знань*.

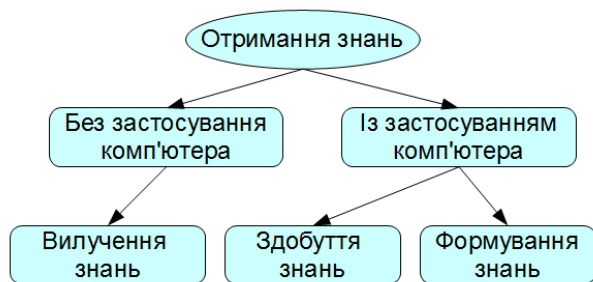


Рис. 3.6. Стратегії одержання знань

3.3. Теоретико-методичні аспекти добування й структурування знань

Розглянемо докладніше ключову проблему основної стратегії одержання знань - безпосереднього добування знань "з" пам'яті експерта. Можна виділити три основних аспекти цього процесу (мал. 2.7).

- психологічний;
- лінгвістичний;
- гносеологічний.

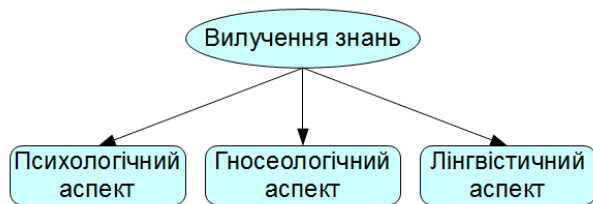


Рис. 3.7. Теоретико-методичні аспекти інженерії знань

3.3.1. Психологічний аспект

Провідним аспектом добування знань є психологічний, оскільки він визначає успішність й ефективність взаємодії інженера по знаннях (аналітика) з основним джерелом знань - експертом-професіоналом. Психологічний аспект виділяється ще й тому, що добування знань відбувається найчастіше в процесі безпосереднього спілкування розробників системи й експертів, і психологічні особливості можуть повністю звести нанівець високий інтелектуальний потенціал розробки.

На жаль, всі члени колективу розробників найчастіше мають технічне утворення й не володіють навіть зачатками професійних прийомів спілкування. Спілкування, або комунікація (від лат. *communicatio* — зв'язок) — це міждисциплінарне поняття, що позначає всі форми безпосередніх контактів між людьми — від дружніх до ділових.

Існує кілька десятків теорій спілкування, і єдине, у чому сходяться всі автори, - це складність

і багатоплановість процедури спілкування. Підкреслюється, що спілкування - не просто односпрямований процес передачі повідомлень й але двотактний обмін порціями відомостей, а нерозчленований процес циркуляції інформації, тобто спільний пошук істини.

Отже, спілкування є процес вироблення нової інформації, загальної для людей, що спілкуються, і породжуючою їх Спільноту, І хоча спілкування перший вид діяльності, яким опановує Людина у своєму розвитку, 110-сьогоденню володіють Культурою й наукою спілкування одиниці.

Виділяють чотири основних рівні спілкування :

- *рівень маніпулювання*, казана один суб'єкт розглядає інший як засіб або перешкоду стосовно проекту своєї діяльності;
- *рівень «рефлексивної гри»*, коли в процесі своєї діяльності людина враховує «контрпроект» іншого суб'єкта, але не визнає за ним самоцінність і прагне до «виграшу», до реалізації свого проекту;
- *рівень правового спілкування*, коли суб'єкти визнають право на існування проектів діяльності один одного й намагаються погодити їх хоча б зовні;
- *рівень морального спілкування*, коли суб'єкти внутрішньо приймають загальний проект взаємної діяльності.

Прагнення й уміння спілкуватися на вищому, четвертому, рівні може характеризувати ступінь професіоналізму інженера по знаннях.

Відомо, що втрати інформації при розмовному спілкуванні великі.

У зв'язку із цим розглянемо проблему збільшення інформативності спілкування аналітика й експерта за рахунок використання психологічних знань.

Можна виділити такі структурні компоненти моделі спілкування при добуванні знань:

- учасники спілкування (партнери);
- засоби спілкування (процедура);
- предмет спілкування (знання).

Відповідно до цієї структури виділимо три «шари» психологічних проблем виникаючих при добуванні знань:

- контактний;
- процедурний;
- когнітивний.

3.3.2. Контактний шар

Розробка проблематики контактного шару дозволила виявити наступні параметри партнерів, що впливають на результати процедури добування знань:

- підлога й вік;
- особистість;
- темперамент;
- мотивація й ін.

3.3.2.1. Стать і вік

Значення параметрів статі й віку хоча й впливають на ефективність контакту, але не є критичними. У літературі відзначається, що гарні результати дають гетерогенні пари (чоловік/жінка), тому що підвищується мотивація.

Що стосується віку, то робоча формула має вигляд

$$20 > (B_э - B_a) > 5,$$

де $B_э$ — вік експерта; B_a — вік аналітика.

3.3.2.2. Особистість

Під особистістю звичайно розуміється стійка система психологічних рис, що характеризує індивідуальність людини. Компоненти особистісного портрета, що рекомендують, аналітика досліджені в роботі і доповнені якостями з керівництва для журналістів: доброзичливість, аналітичність, гарна пам'ять, увага, спостережливість, уява, вразливість, більша зібраність, наполегливість, товарицькість, спритність..

3.3.2.3. Темперамент

Із часів Гіппократа, що виділили чотири класичних типи темпераменту, увійшли в наукову термінологію поняття — *холерик, сангвінік, меланхолік, флегматик*.

Відомо, що флегматики й меланхоліки повільніше засвоюють інформацію. І для забезпечення психологічного контакту з ними не слід задавати бесіді занадто швидкий темп, квапити їх з відповіддю. Зате вони набагато краще засвоюють нове, на відміну від холериків, для яких властиво поверхнєве засвоєння інформації. Останніх варто спеціально наводити на міркування й рефлексію. У меланхоліків часто занижена самооцінка, вони соромливі й у бесіді їх треба підбадьорювати. Таким чином, найбільш успішними в рамках контактного шару є сангвініки й холерики.

3.3.2.4. Мотивація

На ефективність колективного рішення задач впливає також і мотивація, тобто прагнення до успіху. Інженер по знаннях залежно від умов розробки повинен вишукувати різноманітні стимули для експертів (включаючи, зрозуміло, і

матеріальні). Експерт передає аналітику один з найдорожчих у світі продуктів - знання. І якщо одні люди діляться досвідом добровільно й із задоволенням, то інші досить неохоче відкривають свої професійні таємниці. Іноді корисно, виявляється, збудити в експерті дух суперництва, конкуренції (не порушуючи, природно, обстановки кооперативності в колективі).

3.3.3. Процедурний шар

Параметри процедурного шару описують безпосередньо процес проведення процедури добування знань. Фактично це професійні параметри:

- ситуація спілкування (місце, час і тривалість);
- устаткування (допоміжні засоби, освітленість, меблі);
- професійні прийоми (темп, стиль, облік невербальних компонентів спілкування, методи й ін.).

Інженер по знаннях, що успішно опанував наукою встановлення атмосфери довіри й взаєморозуміння з експертом (контактний шар), повинен ще зуміти скористатися сприятливим впливом цієї атмосфери. Проблема процедурного шару стосується проведення самої процедури добування знань. Тут мало проникливості й чарівності, корисних для рішення проблеми контакту, отут необхідні професійні знання.

Зупинимося на загальних закономірностях проведення процедури.

3.3.3.1. Ситуація спілкування

Вона визначається наступними компонентами:

- місце проведення сеансів;
- тривалість і час проведення сеансів.

Місце. Бесіду з експертом найкраще проводити в невеликому приміщенні наодинці, оскільки сторонні люди порушують довірчість бесіди й можуть породити ефект "фасаду". Робоче місце експерта є не самим оптимальним варіантом, тому що його можуть відволікати телефонні дзвінки, співробітники й ін. Атмосфера замкнутого простору й самотності позитивно впливають на ефективність.

Американський психолог И. Атватер вважає, що для ділового спілкування найбільш сприятлива дистанція від 1,2 до 3 м. Мінімальним "комфортним" відстанню можна вважати 0,7-0,8 м.

Тривалість і час. Реконструкція власних міркувань - трудомісткий процес, і тому тривалість одного сеансу звичайно не перевищує 1,5-2 години.

Ці друга година краще вибрати в першій половині дня, наприклад з 10 до 12 годин, якщо експерт типу "жайворонок", і в районі 16-17 годин, якщо він "сова". Ці два піки активності (11 й 16 годин) установлені дослідниками 24-вартового ритмоциклу людської активності.

Відомо, що взаємна стомлюваність партнерів при бесіді настає звичайно через 20-25 хвилин, тому в сеансі потрібні паузи.

3.3.3.2. Устаткування

Звичайно включає:

- допоміжні засоби;
- освітленість і меблі.

Допоміжні засоби являють собою засоби для збільшення ефективності самого процесу добування знань і засобу для протоколювання результатів.

До засобів для збільшення ефективності процесу добування знань, насамперед, ставиться наочний матеріал. Незалежно від методу добування, обраного в конкретній ситуації, його реалізація можлива різними способами.

Наприклад, широко відомо, що людей, що займаються інтелектуальною діяльністю, можна віднести до художнього або розумового типу. Терміни отут умовні й не мають відносини до тієї діяльності, що традиційно називають художньою або розумовою. Важливо, що, визначивши тип експерта, інженер по знаннях може плідотворніше використати кожної з методів добування, знаючи, що люди художнього типу легше сприймають зорову інформацію у формі малюнків, графіків, діаграм, тому що ця інформація сприймається через першу сигнальну систему. Навпроти, експерти розумового типу краще розуміють мову формул і текстову інформацію. При цьому враховується факт, що більшу частину інформації людина одержує від зору. Рада користуватися активніше наочним матеріалом з можна вважати універсальним. Такі методи, як вільний діалог й ігри, надають багаті можливості використати слайди, креслення, малюнки.

Для протоколювання результатів у цей час використовуються наступні способи:

- запис на папір безпосередньо по ходу бесіди (недоліки - це часто заважає бесіді, крім того, важко встигнути записати всі, навіть при наявності навичок стенографії);
- магнітофонний запис (диктофон), що допомагає аналітикові проаналізувати весь хід сеансу й свої помилки (недолік - може скоувати експерта);
- запам'ятовування з наступним записом після

бесіди (недолік - годиться тільки для аналітиків із блискучою пам'яттю).

Найпоширенішим способом на сьогодні є перший. При цьому найбільша небезпека отут - втрата знань, оскільки будь-який запис відповідей - це вже інтерпретація, тобто привнесення суб'єктивного розуміння предмета.

Природним є вимога відключати мобільний телефон під час сеансу (хоча б з боку аналітика).

Освітленість і меблі. Значення параметрів *освітленості й меблів* також впливають на ефективність процедури добування знань. Посадите експерта на незручну табуретку, направте йому світло в очі й подивитися, скільки хвилин він витримає...

3.3.3.3. Професійні прийоми аналітика

Вони включають, зокрема:

- темп і стиль;
- облік невербальних компонентів спілкування;
- методи.

Облік і стиль. Облік індивідуального темпу й стилю експерта дозволяє аналітикові знизити напруженість процедури добування знань. Типовою помилкою є нав'язування власного темпу й стилю. Необхідне вміння переймати темп і стиль спілкування експерта (ефект дзеркала).

На успішність також впливає довжина фраз, які вимовляє інженер по знаннях. Цей факт був установлений американськими вченими - лінгвістом Інгве й психологом Міллером при проведенні дослідження причин низького засвоєння команд на Військово-морському флоті США.

Причина була в довжині команд. Виявилось, що людина найкраще сприймає пропозиції глибиною (або довжиною) $7+2(-2)$ слова. Це число $(7+2)$ одержало назву "число Інгве—Міллера". Можна вважати його мірою "балакучості" мови. Досвідчені лектори використовують у лекції в основному короткі фрази, зменшуючи втрату інформації з 20—30% (у поганих лекторів) до 3-4%.

Облік невербальних компонентів спілкування. Більша частина інформації надходить до інженера по знаннях у формі пропозицій природною мовою. Однак зовнішня мова експерта є відтворення його внутрішньої мови (мислення), що набагато багатіше й багаторізнovidніше. При цьому для передачі цієї внутрішньої мови експерт використовує й невербальні засоби, такі як: інтонація, міміка, жести. Досвідчений інженер по знаннях намагається записувати по можливості в протоколи (у формі ремарок) цю додаткову інтонацію.

У цілому, невербальний компонент стилю спілкування важливий і для проблем контактного шару при встановленні контакту, коли по окремих жестах і вираженню особи експерта інженер по знаннях може встановити границю можливої "дружності" спілкування.

Методи. Конкретні методи докладно розглянуті в наступному розділі, виходячи з позиції, що метод повинен підходити до експертові як "ключ до замка".

3.3.4. Когнітивний шар

Когнітивні (від англ. *cognition* — пізнання) науки досліджують пізнавальні процеси людини з позицій можливості їхнього моделювання (психологія, нейрофізіологія, ергономіка, інженерія знань). Найменш досліджені на сьогоднішній день проблеми когнітивного шару, пов'язані з вивченням семантичного простору пам'яті експерта й реконструкцією його понятійної структури й моделі міркувань.

Основними факторами, що впливають на когнітивну адекватність поля знань, будуть:

- когнітивний стиль;
- семантична репрезентативність поля знань і концептуальної моделі.

3.3.4.1. Когнітивний стиль

Під *когнітивним стилем* людини розуміється сукупність критеріїв переваги при рішенні задач і пізнанні миру, специфічна для кожної людини. Когнітивний стиль визначає не стільки ефективність діяльності, скільки спосіб досягнення результату. Це спосіб пізнання, що дозволяє людям з різними здатностями домагатися однакових результатів у діяльності. Це система засобів й індивідуальних прийомів, до яких прибігає людина для організації своєї діяльності.

Інженерові по знаннях корисно вивчити й прогнозувати свій когнітивний стиль, а також стиль експерта. Особливо важливі такі характеристики когнітивного стилю, як:

- полезалежність - полenezалежність;
- імпульсивність - рефлексивність (рефлексивність);
- ригідність - гнучкість;
- діапазон когнітивної еквівалентності.

Полenezалежність. Ця властивість дозволяє людині акцентувати увагу лише на тих аспектах проблеми, які необхідні для рішення конкретної задачі, і вміння відкидати все зайве, тобто не залежати від тла або навколишню задачу шумового поля. Ця характеристика корелює з такими рисами особистості, як невербальний інтелект, аналітичність

мислення, здатність до розуміння суті. Очевидно, що крім того, що самому аналітикові необхідно мати високе значення параметра, полenezалежний експерт - це теж бажаний фактор. Однак доводиться враховувати, що більше мають потребу в спілкуванні полenezалежні люди, а тому вони й більше контактні.

Особливо корисні для спілкування гетерогенні (змішані) пари, наприклад "полenezалежний - полenezалежний". У літературі описані різні експерименти, що моделюють спілкування, що вимагає розуміння й спільної діяльності. Найбільш успішним у розумінні виявилися полenezалежні випробувані (92% успіху), для порівняння полезалежні давали 56% успіху.

Для спільної професійної діяльності важлива також гнучкість когнітивної організації, що пов'язана з полenezалежністю. Отже, більшу здатність до адекватного розуміння партнера демонструють суб'єкти з високою психологічною диференціацією, тобто полenezалежністю.

Полenezалежність є однією з характерних професійних рис когнітивного стилю найбільш кваліфікованих інженерів по знаннях. По деяких результатах чоловіки більше полenezалежні, ніж жінки.

Імпульсивність. Під *імпульсивністю* розуміється швидке ухвалення рішення (часто без його достатнього обґрунтування), а під *рефлексивністю* — схильність до розважливості. Рефлексивність за експериментальними даними корелює зі здатністю до формування понять і продуктивністю стратегій рішення логічних задач. Таким чином, і інженерові по знаннях, і експертові бажано бути рефлексивним, хоча власний стиль змінюється лише частково й з більшою напругою.

Ригідність. "*Ригідність — гнучкість*" характеризує здатність людини до зміни установок і точок зору відповідно до змінної ситуації. Ригідні люди не схильні міняти свої подання й структуру сприйняття, навпроти, гнучкі легко пристосовуються до нової обстановки. Очевидно, що якщо експерт ще може собі дозволити ригідність (що характерно для довго працюючих над однією проблемою фахівців, особливо старшого віку), то для інженера по знаннях ця характеристика когнітивного стилю явно протипоказана. Збільшення ригідності з віком відзначається багатьма психологами.

Діапазон когнітивної еквівалентності. *Когнітивна еквівалентність* характеризує здатність людини до розрізнення понять і розбивці їх на класи й підкласи. Чим уже діапазон когнітивної еквівалентності, тим більше тонку класифікацію здатний провести індивід, тим більша кількість ознак понять він може виділити. Звичайно в жінок діапазон когнітивної еквівалентності вужче, ніж більше широкий діапазон у чоловіків.

3.3.4.2. Семантична репрезентативність

Ця проблема має на увазі підхід, що виключає традиційне нав'язування експертові якоїсь моделі подань (наприклад, продукційної або фреймової), і, навпроти, змушує інженера по знаннях послідовно відтворювати модель миру експерта, використовуючи як неформальні методи, так і математичний апарат, наприклад багатомірне шкалювання. Проблема семантичної репрезентативності орієнтована на досягнення когнітивної адекватності поля знань і концептуальної моделі. У даний момент вона може бути сформульована як проблема "зіпсованого телефону" — можливі трансформації й втрати в ланцюзі передачі інформації.

3.3.5. Лінгвістичний аспект

Лінгвістичний аспект стосується досліджень мовних проблем, тому що мова - це основний засіб спілкування в процесі добування знань.

Відразу ж слід домовитися про, що оскільки книга присвячена проблемам розробки ЕС, те область розробки природно-мовних інтерфейсів і весь спектр проблем, пов'язаних з нею - лексичних, синтаксичних, семантичних, прагматичних і т.д. , не розглядається.

В інженерії знань можна виділити три шари лінгвістичних проблем:

- "загальний код";
- понятійна структура;
- словник користувача.

3.3.5.1. "Загальний код"

"Загальний код" вирішує проблему мовних розбіжностей між професійною термінологією експерта й повсякденною літературною мовою інженера по знаннях і включає наступні компоненти:

- загальнонаукова термінологія;
- спеціальні поняття із професійної літератури;
- елементи побутової мови;
- неологізми, сформовані за час спільної роботи;
- професійний жаргон й ін.

Деталізуючи схему спілкування, можна представити засобу спілкування як два потоки , що складаються з компонентів V_1 й V_2 — відповідно мови, на яких говорять аналітик й експерт (V_1 й V_2 — невербальні компоненти). Розходження мов V_1 й V_2 і спричиняється "мовний бар'єр" або "мовні розбіжності" у спілкуванні інженера по знаннях й експерта.

Ці дві мови є відбиттям "внутрішньої мови" експерта й аналітика, оскільки більшість психологів і лінгвістів вважають, що мова - це

основний засіб мислення, поряд з іншими знаковими системами "внутрішнього користування" (універсальний семантичний код - УСК , мови "змісту" , концептуальні мови й ін.).

Мова аналітика V_1 складається із трьох компонентів:

- загальнонаукової термінології з його "теоретичного багажу";
- термінів предметної області, які він почерпнув зі спеціальної літератури в період підготовки;
- побутової розмовної мови, яким користується аналітик.

Мова експерта V_2 включає:

- загальнонаукову термінологію;
- спеціальну термінологію, прийняту в предметній області;
- побутова мова;
- неологізми, створені експертом за час роботи, тобто його професійний жаргон.

Якщо вважати, що побутова й загальнонаукова мова у двох учасників спілкування приблизно збігається (хоча реально об'єм другого компонента в експерта істотно більше), то деяка загальна мова або код, яким необхідно виробити партнерам для успішної взаємодії, буде складатися з потоків.

Надалі цей загальний код перетвориться в деяку понятійну (семантичну) мережу, що є прообразом поля знань предметної області.

Вироблення загального коду починається з виписуванням аналітиком всіх термінів, уживаних експертом, і уточнення їхнього змісту. Фактично це складання словника предметної області. Потім треба групування термінів і вибір синонімів (слів, що означають те саме). Розробка загального коду закінчується складанням словника термінів предметної області з попереднім угрупованням їх за змістом, тобто за понятійною близькістю (це вже перший крок структурування знань).

На цьому етапі аналітик повинен з більшою увагою віднестися до всіх спеціальних термінів, намагаючись максимально вникнути в суть розв'язуваних проблем і термінологію. Освоєння аналітиком мови предметної області - перший рубіж на підступах до створення адекватної бази знань.

У семіотиці, науці про знакові системи, проблема інтерпретації є однією із центральних. Інтерпретація зв'язує "знак" й "означуваний предмет". Тільки в інтерпретації знак одержує зміст.

Увага до лінгвістичного аспекту проблеми добування знань сприяє зближенню образу 1 з образом 2 й інтерпретації I_1 з інтерпретацією I_2 , а слова "прилад X" перейдуть у дійсно "загальний"

код.

3.3.5.2. Понятійна структура

Особливості формування понятійної структури обумовлені встановленням постулатом когнітивної психології про взаємозв'язок понять у пам'яті людини й наявності семантичної мережі, що поєднує окремі терміни у фрагменти, фрагменти в сценарії й т.д. Побудова ієрархічної мережі понять, так називаної "піраміди знань", — найважливіша ланка в проектуванні інтелектуальних систем.

Більшість фахівців зі штучного інтелекту й когнітивної психології вважають, що основна особливість природного інтелекту й пам'яті зокрема - це зв'язаність всіх понять у деяку мережу. Тому для розробки бази знань і потрібний не словник, а "енциклопедія", у якій всі терміни пояснені в словникових статтях з посиланнями на інші терміни.

Таким чином, лінгвістична робота інженера по знаннях на даному шарі проблем полягає в побудові таких зв'язаних фрагментів за допомогою "зшивання" термінів. Фактично ця робота є підготовкою до етапу концептуалізації, де це "шиття" (по Шенку - КІП, концептуальна організація пам'яті) здобуває деякий закінчений вид.

При ретельній роботі аналітика й експерта в понятійних структурах починає проглядатися ієрархія понять. Такі структури мають найважливіше гносеологічне й дидактичне значення й останнім часом для них використовується спеціальний термін — *онтологія*. Варто помітити, що ця ієрархічна організація добре погодиться з теорією універсального предметного коду (УПК), відповідно до якої при мисленні використовуються не мовні конструкції, а їхні коди у формі деяких абстракцій, що в загальному погодяться з результатами когнітивної психології.

Онтологія або ієрархія абстракцій - це глобальна схема, що може бути покладена в основу концептуального аналізу структури знань будь-якої предметної області. Лінгвістичний еквівалент ієрархії - ієрархія понять, яку необхідно побудувати в понятійній структурі, формованої інженером по знаннях.

Варто підкреслити, що робота зі складання словника й понятійної структури вимагає лінгвістичного "чуття", легкості маніпулювання термінами й багатого словникового запасу інженера по знаннях, тому що найчастіше аналітик змушений самостійно розробляти словник ознак. Чим багатшим й виразнішим виходить загальний код, тим повніша база знань.

Аналітик змушений увесь час пам'ятати про труднощі передачі образів і подань у вербальній

формі. Корисними тут виявляються властивості багатозначності слів природної мови. Часто інженерів по знаннях доводиться підказувати слова й вираження експертів, і такі нові лексичні конструкції виявляються корисними.

Здатність до словесної інтерпретації залежить і від підлоги аналітика. Установлено, що традиційно жінки надають більшу значимість невербальним компонентам спілкування, а у вербальні мають більше великий алфавіт ознак. І взагалі, існують полові розходження сприйняття не тільки в побутовій сфері, що очевидно, але й у професійній. Отже, в експерта-чоловіка й в експерта-жінки можуть істотно відрізнитися алфавіти для вербалізації ознак сприйманих об'єктів.

3.3.5.3. Словник користувача

Формування загального коду й розробка понятійної структури спрямовані на створення адекватної бази знань. Однак часто професійний рівень кінцевого користувача не дозволяє йому застосувати спеціальна мова предметної області в повному об'ємі.

Несподіваними для починаючих розробників є проблеми формування окремого словника для створення дружнього інтерфейсу з користувачем ЕС. Необхідні спеціальні прийоми, що збільшують "прозорість" і доступність системи. Для розробки користувацького інтерфейсу потрібна додаткова доробка словника загального коду з виправленням на доступність й "прозорість" системи.

Так, при розробці експертної системи по психодіагностиці АВТАНТЕСТ довелося розробити два словники термінів - один для психологів-професіоналів, другий - для неспеціалістів (випробуваних). Оскільки результат психодіагностичного тестування завжди цікавий випробуваному, йому видається лістинг із психологічним висновком загальнолітературною мовою без уживання спеціальних термінів. Цікаво, що при впровадженні системи використався в основному цей другий словник; навіть професійні психологи воліли одержувати тексти повсякденною мовою.

3.3.6. Гносеологічний аспект

Гносеологічний аспект добування знань поєднує методологічні проблеми одержання нового наукового знання, оскільки при створенні БЗ експерт часто вперше формулює деякі закономірності, що до того моменту складають його особистий досвід. Гносеологія це розділ філософії, пов'язаний з теорією пізнання, або теорією відбиття дійсності у свідомості людини.

Інженерія знань як наука, якщо можна так виразитися, двічі гносеологічна — спочатку

дійсність (*ПРО*) відбивається у свідомості експерта (M_1), а потім діяльність і досвід експерта інтерпретуються свідомістю інженера по знаннях (M_2), що служить уже основою для побудови третьої інтерпретації (P_2) — поля знань експертної системи. Процес пізнання, по суті, спрямований на створення внутрішньої репрезентації навколишнього світу у свідомості людини.

Якщо описати процеси інтерпретації I_2 й I_3 у термінології, уведеної в розд. 1, то ми маємо справу з перетворенням експертного знання й теоретичного (книжкового) досвіду Z_1 у поле знань Z_2 , що є матеріалізація моделі миру M_2 інженера по знаннях.

У процесі добування знань аналітика в основному цікавить компонент Z_1 , пов'язаний з неканонічними індивідуальними знаннями експертів, оскільки предметні області, що вимагають саме такого типу знань, вважаються найбільш сприйнятливими до впровадження експертних систем. Ці області звичайно називають емпіричними, тому що в них накопичений великий об'єм окремих емпіричних фактів і спостережень, у той час як їхнє теоретичне узагальнення - питання майбутнього.

Якщо вважати, що інженер по знаннях витягає тільки фрагмент Z_1' , тобто частина із системи знань експерта Z_1 , то його задача, по-перше, намагатися, щоб структура Z_1' відповідала Z_1 й, по-друге, щоб Z_1' як можна більш повно відбивав Z_1 .

Познання часто супроводжується створенням нових понять і теорій. Іноді експерт породжує нові знання прямо в ході бесіди з аналітиком. Така генерація знань корисна й самому експертові, що до того моменту міг не усвідомлювати ряд співвідношень і закономірностей предметної області. Аналітикові може допомогти отут й інструментарій системної методології, що дозволяє використати відомі принципи логіки наукових досліджень, понятійної ієрархії науки. Ця методологія змушує його за часткою завжди прагнути побачити загальне, тобто будувати ланцюжка:

факт -> узагальнений факт -> емпіричний закон
-> теоретичний закон.

Не завжди вдається дійти до останньої ланки цього ланцюжка, але вже саме прагнення до просування буває надзвичайно плідним. Такий підхід повністю погодиться зі структурою самого знання, що має два рівні:

- емпіричний (спостереження, явища);
- теоретичний (закони, абстракції, узагальнення).

Основними методологічними критеріями стрункості виявленої системи знань можна вважати:

- внутрішню погодженість;
- системність;
- об'єктивність;
- історизм.

3.3.6.1. Внутрішня погодженість

На перший погляд критерій внутрішньої погодженості знання не відповідає реальним характеристикам, що описують знання. Будь-яке емпіричне знання *суперечливо й неповно*. Ці характеристики емпіричних знань підкреслюють його "багатоукладність" - настільки часто факти не погодяться один з одним, визначення суперечать, критерії дифузні й т.д. Аналітикові, що знає особливості емпіричного знання, доводиться згладжувати ці "шорсткості" емпірики. Можлива суперечливість емпіричного знання - природний наслідок з основних законів діалектики, і протиріччя ці не завжди повинні дозволятися в поле знань, а навпроти, саме протиріччя служать найчастіше відправною крапкою в міркуваннях експертів.

Неповнота знання пов'язана з неможливістю повного опису предметної області. Задача аналітика цю неповноту обмежити певними рамками "повноти", тобто звужити границі предметної області, або ввести ряд обмежень і допущень, що спрощують проблему.

Крім того, емпіричне знання є модальним. *Модальність знання* означає можливість його існування в різних категоріях, тобто в конструкціях існування й повинності. Таким чином, частина закономірностей можлива, інша обов'язкова й т.ін. Крім того, доводиться розрізняти такі відтінки модальності, як:

- експерт знає, що...;
- експерт думає, що...;
- експерт хоче, щоб...;
- експерт вважає, що...

3.3.6.2. Системність

Системно-структурний підхід до пізнання (що сходить ще до Гегеля) орієнтує аналітика на розгляд будь-якої предметної області з позицій закономірностей системного цілого й взаємодії складових його частин.

Сучасний структуралізм виходить із багаторівневої ієрархічної організації будь-якого об'єкта, тобто всі процеси і явища можна розглядати як множину більше дрібних підмножин (ознак, деталей) і, навпаки, будь-які об'єкти можна (і потрібно) розглядати як елементи більше високих класів узагальнень. Наприклад, системний погляд на проблематику структурування знань дозволяє побачити його ієрархічну організацію.

3.3.6.3. Об'єктивність

Процес пізнання глибоко суб'єктивний, тобто він істотно залежить від особливостей самого суб'єкта, що пізнає. "Факти існують для одного ока й відсутні для іншого" (Віппер). Таким чином, суб'єктивність починається вже з опису фактів і збільшується в міру поглиблення ідеалізації об'єктів.

Отже, більш коректно говорити про глибину розуміння, чим про об'єктивність знання. Розуміння - це співтворчість, процес тлумачення об'єкта з погляду суб'єкта. Це складний і неоднозначний процес, що відбувається в глибинах людської свідомості й вимагає мобілізації всіх інтелектуальних й емоційних здатностей людини. Всі свої зусилля аналітик повинен зосередити на розумінні проблеми.

У психології відомий результат, що підтверджує факт, що люди, швидко й успішно вирішальні інтелектуальні задачі, більшу частину часу витрачають на розуміння її, у той час як погані розв'язувачі швидко приступають до пошуків рішення й найчастіше не можуть його знайти.

3.3.6.4. Історизм

Цей критерій пов'язаний з розвитком. Пізнання сьогодення - є пізнання його минулого, що породило. І хоча більшість експертних систем дають "горизонтальний" зріз знань - без обліку часу (у статистиці), інженер по знаннях повинен завжди розглядати процеси з урахуванням тимчасових змін - як зв'язок з минулим, так і зв'язок з майбутнім. Наприклад, структура поля знань і база знань повинні допускати підстроювання й корекцію як у період розробки, так і під час експлуатації ЕС.

3.3.7. Методологія процесу одержання нового знання

Методологічно діяльність аналітика може бути представлена як деяка послідовність етапів:

- E_1: опис й узагальнення фактів;
- E_2: установлення логічних і математичних зв'язків, дедукція й індукція законів;

- E_3: побудова моделі;
- E_4: пояснення й прогнозування явищ.

3.3.7.1. E_1: опис й узагальнення фактів

Старанність і повнота ведення протоколів під час процесу добування й пунктуальна "домашня робота" над ними - от застава продуктивного першого етапу пізнання й матеріал для опису й узагальнення фактів.

На практиці виявляється важким дотримуватися принципів об'єктивності й системності, описаних вище. Найчастіше на цьому етапі факти просто збирають й як би кидають в "загальний мішок"; досвідчений інженер по знаннях часто відразу намагається знайти "поличку" або "шухлядка" для кожного факту, тим самим підспудно готуючись до етапу концептуалізації.

3.3.7.2. E_2: установлення зв'язків

У пам'яті експерта всі поняття вв'язані й закономірності встановлені, хоча часто й неявно, задача інженера - виявити каркас умовиводів експерта. Реконструюючи міркування експерта, інженер по знаннях може опиратися на дві найбільш популярні теорії мислення - логічну й асоціативну. При цьому якщо логічна теорія завдяки гарячим шанувальникам в особі математиків широко цитується й усіляко експлуатується в роботах по штучному інтелекту, те друг, асоціативна, набагато менш відома й популярна, хоча має також древніх коріння. Так, Р. Фейнман у своїх "Лекціях по фізиці" відзначає, що у фізику як і раніше переважним є вавилонський, а не грецький метод побудови знань. Відомо, що давньосхідні математики вміли робити складні обчислення, але формули їх не були логічно вв'язані. Навпроти, грецька математика дедуктивна (наприклад, "Початку" Евкліда).

Традиційна логіка формує критерії, які гарантують точність, валідність, несуперечність загальних понять міркувань і висновків. Її основи закладені ще в "Органоне" Аристотеля в IV в. до н.е. Великий внесок у розвиток логіки вніс Джон Стюарт Міль (1806-1873).

Інженер по знаннях і сам використовує операції традиційної логіки й виділяє їх у схемі міркувань експерта. Це наступні операції:

- визначення;
- порівняння й розрізнення;
- аналіз;
- абстрагування;
- узагальнення;
- класифікація;
- категоризація;
- утворення суджень;
- умовивід;

- складання силогізмів і т.д.

Однак краса й стрункість логічної теорії не повинні заслоняти той, що людина рідко мислить у категоріях математичної логіки.

Теорія асоціацій представляє мислення як ланцюжок ідей, зв'язаних загальними поняттями. Основними операціями такого мислення є:

- асоціації, придбані на основі різних зв'язків;
- пригадування минулого досвіду;
- проби й помилки з випадковими успіхами;
- звичні ("автоматичні") реакції й ін.

Однак ці дві теорії не вичерпують усього різноманіття психологічних шкіл. Великий інтерес для інженерії знань може представляти гештальтпсихологія. Одним з її засновників є видатний німецький психолог М. Вертгеймер (1880-1943). Під гештальтом (ньому. Gestalt) розуміється принцип цілісності сприйняття - як основа мислення. Гештальтпсихологи намагаються у всьому виділяти якийсь цілісний образ або структуру як базис для розуміння процесів й явищ навколишнього світу. Ця теорія близька теорії фреймів й об'єктному підходу й спрямована на збагнення глибинного знання, що характеризується стабільністю й симетрією. При цьому важливий так званий "центр ситуації", щодо якого розвивається знання про предметну область.

Для інженера по знаннях це означає, що, виявляючи різні фрагменти знань, він не повинен забувати про головний, про гештальт фрагмент, що впливає на інші компоненти й зв'язує їх у деяку структурну одиницю. Гештальтом може бути якийсь головний принцип, або ідея, або гіпотеза експерта, або його віра в силу якихось окремих концепцій. Цей принцип рідко формулюється експертом явно, він завжди як би за "кадром", і мистецтво інженера по знаннях - виявити цей основний гештальт експерта.

У гештальт теорії існує закон "прагнення до гарного гештальту", відповідно до якого структури свідомості прагнуть до гармонії, зв'язності, простоті. Це близько до стародавнього класичного принципу "бритви Оккама" - "сутності не повинні множитися без необхідності" - і формулюється як принцип прегнантності Вертгеймера: "Організація поля має тенденцію бути настільки простий й ясною, наскільки дозволяють дані умови". Міркування про гештальте підводять впритул до третього етапу в структурі пізнання.

3.3.7.3. E_3: побудова моделі

Необхідна спеціалізована мова, за допомогою якого можна описувати й конструювати ті ідеалізовані моделі миру, які виникають у процесі мислення. Мова цей створюється поступово за

допомогою категоріального апарата, прийнятого у відповідній предметній області, а також формально-знакових засобів математики й логіки. Для емпіричних предметних областей така мова поки не розроблена й поле знань, що напівформалізованим способом опише аналітик, може бути першим кроком до створення такої мови.

Будь-яке пізнавальне відбиття містить у собі умовність, тобто спрощення й ідеалізацію. Інженерові по знаннях необхідне оволодіння такими специфічними гносеологічними прийомами, як ідеалізація, огрубіння, абстрагування, які дозволяють адекватно відображати в моделі реальну картину миру. Ці прийоми доводять властивості й ознаки об'єктів до меж, що дозволяють відтворювати закони дійсності в більше лаконічному виді (без впливу несуттєвих деталей).

На тернистому шляху пізнання перевірений діалектичний підхід виявляється кращим "поводирем". Інженер по знаннях, що прагне пізнати проблемну область, повинен бути готів постійно змінювати свої вже, що затвердилися способи, сприйняття й оцінки миру й навіть відмовлятися від них. При цьому ретельніше всього варто перевіряти правильність суджень, які здаються самими очевидними.

3.3.7.4. E_4: пояснення й пророкування явищ

Цей завершальний етап є одночасно й частковим критерієм істинності отриманого знання. Якщо виявлена система знань експерта повна й об'єктивна, то на її підставі можна робити прогнози й пояснювати будь-які явища з даної предметної області. Звичайно бази знань ЕС страждають фрагментарністю й модульністю (незв'язаністю) компонентів. Все це не дозволяє створювати дійсно інтелектуальні системи, які, рівняючись на людину, могли б пророкувати нові закономірності й пояснювати випадки, не зазначені в явному виді в базі. Виключенням тут є навчальні системи, які орієнтовані на генерацію нових знань і на "пророкування".

Пропонована методологія озброює аналітика апаратом, що дозволяє уникнути традиційних помилок, що приводять до неповноти, суперечливості, фрагментарності БЗ, і вказує напрямок, у якому необхідно рухатися розробникам. І хоча на сьогоднішній день більшість БЗ проробляються лише до етапу E_3, знання повної схеми збагачує й поглиблює процес проектування.

3.4. Методи практичного добування знань

Розглянемо, якими практичними методами можна одержати знання. У попередніх розділах за

замовчуванням передбачалося, що ця деяка взаємодія інженера по знаннях й експерта у формі безпосереднього живого спілкування. У літературі згадуються близько 15 ручних (неавтоматизованих) методів добування й більше 20 автоматизованих методів придбання й формування знань.

Розглянемо класифікацію методів добування знань, що дозволить інженерам по знаннях залежно від конкретної задачі й ситуації вибрати підходящий метод.

При такій класифікації як основний принцип ділення виступає джерело знань. Відповідно всі методи діляться на:

- комунікативні;
- текстологічні.

Визначення 3.6

Комунікативні методи добування знань — це набір прийомів і процедур, що припускають контакт інженера по знаннях з безпосереднім джерелом знань — експертом, а **текстологічні** включають методи добування знань із документів (методик, посібників, керівництв) і спеціальної літератури (статей, монографій, підручників).

Поділ цих груп методів на верхньому рівні класифікації не означає їхньої антагоністичності, звичайно інженер по знаннях комбінує різні методи, наприклад спочатку вивчає літературу, потім розмовляє з експертами, або навпаки.

У свою чергу, комунікативні методи можна також розділити на дві групи: *пасивні* й *активні*.

Пасивні методи мають на увазі, що провідна роль у процедурі добування передається експертові, а інженер по знаннях тільки протоколює міркування експерта під час його реальної роботи із прийняття рішень або записує те, що експерт вважає потрібним самостійно розповісти у формі лекції. В активних методах, навпроти, ініціатива повністю в руках інженера по знаннях, що активно контактує з експертом різними способами - в іграх, діалогах, бесідах за круглим столом і т.д.

Треба ще раз підкреслити, що й активні й пасивні методи можуть чергуватися навіть у рамках одного сеансу добування знань. Наприклад, якщо інженер по знаннях соромливий і не має великого досвіду, те спочатку він може використати пасивні методи, а поступово, ближче знайомлячи з експертом, захоплювати ініціативу й переходити "у настання".

Пасивні методи на перший погляд досить прості, але насправді жадають від інженера по знаннях уміння чітко аналізувати потік свідомості

експерта й виявляти в ньому значимі фрагменти знань. Відсутність зворотного зв'язку (пасивність інженера по знаннях) значно послабляє ефективність цих методів, чим і порозумівається їх звичайно допоміжна роль при активних методах.

Активні методи можна розділити на дві групи залежно від числа експертів, що віддають свої знання. Якщо їхнє число більше одного, то доцільно крім серії індивідуальних контактів з кожним застосовувати й методи групових обговорень предметної області.

Такі групові методи звичайно активізують мислення учасників дискусій і дозволяють виявляти досить нетривіальні аспекти їхніх знань. У свою чергу, індивідуальні методи на сьогоднішній день залишаються ведучими, оскільки настільки делікатна процедура, як "відібрання знань", не терпить зайвих свідків.

Ігрові методи зараз широко використовуються в соціології, економіці, менеджменті, педагогіці для підготовки керівників, учителів, лікарів й інших фахівців. Гра - це особлива форма діяльності й творчості, де людина розкріпається й відчуває себе набагато вільніше, ніж у звичайній трудовій діяльності.

На вибір методу впливають три фактори: особистісні особливості інженера по знаннях, особистісні особливості експерта й характеристика предметної області.

Одна з можливих класифікацій людей по психологічних характеристиках ділить усіх на три типи:

- мислитель (пізнавальний тип);
- співрозмовник (емоційно-комунікативний тип);
- практик (практичний тип).

Мислителі орієнтовані на інтелектуальну роботу, навчання, теоретичні узагальнення й мають такі характеристики когнітивного стилю, як полонезалежність і рефлексивність.

Співрозмовники - це товариські, відкриті люди, готові до співробітництва. Практики віддають перевагу дії розмовам, добре реалізують задуми інших, спрямовані на результативність роботи.

Для характеристики предметних областей можна запропонувати наступну класифікацію:

- добре документовані;
- середньо документовані;
- слабо документовані.

Ця класифікація пов'язана зі співвідношенням двох видів знань Z_1 й Z_2 , уведених у розд. 1.1, де Z_1 — це експертне "особисте" знання, а Z_2 — матеріалізоване в книгах "загальне" знання в даній конкретній області.

Крім цього, предметні області можна розділити

за критерієм структурованості знань. Під структурованістю будемо розуміти ступінь теоретичного осмислення й виявлення основних закономірностей і принципів, що діють у даній предметній області. І хоча ЕС традиційно застосовуються в слабо структурованих предметних областях, зараз спостерігається тенденція розширення сфери впровадження експертних систем.

По ступені структурованості знань предметні області можуть бути:

- *добре структурованими* — із чіткою аксіоматизацією, широким застосуванням математичного апарата, усталеною термінологією;
- *середньо структурованими* — з термінологією, що визначилася, що розвивається теорією, явними взаємозв'язками між явищами;
- *слабо структурованими* — з розмитими визначеннями, багатою емпірикою, схованими взаємозв'язками, з більшою кількістю "білих плям".

Уведені в даному розділі класифікації методів і предметних областей допоможуть інженерові по знаннях, чітко визначивши свою предметну область, співвіднести її із запропонованими типами й намітити відповідний метод або групу методів

добування знань. Однак, швидше за все, реальна робота повністю закреслить його вибір, і виявиться, що його добре документована область є слабо документованою, а метод спостережень треба терміново замінити іграми!

Така реальна складність процедур добування знань, що вимагає спеціальної підготовки аналітика. Краща підготовка — це підвищення кваліфікації на спеціальних курсах. Самопідготовка включає:

- загальну;
- спеціальну (частково може збігатися із пропонованою в підготовкою до журналістського інтерв'ю), доповнену у зв'язку зі специфікою інженерії знань;
- конкретну (базується на прочитанні спеціальних текстів, обраних разом з експертами з деякого "базового" списку літератури, що поступово введе аналітика в предметну область. У цьому списку можуть бути підручники для початківців, глави й фрагменти з монографій, популярні видання);
- психологічну.

Підготовка забирає різний час залежно від ступеня професіоналізму аналітика, але в кожному разі вона необхідна, тому що трохи зменшує ймовірність самого нераціонального методу - методу проб і помилок.

Тема 4. Практикум з інженерії знань

4.1. Практикум по інженерії знань

4.1.1. Текстологічні методи

З добуванням знань із текстів ми зіштовхуємося щодня. Цей спосіб завжди повинен передувати комунікативним методам, підготовляючи аналітика й знайомлячи його з термінологією й основними ідеями.

Група текстологічних методів поєднує методи добування знань, засновані на вивченні спеціальних текстів з підручників, монографій, статей, методик й інших носіїв професійних знань.

У буквальному значенні текстологічні методи не ставляться до текстології, науки, що народилася в руслі філології з метою критичного прочитання літературних текстів, вивчення й інтерпретації джерел з вузько прикладною задачею - підготовки текстів до видання.

Зараз текстологія розширила свої границі включенням аспектів суміжних наук - герменевтики (науки правильного тлумачення древніх текстів - біблії, античних рукописів й ін.), семіотики, психолінгвістики й ін.

Текстологічні методи добування знань, безумовно, використовуючи основні положення текстології, відрізняються принципово від її методології,

по-перше, характером і природою своїх джерел (професійна спеціальна література, а не художня, живуча по своїх особливих законах),

а по-друге, твердою прагматичною спрямованістю добування конкретних професійних знань.

Серед методів добування знань ця група є найменш розробленою, по ній практично немає ніякої бібліографії, тому подальший виклад є як би введенням у методи вивчення текстів у тому вигляді, як це представляють автори.

4.1.1.1. Розуміння тексту

Задачу добування знань із текстів можна сформулювати як задачу розуміння й виділення змісту тексту. Сам текст природною мовою є лише провідником змісту, а задум і знання автора лежать у вторинній структурі (змістовній структурі або макроструктурі тексту), що набуває над природним текстом, або "текст не містить і не передає зміст, а є лише інструментом для автора тексту".

При цьому можна виділити дві такі значеннєві структури:

M_1 — зміст, що намагався закласти автор, це його модель миру, і

M_2 — зміст, що осягає читач, у цьому випадку інженер по знаннях у процесі інтерпретації І. При цьому T — це словесне одіяння M_1 , тобто результат вербалізації V .

Складність процесу полягає в принциповій неможливості збігу знань, що утворюють M_1 і M_2 , через те, що M_1 утвориться за рахунок всієї сукупності подань, потреб, інтересів і досвіду автора, лише мала частина яких знаходиться висвітлення в тексті T . Відповідно й M_2 утвориться в процесі інтерпретації тексту T за рахунок залучення всієї сукупності професійного й людського багажу читача. Таким чином, два інженери по знаннях витягнуть із одного T дві різні моделі: M'_1 і M'_2 .

Встає задача: з'ясувати, за рахунок чого можна досягти максимальної адекватності M_1 і M_2 , пам'ятаючи при цьому, що розуміння завжди відносно, оскільки це синтез двох змістів "своє — чуже".

Розглянемо докладніше, які джерела харчують модель M_1 і створюють текст T . Можна виділити два компоненти будь-якого професійного тексту:

- первинний фактичний матеріал;
- система понять, що склалася в даній предметній області в момент створення тексту.

На додаток до цього, на наш погляд, крім об'єктивних даних експериментів і спостережень, у тексті обов'язково присутні суб'єктивні погляди автора, результат його особистого досвіду, а також деякі "загальні місця" або "вода". Крім цього, будь-який текст містить запозичення з інших джерел (методики, вказівки, документи, статті, монографії) і т.д.

При добуванні знань аналітикові, що інтерпретує текст, доводиться вирішувати задачу декомпозиції цього тексту на перераховані вище компоненти для виділення істинно значимих для реалізації бази знань фрагментів. Складність інтерпретації професійних текстів полягає ще й у тім, що будь-який текст набуває сенсу тільки в контексті, де під контекстом розуміється оточення, у яке "занурений" текст.

Розрізняють мікро- і макроконтекст. *Мікроконтекст* — це найближче оточення тексту. Так, пропозиція одержує зміст у контексті абзацу, абзац у

контексті глави й т.д. *Макроконтекст* — це вся система знань, пов'язана із предметною областю (тобто знання про особливості й властивості, явно не зазначених у тексті). Інакше кажучи, будь-яке знання знаходить зміст у контексті деякого мета знання.

Розглянемо докладніше про центральну ланку процедури добування знання - про розуміння тексту. Класичним у текстології є визначення німецького філософа й мовознавця Гумбольдта :

"...Люди розуміють один одного не тому, що передають співрозмовникові знаки предметів, і навіть не тому, що взаємно набудовують один одного на точне й повне відтворення ідентичного поняття, а тому, що взаємно зачіпають друг у другу те саме ланка ланцюга почуттєвих подань і зачатків внутрішніх понять, доторкаються до тим самим клавіш інструмента свого духу, завдяки чому в кожного спалахує у свідомості відповідні, але не тотожні змісти".

Говорячи мовою сучасного мовознавства, *розуміння* — це формування "другого тексту", тобто семантичної структури (понятійної структури). У нашій термінології — це спроба відтворення семантичної структури M_1 у процесі формування моделі M_2 , тобто це перший крок структурування знань.

Як відбувається процес розуміння? Одна з можливих схем викладена надалі. Ми внесли кілька змін у цю схему у зв'язку з тим, що в ній трактується розуміння тексту іноземною мовою, а нас цікавить розуміння тексту в новій для суб'єкта, що пізнає, предметної області. Крім цього, доповнимо її деякими положеннями герменевтики. У цілому отримана схема погодиться зі стратегією вивчення всього нового.

Основними моментами розуміння тексту є наступні кроки:

1. Висування попередньої гіпотези про зміст усього тексту (передбачення).
2. Визначення значень незрозумілих слів (тобто спеціальної термінології).
3. Виникнення загальної гіпотези про зміст тексту (про знання).
4. Уточнення значення термінів й інтерпретація окремих фрагментів тексту під впливом загальної гіпотези (від цілого до частин).
5. Формування деякої змістовної структури тексту за рахунок установлення внутрішніх зв'язків між окремими важливими (ключовими) словами й фрагментами, а також за рахунок утворення абстрактних понять, що узагальнюють конкретні фрагменти знань.
6. Коректування загальної гіпотези щодо

фрагментів, що втримуються в тексті, знань (від частин до цілого).

7. Прийняття основної гіпотези, тобто формування M_2 .

Слід відзначити наявність як дедуктивної (від цілого до частин), так й індуктивної (від частин до цілого) складового процесу розуміння. Такий двоєдиний підхід дозволяє охоплювати текст як значеннєва єдність особливого роду з його основними ознаками, такими як зв'язність, цілісність, закінченість й ін.

4.1.1.2. Змістовна структура тексту

Центральними моментами процесу I є кроки 5 й 7, тобто формування змістовної структури або виділення "опорних", ключових, слів або "змістовних віх", а також заключне зв'язування "змістовних віх" у єдину семантичну структуру.

При аналізі тексту важливе виявлення внутрішніх зв'язків між окремими елементами тексту й поняттями. Традиційно виділяють два види зв'язків у тексті — *експліцитні* (або явні зв'язки), які виражаються в зовнішнім дробленні тексту, і *імпліцитні* (сховані зв'язки).

Експліцитні зв'язку ділять текст на параграфи за допомогою перерахування компонентів, вступних слів (або конекторів) типу "по-перше..., по-друге..., однак і т.д."

Імпліцитні, або внутрішні, зв'язку між окремими "змістовними віхами" викликають основне утруднення при розумінні.

Отже, семантична структура тексту утвориться у свідомості суб'єкта, що пізнає, за допомогою знань про мову, знань про світ, а також загальних (фонових) знань у тій предметній області, який присвячений текст. "Тексти пишуть для присвячених". Інакше кажучи, якщо текст не є науково-популярним, то для його адекватного прочитання потрібно деяка підготовка, наприклад читання популярних підручників на рівні менеджерів середньої ланки.

Процес I — це складний, що не піддається формалізації процес, на який істотно впливають такі чисто індивідуальні компоненти, як когнітивний стиль пізнання, інтелектуальні характеристики й ін.

Одна з гіпотез лінгвостатистики про те, що найчастіше вживані слова є найбільш важливими з погляду змісту тексту, тобто відбивають його тематичну структуру, підтверджується при роботі із професійними текстами.

Варто сказати кілька слів про те, чому ми виділяємо три види текстологічних методів:

- аналіз спеціальної літератури;

- аналіз підручників;
- аналіз методик.

Перераховані три методи відрізняються, по-перше, по ступені концентрованості спеціальних знань, і, по-друге, по співвідношенню спеціальних і фонових знань. Найбільш простим методом є аналіз підручників, у яких логіка викладу звичайно відповідає логіці предмета й тому структура такого тексту буде, напевно, більше значима, чим структура тексту якої-небудь спеціальної статті. Аналіз методик утруднений саме стислістю викладу й практичною відсутністю коментарів, тобто фонових знань, що полегшують розуміння для неспеціалістів. Тому можна рекомендувати для практичної роботи комбінацію перерахованих методів.

4.1.1.3. Алгоритм добування знань із тексту

На закінчення запропонуємо одну з можливих практичних методик аналізу текстів з метою добування й структурування знань.

- Складання "базового" списку літератури для ознайомлення із предметною областю й читання за списком.
- Перше знайомство з текстом (швидке прочитання) з виписуванням незнайомих слів.
- Консультації з фахівцями або залучення довідкової літератури для їхнього розуміння.
- Уважне (друге) прочитання тексту з виписуванням наборів ключових слів (ИКС), тобто виділення "змістовних віх" (компресія тексту).
- Визначення зв'язків між НКС, розробка семантичної структури тексту у формі графа або "стислого" тексту (реферату), фактично формування поля знань.
- Третє прочитання тексту й корекція поля знань при необхідності.

4.1.2. Комунікативні методи

4.1.2.1. Пасивні методи

Термін "пасивні" не повинен викликати ілюзій, у реальності ж пасивні методи жадають від інженера по знаннях не меншої віддачі, чим такі активні методи, як гри й діалог.

Визначення 2.7

Пасивні методи добування знань включають методи, де провідна роль у процедурі добування фактично передається експертові, а інженер по знаннях тільки фіксує міркування експерта під час роботи із прийняття рішень.

Відповідно до класифікації до цієї групи ставляться:

- спостереження;

- аналіз протоколів "думок уголос";
- лекції.

4.1.2.1.1. Спостереження

У процесі спостережень інженер по знаннях перебуває безпосередньо поруч із експертом під час його професійної діяльності або імітації цієї діяльності. При підготовці до сеансу добування експертові необхідно пояснити мету спостережень і попросити максимально коментувати свої дії.

Під час сеансу аналітик записує всі дії експерта, його репліки й пояснення. Корисної може виявитися й відеозапис у реальному масштабі часу, якщо експерт погодиться. Неодмінна умова цього методу - невтручання аналітика в роботу експерта хоча б спочатку. Саме метод спостережень є єдино "чистим" методом, що виключає втручання інженера по знаннях і нав'язування їм якихось своїх структур подань.

Існують два основні різновиди проведення спостережень:

- спостереження за реальним процесом;
- спостереження за імітацією процесу.

Звичайно використовуються обидві різновиди. Спочатку інженерові по знаннях корисно спостерігати за реальним процесом, щоб глибше зрозуміти предметну область і відзначити всі зовнішні особливості процесу ухвалення рішення. Це необхідно для проектування ефективного інтерфейсу користувача. Адже майбутня ЕС повинна працювати саме в контексті такого реального виробничого процесу. Крім того, тільки спостереження дозволить аналітикові побачити предметну область, а, як відомо, "краще один раз побачити, чим сто разів почути".

Спостереження за імітацією процесу проводять звичайно також за робочим місцем експерта, але сам процес діяльності запускається спеціально для аналітика. Перевага цього різновиду полягає в тому, що експерт менш напружений, чим у першому варіанті, коли він працює на "два фронти" - і веде професійну діяльність, і демонструє її. Недолік збігається з перевагою - саме менша напруженість експерта може вплинути на результат - раз робота несправжня, те й рішення може відрізнитися від сьогодення.

Спостереження за імітацією проводять також й у тих випадках, коли спостереження за реальним процесом за якимись причинами неможливі (наприклад, професійна етика лікаря-психіатра може не допускати присутності стороннього на прийомі).

Сеанси спостережень можуть зажадати від інженера по знаннях:

- оволодіння технікою стенографії для фіксації дій експерта в реальному масштабі часу;

- ознайомлення з методиками хронометражу для чіткого структурування виробничого процесу за часом;
- розвитку навичок "читання по очах", тобто спостережливості до жестів, міміки й іншим невербальним компонентам спілкування;
- серйозного попереднього знайомства із предметною областю, тому що через відсутність "зворотного зв'язку" іноді багато чого незрозуміло в діях експертів.

Протоколи спостережень після сеансів у ході домашньої роботи ретельно розшифровуються, а потім обговорюються з експертом.

Таким чином, спостереження - один з найпоширеніших методів добування знань на початкових етапах розробки. Звичайно він застосовується не самостійно, а в сукупності з іншими методами.

4.1.2.1.2. Аналіз протоколів "думок у голос"

Протоколювання "думок у голос", або "вербальні звіти", відрізняється від спостережень тим, що експерта просять не просто прокоментувати свої дії й рішення, але й пояснити, як це рішення було знайдено, тобто продемонструвати весь ланцюжок своїх міркувань. Під час міркувань експерта всього його слова, весь "потік свідомості" протоколюється інженером по знаннях, при цьому корисно відзначити навіть паузи й вигук.

Питання про використання для цієї мети магнітофонів і диктофонів є дискусійним, оскільки магнітофон іноді паралізує діє на експерта, руйнуючи атмосферу довіри, що може й повинна виникати при безпосереднім спілкуванні.

Основними труднощами при протоколюванні "думок у голос" є принципова складність для будь-якої людини пояснити, як він думає. При цьому існують експериментальні психологічні докази того факту, що люди не завжди в стані вірогідно описувати розумові процеси. Крім того, частина знань, що зберігаються в невербальній формі (наприклад, різні процедурні знання типу "як зав'язувати шнурки"), взагалі слабо корелюють із їхнім словесним описом. Автор теорії фреймів М. Мінський уважає, що "тільки як виключення, а не як правило, людина може пояснити те, що він знає". Однак існують люди, схильні до рефлексії, для яких ця робота є цілком доступною. Отже, така характеристика когнітивного стилю, як рефлексивність, є для експерта більш ніж бажаною.

Розшифровка отриманих протоколів виробляється інженером по знаннях самостійно, з корекціями на наступних сеансах добування знань. Вдало проведене протоколювання "думок у голос" є одним з найбільш ефективних методів добування, оскільки в ньому експерт може виявити себе максимально яскраво, він нічим не скований, ніхто

йому не заважає, він як би вільно ширяє в потоці власних умовиводів і міркувань. Він може тут блиснути ерудицією, продемонструвати глибину своїх пізнань. Для великої кількості експертів це самий приємний й утішний спосіб добування знань.

Від інженера по знаннях метод "думки вголос" вимагає тих же вмінь, що й метод спостережень. Звичайно "думки вголос" доповнюються потім одним з активних методів для реалізації зворотного зв'язку між інтерпретацією інженера по знаннях і поданнями експерта.

4.1.2.1.3. Лекції

Лекція є самим старим способом передачі знань. Лекторське мистецтво здавна дуже високо цінувалося у всіх галузях науки й культури.

Але нас зараз цікавить не стільки здатність до підготовки й читання лекцій, скільки здатність цю лекцію слухати, конспектувати й засвоювати. Уже говорилося, що найчастіше експертів не вибирають, і тому вчити експерта читанню лекції інженер по знаннях не зможе. Але якщо експерт має досвід викладача (наприклад, професор клініки або досвідчений керівник виробництва), те можна скористатися таким концентрованим фрагментом знань, як лекція.

У лекції експертові надано багато ступенів волі для самовираження; при цьому необхідно сформулювати експертові тему й задачу лекції. Наприклад, тема циклу лекцій "Постановка діагнозу - запалення легенів", тема конкретної лекції "Міркування по аналізі рентгенограм", задача - навчити слухачів по перерахованим експертом ознакам ставити діагноз запалення легенів і робити прогноз. При такій постановці досвідчений лектор може заздалегідь структурувати свої знання й хід міркувань. Від інженера по знаннях у цій ситуації потрібно лише грамотно законспектувати лекцію й наприкінці задати необхідні питання.

Студенти добре знають, що конспекти лекцій того самого лектора в різних студентів істотно відрізняються. Списати конспект лекцій просять, як правило, в одного-двох студентів із групи. Люди, уміло провідний конспект, звичайно сильні студенти. Зворотне не вірно. У чому ж полягає мистецтво ведення конспекту? В "завадостійкості". Записувати головне, опускаючи другорядне, виділяти фрагменти знань (параграфи, підпараграфи), записувати тільки осмислені пропозиції, уміти узагальнювати.

Гарне питання по ходу лекції допомагає й лекторові, і слухачеві. Серйозні й глибокі питання можуть істотно підняти авторитет інженера по знаннях в очах експерта.

Досвідчений лектор знає, що всі питання можна

умовно розбити на три групи:

- розумні питання, що поглиблюють лекцію;
- дурні питання або питання не власне кажучи;
- питання "на засипання", або провокаційні.

Якщо інженер по знаннях задає питання другого типу, то можливі дві реакції. Увічливий експерт буде розмовляти з таким аналітиком як з дитиною, що зараз не розуміє й однаково нічого вже не зрозуміє. Зарозумілий експерт просто вийде з контакту, не бажаючи гаяти час. Якщо ж інженер по знаннях захоче продемонструвати свою ерудицію питаннями третього типу, то нічого, крім роздратування й відчуження, він, очевидно, у відповідь не одержить.

Тривалість лекції рекомендується стандартна - від 40 до 50 хвилин і через 5-10 хвилин - ще стільки ж. Курс звичайно від двох до п'яти лекцій.

Метод добування знань у формі лекцій, як і всі пасивні методи, використовують на початку розробки як ефективний спосіб швидкого занурення інженера по знаннях у предметну область.

4.1.2.2. Активні індивідуальні методи

Активні індивідуальні методи добування знань на сьогоднішній день - найпоширеніші. У тім або іншому ступені до них прибігають при розробці практично будь-який ЕС.

До основних активних методів можна віднести:

- анкетування;
- інтерв'ю;
- ігри з експертом.

У всіх цих методах активну функцію виконує інженер по знаннях, що пише сценарій і режисує сеанси добування знань. Гри з експертом істотно відрізняються від інших питальних методів.

4.1.2.2.1. Анкетування

Анкетування — самий твердий метод, тобто найбільш стандартизований. У цьому випадку інженер по знаннях заздалегідь становить запитальник або анкету, розмножує її й використає для опитування декількох експертів. Це основна перевага анкетування.

Сама процедура може проводитися двома способами:

- аналітик уголос задає питання й сам заповнює анкету по відповідях експерта;
- експерт самостійно заповнює анкету після попереднього інструктування.

Вибір способу залежить від конкретних умов (наприклад, від оформлення анкети, її зрозумілості, готовності експерта). Другий спосіб нам здається переважніше, тому що в експерта з'являється необмежений час на обмірковування відповідей.

Основними факторами, на які можна істотно вплинути при анкетуванні, є засоби спілкування (у цьому випадку це запитальник) і ситуація спілкування.

Запитальник (анкета) заслуговує особливої розмови. Існує кілька загальних рекомендацій при складанні анкет. Ці рекомендації є універсальними, тобто не залежать від предметної області.

Найбільший досвід роботи з анкетами накопичений у соціології й психології.

- Анкета не повинна бути монотонною й одноманітною, тобто викликати нудьгу або утому. Це досягається варіаціями форми питань, зміною тематики, вставкою питань-жартів й ігрових питань.
- Анкета повинна бути пристосована до мови експертів.
- Варто враховувати, що питання впливають один на одного, і тому послідовність питань повинна бути строго продумана.
- Бажано прагнути до оптимальної надмірності. Відомо, що в анкеті завжди багато зайвих питань, частина з них необхідна — це так називані контрольні питання (*див. про них нижче*), а інша частина повинна бути мінімізована.

Зайві питання з'являються, наприклад, у таких ситуаціях.

Фрагмент анкети: "В12. Чи вважаєте Ви, що для лікування ангіни ефективний еритромицин? В13. Які дози еритромицину Ви звичайно рекомендуєте?"

При негативній відповіді на 12-й питання 13-й є зайвим. Його можна уникнути, ускладнивши питання.

"В12. Чи застосовуєте Ви еритромицин для лікування ангіни і якщо так, то в яких дозах?"

- Анкета повинна мати "гарні манери", тобто її мова повинна бути ясною, зрозумілою, гранично ввічливою.

Методичною майстерністю складання анкети можна опанувати тільки на практиці.

4.1.2.2.2. Інтерв'ю

Під інтерв'ю будемо розуміти специфічну форму спілкування інженера по знаннях й експерта, у якій інженер по знаннях задає експертові серію заздалегідь підготовлених питань із метою добування знань про предметну область. Найбільший досвід у проведенні інтерв'ю накопичений, напевно, у журналістиці й соціології. Більшість фахівців цих областей відзначають, проте, крайню недостатність теоретичних і методичних досліджень по тематиці інтерв'ювання.

Інтерв'ю дуже близько тому способу анкетування, коли аналітик сам заповнює анкету, заносючи туди відповіді експерта. Основна відмінність інтерв'ю в тім, що воно дозволяє аналітику опускати ряд питань залежно від ситуації, вставляти нові питання в анкету, змінювати темп, різноманітнити ситуацію спілкування. Крім цього, в аналітика з'являється можливість "взяти в полон" експерта своєю чарівністю, зацікавити його самою процедурою й, тим самим, збільшити ефективність сеансу добування.

Питання для інтерв'ю. Тепер трохи докладніше про центральну ланку активних індивідуальних методів - про питання. Інженери по знаннях рідко сумніваються у своїй здатності задавати питання. У той час як й у філософії, так у математику ця проблема обговорюється з давнього років. Існує навіть спеціальна галузь математичної логіки - еротетична логіка (логіка питань). Є цікава робота Белнапа й Стила "Логіка питань і відповідей", але, на жаль, використати результати, отримані логіками, безпосередньо при розробці інтелектуальних систем не вдається.

Всі питальні речення можна розбити на два типи:

- питання з невизначеністю, що ставиться до всієї пропозиції ("Дійсно, введення більших доз антибіотиків може викликати анафілактичний шок?");
- питання з неповною інформацією ("При яких умовах необхідно включати кнопку?"), що часто починаються зі слів "хто", "що", "де", "коли" і т.д.

Відкрите питання називає тему або предмет, залишаючи повну волю експертові за формою й змістом відповіді ("Не могли б Ви розповісти, як краще збити високу температуру у хворого із запаленням легенів?").

У закритому питанні експерт вибирає відповідь із набору запропонованих ("Укажіть, будь ласка, що Ви рекомендуєте при ангіні: а) антибіотики, б) полоскання, в) компреси, г) інгаляції"). Закриті питання легше обробляти при наступному аналізі, але вони більше небезпечні, тому що "закривають" хід міркувань експерта й "програмують" його відповідь у певному напрямку. При складанні сценарію інтерв'ю корисно чергувати відкриті й закриті питання, особливо ретельно продумувати закриті, оскільки для їхнього складання потрібна певна ерудиція в предметній області.

Особисте запитання стосується безпосередньо індивідуального досвіду експерта ("Скажіть, будь ласка, Іван Данилович, у Вашій практиці Ви застосовуєте вулнузан при фурункульозах?"). Особисті питання звичайно активізують мислення експерта, "грають" на його самолюбстві, вони завжди

прикрашають інтерв'ю.

Безособове питання спрямоване на виявлення найпоширеніших і загальноприйнятих закономірностей предметної області ("Що впливає на швидкість процесу ферментації лізіна?").

При складанні питань варто враховувати, що мовні здатності експерта, як правило, обмежені й внаслідок скутості, замкнутості, боязкості він не може відразу висловити свою думку й надати знання, які від нього потрібні (навіть якщо припустити, що він їх чітко для себе формулює). Тому часто при "закатості" експерта використовують не прямі запитання, які безпосередньо вказують на предмет або тему ("Як Ви ставитеся до методики доктора Сухарева?"), а непрямі, які лише побічно вказують на предмет, що цікавить ("Чи застосовуєте Ви методику доктора Сухарева? Опишіть, будь ласка, результати лікування"). Іноді доводиться задавати кілька десятків непрямих запитань замість одного прямого.

Вербальні питання — це традиційні усні питання. Питання з використанням наочного матеріалу вносять розмаїтість в інтерв'ю й знижують стомлюваність експерта. У цих питаннях використовують фотографії, малюнки й картки. Наприклад, експертові пропонуються кольорові картонні картки, на яких виписані ознаки захворювання. Потім аналітик просить розкласти ці картки в порядку убавання значимості для постановки діагнозу.

Ділення питань по функції на основні, зондувальні, контрольні пов'язане з тим, що часто основні питання інтерв'ю, спрямовані на виявлення знань, не спрацьовують - експерт із якихось причин іде убік від питання, відповідає нечітко.

Тоді аналітик використовує зондувальні питання, які направляють міркування експерта в потрібну сторону. Наприклад, якщо не спрацював основне питання: "Які параметри визначають момент закінчення процесу ферментації лізіна?", аналітик починає задавати зондувальні питання:

"Чи завжди процес ферментації триває 72 години? А якщо він закінчується раніше, як це довідатися? Якщо він протриває більше, те що змусить мікробіолога не закінчити процес на 72-м години?" і т.д.

Контрольні питання застосовують для перевірки вірогідності й об'єктивності інформації, отриманої в інтерв'ю раніше ("Скажіть, будь ласка, а московська школа психологів так само як Ви трактує шкалу до опросника ММРІ?" або "Чи рекомендуєте Ви ін'єкції АТФ?" (АТФ - препарат, знятий з виробництва). Контрольні питання повинні бути "хитро" складені, щоб не скривдити експерта недовірою (для цього використовують повторення питань в іншій формі, уточнення, посилення на інші джерела). "Краще два рази

запитати, чим один раз наплутати" (Шолом Алейхем).

І нарешті, про нейтральні й навідні запитання. У принципі інтерв'юєрові (у нашому випадку інженерові по знаннях) рекомендують бути безстороннім, звідси й питання його повинні носити *нейтральний характер*, тобто не повинні вказувати на відношення інтерв'юєра до даної теми. Навпроти, навідні запитання змушують респондента (у цьому випадку експерта) прислухатися або навіть взяти до уваги позицію інтерв'юєра. Нейтральне питання: "чи збігаються симптоми крововиливу в мозок і струси мозку?" Навідне запитання: "Не чи правда, дуже важко диференціювати симптоми крововиливу в мозок?"

Крім перерахованих вище, корисно розрізнити й включати в інтерв'ю наступні питання :

- контактні ("ломающие лід" між аналітиком й експертом);
- буферні (для розмежування окремих тим інтерв'ю);
- оживляючу пам'ять експертів (для реконструкції окремих випадків із практики);
- "провокуючі" (для одержання спонтанних, непередбачених відповідей).

Три основні характеристики питань, які впливають на якість інтерв'ю:

- мова питання (зрозумілість, лаконічність, термінологія);
- порядок питань (логічна послідовність і немонотонність);
- доречність питань (етика, увічливість).

Питання в інтерв'ю - це не простий засіб спілкування, але й спосіб передачі думок і позиції аналітика.

"Питання являє собою форму руху думки, у ньому яскраво виражений момент переходу від незнання до знання, від неповного, неточного знання до більше повного й більше точного". Звідси необхідність у протоколах фіксувати не тільки відповіді, але й питання, попередньо ретельно відпрацьовуючи їхню форму й зміст.

Очевидно, що будь-яке питання має сенс тільки в контексті. Тому питання може готувати інженер по знаннях, що вже опанував ключовим набором знань.

Питання мають для експерта діагностичне значення - кілька відвертих "дурних" питань можуть повністю розчарувати експерта й відбити в нього полюбання до подальшого співробітництва. Відома відповідь Маркса на питання Прудона: "Питання було настільки неправильно поставлене, що на нього неможливо було дати правильну відповідь".

У досвідченого аналітика інтерв'ю зовні може бути схоже на вільний діалог - це метод добування

знань у формі бесіди інженера по знаннях й експерта, у якій немає твердого регламентованого плану й запитальника. Це визначення не означає, що до вільного діалогу не треба готуватися. Навпроти, зовні вільна й легка форма цього методу вимагає найвищої професійної й психологічної підготовки.

Кваліфікована підготовка до діалогу допомагає аналітикові стати щирим драматургом або сценаристом майбутніх сеансів, тобто запланувати гладкий плин процедури добування: від приємного враження на початку бесіди перехід до професійного контакту через пробудження інтересу й завоювання довіри експерта. При цьому для забезпечення бажання експерта продовжити бесіду необхідно проводити "поглаживання", тобто підбадьорювати експерта й підтверджувати всіляко його впевненість у власній компетентності (вставки-фрази: "Я Вас розумію...", "...це дуже цікаво" і т.д.)

Так, в одному з досліджень до техніки ведення професійних журналістських діалогів було експериментально доведено, що схвальне й заохочувальне "хмыканье" інтерв'юєра збільшує довжину відповідей респондента. При цьому схвалення повинне бути щирим, як показав опитування інтерв'юєрів Інституту демоскопії Німеччини: "Кращий виверт - це уникати всяких вивертів: ставитися до опитуваного із щирим людинолюбством, не з награвим, а зі справжнім інтересом". Щоб розговорити співрозмовника, можна спочатку аналітикові розповісти про себе, про роботу, тобто поговорити самому.

У вільному діалозі важливо також вибрати правильний темп або ритм бесіди: без більших пауз, тому що експерт може відволіктися, але й без гонки, інакше швидко стомлюються обидва учасники, і наростає напруженість, крім того, деякі люди говорять і думають дуже повільно. Уміння чергувати різні темпи, напругу й розрядку в бесіді істотно впливає на результат.

Підготовка до діалогу так само, як і до інших активних методів добування знань, включає складання плану проведення сеансу добування, у якому необхідно передбачити наступні стадії:

- Початок бесіди (знайомство, створення в експерта "образа" аналітика, пояснення цілей і задач роботи).
- Діалог по добуванню знань.
- Заключна стадія (подяка експертові, підведення підсумків, договір про наступні зустрічі).

Девізом для інженера по знаннях можуть послужити погляди одного із класиків вітчизняного літературознавства М. М. Бахтина.

Діалог припускає:

- унікальність кожного партнера і їхня

- принципова рівність один одному;
- розходження й оригінальність їхніх точок зору;
- орієнтацію кожного на розуміння й на активну інтерпретацію його точки зору партнером;
- очікування відповіді і його передбачення у власному висловленні;
- взаємну додатковість позицій учасників спілкування, співвіднесення яких й є метою діалогу".

4.1.2.3. Активні групові методи

До групових методів добування знань ставляться:

- рольові ігри;
- дискусії за "круглим столом" за участю декількох експертів;
- "мозкові штурми".

Основне достоїнство групових методів - це можливість одночасного "поглинання" знань від декількох експертів, взаємодія яких вносить у цей процес елемент принципової новизни від накладення різних поглядів і позицій.

Активні групові методи звичайно використовуються як гостра приправа при добуванні знань, самі по собі вони не можуть служити джерелом більш-менш повного знання. Їх застосовують як додаткові до традиційних індивідуальних методів (спостереження, інтерв'ю й т.д.), для активізації мислення й поведінки експертів.

Оскільки ці методи менш популярні, чим індивідуальні (що зв'язано зі складністю організації), опишемо їх докладно.

4.1.2.3.1. "Круглий стіл"

Метод круглого стола (термін запозичений з журналістики) передбачає обговорення якої-небудь проблеми з обраної предметної області, у якому беруть участь із рівними правами кілька експертів. Звичайно спочатку учасники висловлюються в певному порядку, а потім переходять до живої вільної дискусії. Число учасників дискусії коливається від трьох до п'яти - семи.

Більшість загальних рекомендацій з добування знань, запропонованих раніше, застосовно й до даного методу. Однак існує й специфіка, пов'язана з поведінкою людини в групі.

По-перше, від інженера по знаннях підготовка "круглого стола" зажадає додаткових зусиль: як організаційних (місце, час, обстановка, мінеральна вода, чай, кворум і т.д.), так і психологічних (уміння вставляти доречні репліки, почуття гумору, пам'ять на імена та по батькові, здатність гасити конфліктні ситуації й т.д.).

По-друге, більшість учасників буде говорити під впливом "ефекту фасаду" зовсім не те, що вони сказали б в іншій обстановці, тобто бажання зробити враження на інших експертів буде істотно "подсвечивать" їхнього висловлення. Цей ефект часто спостерігається на захистах дисертацій. Члени вченої ради запитують звичайно не те, що їм дійсно цікаво, а те, що демонструє їхню власну компетентність.

Хід бесіди за круглим столом зручно записувати на магнітофон, а при розшифровці й аналізі результатів урахувати цей ефект, а також взаємні відносини учасників.

Задача дискусії - колективно, з різних точок зору, під різними кутами досліджувати спірні гіпотези предметної області. Звичайно емпіричні області багаті таким дискусійним матеріалом. Для гостроти на "круглий стіл" запрошують представників різних наукових напрямків і різних поколінь, це також зменшує небезпека одержання однобічних однобоких знань.

Обмін думками по наукових питаннях має давню традицію в історії людства (антична Греція, Індія). До наших днів дійшли літературні пам'ятники обговорення спірних питань (наприклад, Протагор "Мистецтво сперечатися", роботи софістів), що послужили першоосновою діалектики — науки вести бесіду, сперечатися, розвивати теорію. У самому слові дискусія (від лат. *discussio* — дослідження) утримується вказівка на те, що це метод наукового пізнання, а не просто суперечки (для порівняння, полеміка — від греч. *polemikos* — войовничий, ворожий).

Кілька практичних рад по процедурних питаннях "круглого стола". Перед початком дискусії ведучому корисно:

- переконатися, що все правильно розуміють задачу (тобто відбувається сеанс добування знань);
- установити регламент;
- чітко сформулювати тему.

По ходу дискусії важливо простежити, щоб занадто емоційні й говірки експерти не підмінили тему, і щоб критика позицій один одного була обґрунтованою.

Наукова плідність дискусій робить цей метод привабливим і для самих експертів, особливо для тих, хто знає менше. Це помітив ще Епікур: "При філософській дискусії більше виграє переможений - у тім відношенні, що він множить знання".

4.1.2.3.2. "Мозковий штурм"

"Мозковий штурм" або "мозкова атака" - один з найпоширеніших методів розкріпачення й активізації творчого мислення. Інші методи (метод

фокальних об'єктів, синектика, метод контрольних питань) застосовуються набагато рідше через меншу ефективність.

Уперше цей метод був використаний в 1939 р. у США А. Осборном як спосіб одержання нових ідей в умовах заборони критики. Замічено, що острах критики заважає творчому мисленню, тому основна ідея штурму - це відділення процедури генерування ідей у замкнутій групі фахівців від процесу аналізу й оцінки висловлених ідей.

Як правило, штурм триває недовго (близько 40 хвилин). Учасникам (до 10 чоловік) пропонується висловлювати будь-які ідеї (жартівливі, фантастичні, помилкові) на задану тему (критика заборонена). Звичайно висловлюється більше 50 ідей. Регламент до 2 хвилин на виступ. Самий цікавий момент штурму - це настання піка (ажіотажу), коли ідеї починають "фонтанувати", тобто відбувається мимовільна генерація гіпотез учасниками. Цей пік має теоретичне обґрунтування в роботах видатного швейцарського психолога й психіатра З. Фрейда про несвідомий. При наступному аналізі всього лише 10-15% ідей виявляються розумними, але серед них бувають досить оригінальні. Оцінює результати звичайно група експертів, що не брала участь у генерації.

Ведучий "мозкового штурму" - інженер по знаннях - повинен вільно володіти аудиторією, підібрати активну групу експертів - "генераторів", не затискати погані ідеї - вони можуть служити каталізаторами гарних. Мистецтво ведучого - це мистецтво задавати питання аудиторії, "підігриваючи" генерацію. Питання служать "гачком", яким витягаються ідеї. Питання також можуть зупиняти багатослівних експертів і служити способом розвитку ідей інших.

Основний девіз штурму - "чим більше ідей, тим краще". Фіксація ходу сеансу - традиційна (протокол або магнітофон).

4.1.2.3.3. Експертні ігри

Грою називають такий вид людської діяльності, що відбиває (відтворює) інші її види. При цьому для гри характерні одночасно умовність і серйозність.

Поняття експертної гри або гри з експертами з метою добування знань сходиться до трьох джерел - це

- ділові ігри, широко використовувані при підготовці фахівців і моделюванні;
- діагностичні ігри;
- і комп'ютерні ігри, всі частіше застосовувані в навчанні.

У цей час у психолого-педагогічних науках немає розвинутої теоретичної концепції ділових ігор й інших ігрових методів навчання. Проте на

практиці ці ігри широко використовуються. Під *діловою* грою найчастіше розуміють експеримент, де учасникам пропонується виробнича ситуація, а вони на основі свого життєвого досвіду, своїх загальних і спеціальних знань і подань приймають рішення. Рішення аналізуються, і розкриваються закономірності мислення учасників експерименту. Саме ця частина, що аналізує, ділової гри корисна для одержання знань. І якщо учасниками такої гри стають експерти, то гра з ділової перетворюється в експертну. Із трьох основних типів ділових ігор (навчальних, планово-виробничих і дослідницьких) до експертів ближче всього дослідницькі, які використовуються для аналізу систем, перевірки правил прийняття рішень.

Діагностична гра - ця та ж ділова гра, але застосовувана конкретно для діагностики методів ухвалення рішення в медицині (діагностика методів діагностики). Ці ігри виникли при дослідженні способів передачі знань від досвідчених лікарів новачкам. У нашому розумінні діагностична гра - це гра, безумовно, експертна без усяких застережень, тільки з жорстко закріпленою предметною областю - медициною.

Плодотворність моделювання реальних ситуацій в іграх підтверджується сьогодні практично у всіх галузях науки й техніки. Вони розвивають логічне мислення, уміння швидко приймати рішення, викликають інтерес в експертів.

Відповідно до класифікації введеної раніше будемо розділяти експертні ігри на :

- індивідуальні ;
- групові.

Крім того, продовжимо й розвинемо цю класифікацію введенням інших критеріїв:

- використання спеціального устаткування;
- застосування обчислювальної техніки.

4.1.2.3.4. Ігри з експертом

У цьому випадку з експертом грає інженер по знаннях, що бере на себе яку-небудь роль у модельованій ситуації. Наприклад, одному з авторів часто доводилося грати з експертом гру "Учитель й учень", у якій інженер по знаннях бере на себе роль учня й на очах в експерта виконує його роботу (наприклад, пише психодіагностическое висновок), а експерт поправляє помилки "учня". Ця гра - зручний спосіб розговорити соромливого експерта.

Інша гра змушує інженера по знаннях взяти на себе роль лікаря, що знає добре хворого, а експерт відіграє роль консультанта. Консультант задає питання й робить прогноз про доцільність застосування того або іншого виду лікування (в описаній грі це був прогноз доцільності електростимульної терапії при серцевій аритмії). Гра

"двох лікарів" дозволила виявити, що експертові знадобилося всього 30 питань для успішного прогнозу, у той час як первісний варіант запитальника, складений медиками для тієї ж мети, містив 170 питань.

Ще один приклад гри. Спочатку експерта просять написати обґрунтування для власного прогнозу. Наприклад, чому він вважає, що виразка у хворого *X* гоїться. Накопичується кілька таких обґрунтувань, а через якийсь час експертові зараховують тільки його обґрунтування й просять зробити прогноз. Як правило, цього він зробити не може, тобто обґрунтування (або його знання) було неповним. Експерт доповнює обґрунтування, тим самим виявляються сховані (для самого експерта) шари знань. Так, в іграх "Обґрунтування прогнозу рецидиву виразкової кровотечі" удалося виявити, що значимими для прогнозу є всього три правила. Причому два правила входили в традиційно-діагностичний запитальник, а третє було сформульовано під час гри.

Гра "Фокусування на контексті": експерт відіграє роль ЕС, а інженер по знаннях - роль користувача. Розігрується ситуація консультації. Перші питання експерта виявляють найбільш значимі поняття, найважливіші аспекти проблеми. Роль користувача може взяти на себе й іншого експерта.

Основні ради інженерові по знаннях по проведенню індивідуальних ігор:

- грайте сміло, весело, нешаблонно, придумуйте гри самі;
- не нав'язуйте гру експертові, якщо він не розташований;
- у грі "не дайте" на експерта, не забувайте мети гри;
- не забувайте про час і про те, що гра стомлююча для експерта.

4.1.2.3.5. Рольові ігри в групі

Групові ігри передбачають участь у грі декількох експертів. До такої гри звичайно заздалегідь складається сценарій, розподіляються ролі, до кожної ролі готується портрет-опис (краще з девізом) і розробляється система оцінювання гравців.

Існує кілька способів проведення рольових ігор. В одних іграх гравці можуть придумати собі нові імена й грати під ними. В інші всі гравці переходять на "ти". У третій ролі вибирають гравці, у четвертій ролі витягають по жеребі. Роль - це комплекс зразків поведінки. Роль пов'язана з іншими ролями. "Короля грає звита". Оскільки в нашому випадку режисером і сценаристом гри є інженер по знаннях, то йому й надається повна воля у виборі форми проведення гри.

Так, описана гра "План", призначена для добування знань із фахівців підприємства, що розробляють виробничі плани випуску для цехів і приймаючих різних рішень по керуванню виробництвом.

У грі експертів розбили на три ігрові групи: ОПР₁ — група планування; ОПР₂ — група менеджерів; Э — група експертизи по оцінюванню дії ОПР₁ й ОПР₂. Групам ОПР₁ й ОПР₂ задавалися різні виробничі ситуації й ретельно протоколювалися їхні суперечки, міркування, аргументи по прийнятті рішень. У результаті гри був створений прототип бази знань експертної системи планування.

Звичайно в грі бере участь від трьох до шести експертів, якщо їх більше, те можна розбити всіх на кілька конкуруючих ігрових бригад. Елемент змагальності пожвавлює гру. Наприклад, чий діагноз виявиться ближче до щирого, чий план раціональніше використає ресурси, хто швидше визначить причину несправності в технічному блоці.

Створення ігрової обстановки зажадає чимало фантазії й творчу вигадку від інженера по знаннях. Рольова гра, як правило, вимагає деяких найпростіших заготівель (наприклад, табличок "Директор", "Бухгалтерія", "Плановий відділ", спеціально надрукованих інструкцією із правилами гри). Але головне, звичайно, щоб експерти в грі дійсно "заграли", розкріпачилися й "розкрили свої карти".

4.1.2.3.6. Ігри із тренажерами

Гри із тренажерами в значній мірі ближче не до ігор, а до імітаційних вправ у ситуації, наближеної до дійсності.

Наявність тренажера дозволяє відтворити майже виробничу ситуацію й спостерігати за експертом. Тренажери широко застосовують для навчання (наприклад, льотчиків або операторів атомних станцій). Очевидно, що застосування тренажерів для добування знань дозволить зафіксувати фрагменти "летучих" знань, що виникають під час і на місці реальних ситуацій і випадаючі з пам'яті при виході за межі ситуації.

4.1.2.3.7. Комп'ютерні експертні ігри

Ідея використати комп'ютери в ділових іграх відома давно. Але тільки коли комп'ютерні ігри взяли в полон практично всіх користувачів персональних ЕОМ від мала до велика, стала очевидною особлива привабливість такого роду ігор.

Приведемо традиційну сучасну класифікацію комп'ютерних ігор з журналу GAME.EXE.

- Action/Arcade games (екшн/аркади). Гри-дії.

Вимагають гарного окоміру й швидкої реакції.

- Simulation games (симулятори). Базуються на моделюванні реальної дійсності й відпрацьовування практичних навичок, наприклад у водінні автомобіля, пасажирського літака, поїзда, авіадиспетчера й навіть симулятори рибного лову. Також популярні спортивні симулятори - теніс, бокс й ін.
- 3D Action games ("стрелялки"). Те ж, що й екшн, але з активним використанням тривимірної графіки.
- Strategy games (стратегічні ігри). Вимагають стратегічного планування й відповідальності при прийнятті рішень, наприклад розвиток цивілізацій, суперництво мирів, економічна боротьба. Особливий клас стратегічних ігор - wargames (військові гри). Останнім часом упор в 3D Action робиться на багатокористувацький режим (гру по мережі).
- Puzzles (настільні ігри-головоломки). Комп'ютерні реалізації різних логічних ігор.
- Adventure/Quest (пригодницькі ігри). Звичайно мають розгалужений сценарій, гарною графікою й звуком. Управляючи одним або декількома персонажами, гравець повинен правильно вести діалоги, розгадувати множину загадок і головоломок, зауважувати й правильно використати предмети, заховані в грі.
- Role-playing games RPG (рольові ігри). Розповсюджений жанр, що бере свій початок у старих англійських настільних іграх. Існують один або кілька персонажів, що володіють індивідуальними здатностями й характеристиками. Їм доводиться боротися з ворогами, вирішувати загадки. У міру виконання цих задач, у героїв накопичується досвід, і по

досягненню певного значення, їхні характеристики поліпшуються...

Слід зазначити, що багато ігор можуть бути віднесені відразу до декількох класів, і в цілому цю класифікацію не можна вважати строгою. Ігри іноді корисні для розваги експертів перед сеансом добування знань. Крім того, мабуть, що експертні ігри, сполучаючи елементи перерахованих вище класів, можуть успішно застосовуватися для безпосереднього добування знань. Однак розробка й програмна реалізація такої гри зажадають істотних вкладень тимчасових і грошових ресурсів.

Основний принцип гри "Зоосад" складається в створенні ігрової ситуації при організації діалогу з експертом. При цьому задача добування знань маскується націленістю на рішення чисто ігрової задачі: необхідно визначити вміст "чорного ящика", у якому перебуває якась тварина, при цьому треба набрати найбільшу кількість окулярів, не витративши виділеного ресурсу грошей. У ході гри експерт робить ставки на різні гіпотези, указуючи при цьому, якими ознаками володіє та або інша тварина. По ходу гри невидимо для експерта формуються правила, що відбивають знання експерта на підставі зроблених їм ходів. У даній грі - це знання про те, якими ознаками володіють ті або інші тварини. Таким чином, виявляється алфавіт значимих ознак для діагностики й класифікації тварин.

Тема 5. Філософія CLIPS - C Language Integrated Production System

5.1. Введення до CLIPS

Спочатку аббревіатура **CLIPS** була назвою мови — **C Language Integrated Production System** (тобто мова С, інтегрована із продукційними системами), зручної для розробки баз знань і макетів експертних систем.

Тепер CLIPS являє собою сучасний інструмент, призначений для створення експертних систем (expert system tool). CLIPS складається з інтерактивного середовища — експертної оболонки зі своїм способом подання знань, гнучкої й потужної мови й декількох допоміжних інструментів. Зараз, завдяки добрій волі своїх творців, CLIPS є абсолютно вільно розповсюджуваним програмним продуктом.

Офіційний сайт CLIPS розташовується за адресою:

<http://www.ghg.net/clips/CLIPS.html>

Цей сайт допоможе вам одержати як сам CLIPS, так і всілякий матеріал для його вивчення й освоєння (документацію, приклади, поради фахівців, вихідні коди й багато чого іншого).

Зараз на ринку доступно не так багато експертних оболонок (інструментів, призначених для створення експертних систем). Незважаючи на те, що CLIPS поширюється безкоштовно, він досить успішно конкурує навіть із найвідомішими комерційними проектами. Кількість користувачів CLIPS росте рік у рік. Про це можна судити по активності відвідування сайтів, форумів і конференцій, присвячених CLIPS. Якщо ви ще не встановили CLIPS на свій комп'ютер - можливо, найкращий час зробити це.

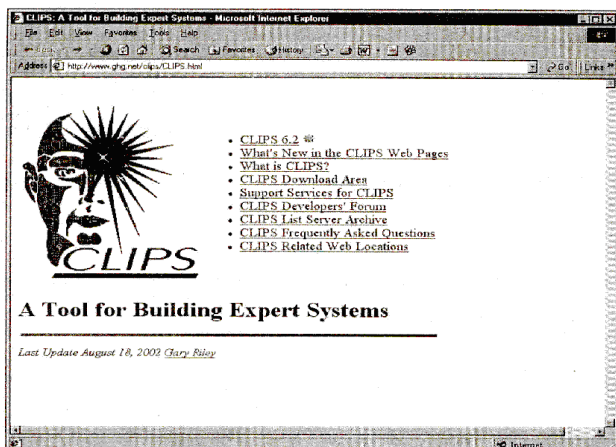


Рис. 5. 1. Домашня сторінка CLIPS

5.2. Історія створення CLIPS

Поява мови CLIPS можна датувати 1984 р., місце народження CLIPS — космічний центр Джонсона NASA. Саме в цей час відділ штучного інтелекту (тепер Software Technology Branch) розробив множину прототипів експертних систем, що використовують сучасне програмне й технічне забезпечення. Однак, незважаючи на великий потенціал експертних систем, не багато хто із цих додатків дійшли до кінцевого споживача. Ця невдача обумовлювалася технологією створення експертних систем, який у той час оперували в NASA. Основні обмеження накладав мову LISP, використовувана як базова мова для розробки експертних систем.

Як головні недоліки мови LISP можна виділити наступні три:

- 1) недостатня адаптованість LISP до широкого кола стандартних комп'ютерів;
- 2) висока ціна технічного й програмного забезпечення, призначеного для роботи з LISP;
- 3) низька здатність інтеграції систем, написаних на LISP, із системами, написаними на інших мовах (виробництво вкладених додатків).

Співробітники відділу штучного інтелекту зрозуміли, що використання традиційних мов програмування, таких як С, усуне більшість виниклих проблем, і відділ почав пошуки виробників і постачальників інструментів для створення експертних систем, що оперують одним із традиційних мов програмування. Незважаючи на те, що число подібних інструментів було досить велике, ціна таких інструментів виявилася досить висока. Крім того, більшість із цих інструментів працювали на дуже невеликому числі платформ, а швидкість їхньої роботи залишала бажати кращого. Стало очевидно, що для одержання інструмента, що задовольняє всім вимогам NASA, необхідна розробка власного засобу для створення експертних систем.

Прототип CLIPS був розроблений навесні 1985 р., деяким більш ніж за два місяці. Особлива увага була приділена створенню мови, сумісного з мовами, що використовуються в NASA у той момент. Таким чином, синтаксис мови CLIPS був зроблений дуже схожим на синтаксис експертної оболонки ART, розробленою корпорацією Inference. Незважаючи на те, що ART послужив прообразом, CLIPS розроблявся зовсім без допомоги Inference або доступу до вихідних кодів системи ART.

Завдяки тому, що CLIPS є вільно розповсюджуваним програмним продуктом з доступними вихідними кодами, останнім часом була випущена множина програм і бібліотек, що вдосконалюють і доповнюють можливості CLIPS. Деякі із цих продуктів є власністю їхніх компаній, що випустили, і призначені для внутрішнього використання або комерційного поширення, інші, як і сам CLIPS, поширюються вільно.

Як самі відомі приклади подібних проектів можна привести DLL/OCX-бібліотеку, що дозволяє використати механізм логічного висновку CLIPS у ваших додатках, FuzzyCLIPS, CLIPS++, CLIPS code generator. Більшість перерахованих продуктів, а також різна корисна інформація доступна за адресою:

<http://ourworld.compuserve.com/homepages/marktoml/clipstuf.htm> (рис. 5.2).

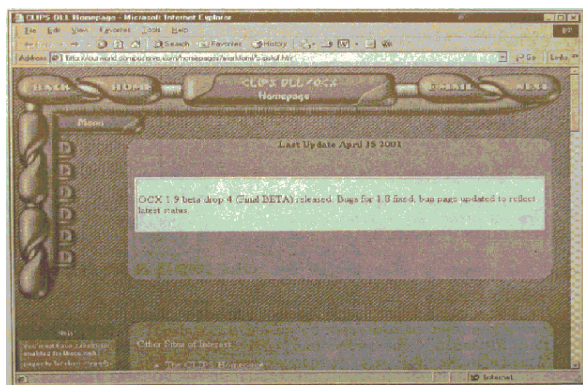


Рис. 5.2. Домашня сторінка CLIPS DLL/OCX

5.3. Робота з CLIPS

Для демонстрації прикладів, використовуваних у цій книзі, буде застосовуватися Windows-версія CLIPS 6.2. Незважаючи на повну сумісність із Apple Macintosh й UNIX-версіями, при роботі з даною книгою бажано використати саме Windows-версію середовища CLIPS. Зовнішній вигляд головного вікна CLIPS показаний на мал. 6.3.

Windows-версія середовища CLIPS повністю сумісна з базовою специфікацією мови. Уведення команд здійснюється безпосередньо в головне вікно CLIPS. Однак у порівнянні з базовою Windows-версією надає множину додаткових візуальних інструментів (наприклад, менеджери фактів або правил), значно полегшуюче життя розробника експертних систем.

Експертні системи, створені за допомогою CLIPS, можуть бути запущені трьома основними способами:

- введенням відповідних команд і конструкторів мови безпосередньо в середовище CLIPS;
- використанням інтерактивного віконного

інтерфейсу CLIPS (наприклад для версій Windows або Macintosh);

- за допомогою програм-оболонок, що реалізують свій інтерфейс спілкування з користувачем і механізмів, що використовують, знань і логічного висновку CLIPS.

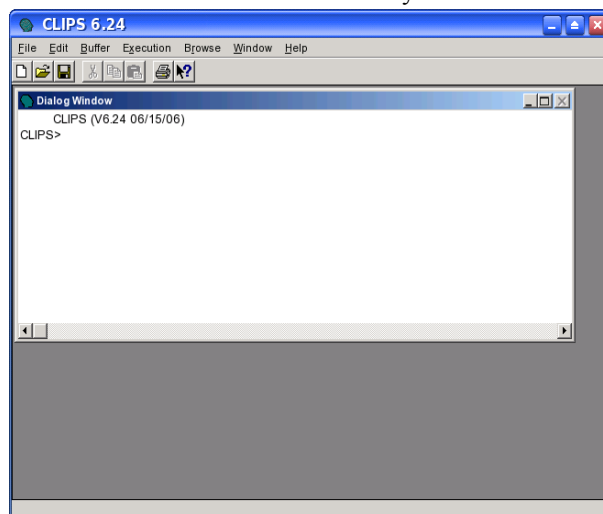


Рис. 5.3. Головне вікно CLIPS

Крім того, CLIPS при запуску дозволяє виконувати командні файли власного формату (ця можливість також доступна за допомогою команди batch). Для реалізації цієї можливості необхідно запустити CLIPS (у нашому випадку це файл CLIPSWin.exe) з одним із трьох наступних аргументів:

- -f <імені-файлу>
- -f2 <імена-файли>
- -l <імені-файлу>

Текстовий файл, заданий за допомогою опції -f, повинен містити команди CLIPS.

Якщо заданий файл містить команду exit, то CLIPS завершить свою роботу й користувач повернеться в операційну систему. У випадку якщо команда exit відсутня, то після виконання всіх команд із заданого файлу користувач потрапить у головне вікно CLIPS.

Команди в текстовому файлі повинні бути набрані так само, як якби вони вводилися безпосередньо в командний рядок CLIPS (тобто всі команди повинні бути укладені в круглі дужки й розділені символом переходу на новий рядок).

Опція -f фактично еквівалентна запуску команди batch відразу після запуску CLIPS.

Опція -f2 ідентична -f, але, на відміну від опції -f, вона використовує команду batch*. Файл, заданий цією опцією, також виконується після запуску CLIPS, але результати виконання команд не відображаються на екрані.

Опція -l задає текстовий файл, що містить конструктори CLIPS, які відразу запускаються на виконання. Використання цієї опції еквівалентно

використанню команди `load` відразу після запуску CLIPS.

Основним методом спілкування з CLIPS, використовуваним у даному навчальному курсі, є застосування командного рядка. Після появи в головному вікні CLIPS запрошення - `CLIPS >` - команди користувача можуть вводитися в середовище безпосередньо із клавіатури. Команди можуть бути викликами системних або користувацьких функцій, конструкторами різних даних CLIPS і т.д. У випадку виклику користувачем деякої функції, вона негайно виконується, і результат її роботи відображається користувачеві.

Для виклику функцій або операцій CLIPS використовує префіксну нотацію - аргументи завжди впливають після імені функції або операції. При виклику конструкторів CLIPS створює новий об'єкт відповідного типу, так чи інакше представляє деякого знання в системі. При введенні в середовище імені створеної раніше глобальної змінної CLIPS відобразить її поточне значення. Введення в середовище деякої константи просто приведе до її негайного відображення в головному вікні CLIPS.

Як приклад уведіть у середовище наступні команди (лістинг 5.1).

Лістинг 5.1. Приклад команд CLIPS

```
(+ 3 4)
(defglobal ?*x* = 3)
?*x*
red
```

Результат виконання цих команд наведений на рис. 5.4.

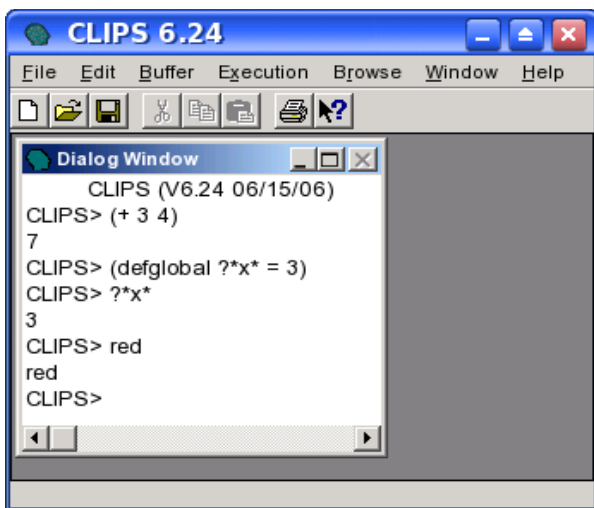


Рис.5. 7. Результат виконання команд лістингу 5.1

Першою командою даного приклада викликається функція арифметичного додавання двох чисел: 3 й 7. Результат роботи цієї функції - 7 відразу ж відображається на екран.

Після цього, за допомогою конструктора `defglobal`, створюється глобальна змінна `?*x*`, що ініціюється значенням 3. При введенні в командний рядок імені глобальної змінної `?*x*` CLIPS відображає її поточне значення, рівне 3. Останньою командою була уведена якась константа `red`, значення якої було відразу виведене на екран.

5.4. Синтаксис визначень

Як базовий синтаксис для визначення конструкцій мови використовується стандартна **БНФ-нотація**. Нижче наведені правила, використовувані для побудови визначень.

Слово або вираження, укладене в кутові дужки, називається нетермінальним символом (наприклад, `<string>`).

Нетермінальний символ вимагає подальшого визначення. Слова або вираження, не ув'язнені в кутові дужки, називаються термінальними символами, і представляють синтаксис описуваної конструкції мови CLIPS.

Термінальні символи (особливо круглі дужки) повинні вводитися в командний рядок саме так, як показано у визначенні. Якщо за нетермінальним символом треба символ `*`, то це означає, що в даному місці може перебувати список з нуля або більше елементів цього типу. Якщо ж за нетермінальним символом треба `+`, то в даному місці може перебувати список з одного або більше елементів цього типу. Символи `*` й `+`, що зустрічаються самі по собі (не наступні після нетермінальних символів), є термінальними. Багатокрапки, як горизонтальні, так і вертикальні, також використовується для відображення списку з одного або більше елементів. Елементи, укладені у квадратні дужки (наприклад, `[<коментарі>]`), є необов'язковими елементами, які можуть входити у визначення. Вертикальна риса, що розділяє два або більше елементи визначення, указує на те, що в конструкції необхідно використати один з перерахованих елементів. Символ `::=` використовується для позначення необхідності заміни деякого нетермінального символу. Наприклад, визначення:

`<lexeme> ::= <symbol> I`
`<string>` позначає, що нетермінальний символ **`<lexeme>`**, що зустрічається в деякому визначенні, повинен бути замінений або на символ **`<symbol>`**, або на символ **`<string>`**.

Пробіли, символи табуляції, переходи на інший рядок використовуються тільки для логічного поділу елементів визначення й ігноруються CLIPS (крім рядків, укладених у подвійні лапки).

Тема 6. Огляд можливостей системи CLIPS

6.1. Основні елементи мови

Синтаксис мови CLIPS можна розбити на три основних групи елементів, призначених для написання програм:

- примітивні типи даних;
- функції, що використовуються для обробки даних;
- конструктори, призначені для створення таких структур мови, як факти, правила, класи й т.д.

Розглянемо кожну із цих трьох груп більш докладно.

6.1.1. Типи даних

CLIPS підтримує 8 *примітивних типів* даних: **float**, **integer**, **symbol**, **string**, **external-address**, **fact-address**, **instance-name**, **instance-address**.

Для зберігання чисельної інформації призначаються типи **float** й **integer**, для символічної - **symbol** й **string**.

Число в CLIPS може складатися тільки із символів цифр (0-9), десяткової крапки (.), знака (+ або -) і експонентного символу (e) з відповідним знаком, у випадку подання числа в експонентній формі. Нижче наведені приклади припустимих в CLIPS подань цілих і речовинних типів:

Приклад 6.1. Подання чисел в CLIPS

Цілі: 237 15 +12 -32

Речовинні: 237e3 15.09 +12.0 -32.3e-7

Визначення цілого значення можна представити в такий спосіб:

Визначення 6.1. Подання цілого числа

```
<ціле> ::= [+ | -] <цифра>+
<цифра> ::= 0 | 1:2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Речовинне значення має наступний синтаксис:

Визначення 7.2. Подання речовинного числа

```
<речовинне> ::= <ціле>
<експонента> | <ціле> . [експонента] |
<беззнаковое-целое> [експонента] |
<ціле> . <беззнаковое-целое>
[експонента] <беззнаковое-целое> ::=
<цифра>+<експонента> : : = e | E
<ціле>
```

Якщо послідовність символів не відповідає наведеним вище визначенням цілого або речовинного числа, то дана послідовність сприймається CLIPS як значення типу **symbol**.

Значенням типу **symbol** може бути будь-яка послідовність символів, що починається з будь-якого не керуючого ASCII-символу. Значення типу **symbol** закінчується *обмежником*.

Обмежниками є будь-які невідображувані символи (наприклад, пробіл, символ табуляції або переходу на інший рядок), подвійні лапки, що відкривають або закриває кругла дужка, символи &, |, < й ~. Крапка з коми (;) є символом початку коментарів і також може обмежувати значення типу **symbol**. Обмежувачі-символи-обмежники не можуть утримуватися в значенні **symbol**, за винятком <, що може бути першим символом значення. Значення типу **symbol** не може починатися із символу ? або \$?, але може містити ці символи. CLIPS є мовою, чутливим до регістра.

Нижче наведені кілька прикладів значень типу **symbol**:

Приклад 6.2. Припустимі значення типу symbol

```
foo Hello B76-HI bad_value 127A
456-93-039 @+=% 2each
```

Значення типу, **string** являє собою рядок символів, укладену в подвійні лапки. Символ подвійних лапок також може бути включений у рядок. Для цього перед символом " необхідно поставити символ зворотної косої риси (\). Для включення в рядок символу зворотної косої риси необхідно використати два послідовних символи \. Приклади припустимих значень **string** наведені нижче:

Приклад 6.3. Припустимі значення типу string

```
"foo" "a and b" "1 number" "a\"quote"
```

Значення "abcd" типу **string** не еквівалентно значенню **abcd** типу **symbol**. Незважаючи на те, що вони складаються з ідентичних символів, вони *ставляться* до різних типів.

Значення типу **external-address** являє собою *адреса* структури даних, повернутою зовнішньою функцією (наприклад, написаною мовою 3 або Ada), *інтегрованої* із програмою CLIPS. Значення цього типу може бути *створено* тільки за допомогою виклику *зовнішньої функції*. Використання зовнішніх функцій виходить за рамки даної книги, тому ви не знайдете прикладів створення й використання цього типу. В CLIPS значення даного типу відображаються в такий спосіб:

Приклад 6.4. Значення типу external-address

<Pointer-XXXXXX>

де XXXXXX - число, що **представляє** зовнішню адресу.

Факт в CLIPS являє собою список атомарних значень примітивних типів, **посилатися** на які можна або використовуючи порядок визначення цих значень, у випадку *впорядкованих фактів*, або по імені, у випадку *неупорядкованих фактів* або *шаблонів*. Оперувати з фактом можна, використовуючи його адресу. Адреса факту являє собою значення типу fact-address.

Приклад 7.5. Значення типу fact-address

<Fact-XXX>

де XXX являє собою індекс факту.

Об'єкт в CLIPS являє собою **екземпляр певного користувачем класу**. Для визначення класу використовується конструктор defclass. Для створення об'єкта використовується функція make-instance. Посилатися на об'єкт можна або за адресою, або, у рамках окремого модуля, по імені об'єкта. Тип instance-name призначений для зберігання значення імені об'єкта. Для подання імені використовується значення типу symbol, оточене квадратними дужками ([й]). Нижче наведено кілька прикладів припустимих значень типу instance-name:

Приклад 7.6. Значення типу instance-name

[rump-l] [foo] [+++] [123-890]

Квадратні дужки не є частиною імені об'єкта, а служать своєрідними обмежниками, які дозволяють системі відрізнити значення типу instance-name від значення типу symbol.

Тип instance-address призначений для зберігання значення, що представляє адресу об'єкта. Значення цього типу може бути отримане за допомогою виклику функції instance-address або в результаті виконання операції зіставлення зразків у правилі. Посилання на об'єкт, з використанням значення типу instance-address, відбуваються значно швидше, ніж посилання за значенням instance-name. Значення типу instance-address відображаються в CLIPS у такий спосіб:

Приклад 7.7. Значення типу instance-address

<Instance-XXX>

де XXX - індекс об'єкта.

Місце для зберігання значення одного із примітивних типів в CLIPS називається **полем** або **простим полем**. **Константа** являє собою незмінне просте поле, задане послідовністю символів (за допомогою констант не можна задавати значення типів external-address, fact-address й instance-address — значення цих типів можуть бути отримані тільки за допомогою викликів відповідних функцій і повинні зберігатися в змінні). Послідовність із 0 або більше простих полів утворює **складене поле**. Для виведення складеного поля на екран CLIPS групує дані такого поля за допомогою круглих дужок. Кілька прикладів складових полів наведено нижче:

Приклад 7.8. Складові поля

(a) (1 bar foo) () (x 3.0 "red" 567)

Складене поле (a) не еквівалентно простому полю a.

Складені значення створюються або викликом функцій, що повертають складені значення, або за допомогою спеціального групового аргументу в конструкторах функцій, методів або оброблювачів повідомлень, або в результаті виконання процесу зіставлення зразків у правилах.

Змінної є значення деякого типу, збережене в простому або складеному полі й ім'я, що має деяке. Змінні використовуються в конструкторах CLIPS (зокрема в defrule, deffunction, defmethod й defmessage-handler). Опису використання змінних у конструкторах наведено у відповідних розділах.

6.1.2. Функції

Функцією в CLIPS називається частина коду, що має ім'я й повертає корисний результат або виконуючу корисну дію (наприклад, відображення інформації на екрані). Надалі в книзі функціями будуть називатися, як правило, тільки функції, що повертають результат. Функції, що не повертають результат і виконуючу деяку корисну роботу, як правило, називаються **командами**.

CLIPS оперує з декількома типами функцій — *певні користувачем зовнішні функції, системні (внутрішні) функції, функції, певні в середовищі CLIPS за допомогою конструктора deffunction, родові функції*. Певні користувачем зовнішні функції й системні функції створюються на зовнішніх мовах програмування (наприклад, 3), і потім підключаються до CLIPS на етапі компілювання або функціонування середовища.

Конструктор deffunction дозволяє користувачам визначати нові функції безпосередньо в CLIPS. Функції, створені таким чином, діють так само, як зовнішні або системні функції CLIPS, за винятком

того, що замість безпосереднього виконання (як, наприклад, у випадку виклику певної користувачем зовнішньої функції) виклик такої функції обробляється убудованим інтерпретатором мови CLIPS.

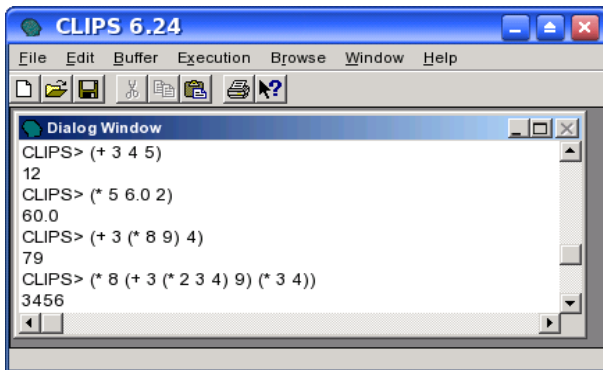
Родові функції визначаються за допомогою конструкторів `defgeneric` й `defmethod`. Родові функції дозволяють виконувати різні дії, залежно від набору аргументів, заданих при виклику функції. У такий спосіб функція перевантажується різними реалізаціями (подібний механізм перевантаження функцій можна зустріти, наприклад, у мові `C++`).

Виклик функцій в CLIPS має **префіксну нотацію** — аргументи функції завжди впливають після імені функції. При виклику ім'я функції разом з усіма аргументами полягає в круглі дужки.

Аргументи відокремлюються друг від друга принаймні одним пробілом. Аргументами функцій можуть бути змінні примітивних типів, константи або виклики інших функцій. Нижче наведені приклади використання функцій `+` (арифметичне додавання) і `*` (арифметичне множення):

Приклад 7.9. Використання функцій `+` й `*`

```
( + 3 4 5 )
( * 5 6.0 2 )
( + 3 ( * 8 9 ) 4 )
( * 8 (+ 3 (* 2 3 4) 9) )
( * 3 4 )
```



Виразом в CLIPS називається неіменованний відрізок коду, що викликає функції з деяким набором аргументів. Фактично попередній приклад складається із чотирьох виразень.

6.1.3. Конструктори

В CLIPS визначені наступні конструктори: `defmodule`, `defrule`, `def facts`, `def template`, `def global`, `def function`, `def class`, `def instances`, `def message-handler`, `def generic` й `def method`. Виклики всіх конструкторів полягають у круглі дужки. Конструктори відрізняються від убудованих

функцій по виконуваним ними діям. Як правило, функції не міняють стан бази знань середовища CLIPS (за деяким виключенням, наприклад, функцій, що очищають середовище або завантажують на виконання деякий файл). Конструктори, навпаки, призначені для додавання в базу знань нових елементів. Крім того, на відміну від функцій, конструктори не повертають ніяких значень.

Як й у будь-якій мові програмування, в CLIPS гарним тоном вважається використання коментарів. Всі конструктори (за винятком `def global`) дозволяють вставляти коментарі безпосередньо в код визначення нового елемента бази знань. Коментарі також можуть бути поміщені в будь-яке місце програми за допомогою символу `;`. Всі символи, що впливають за `;` до кінця рядка, ігноруються CLIPS. Коментарі, створені за допомогою символу `;`, не зберігаються в середовищі CLIPS, тому їхнє використання розумно тільки в текстових файлах. Надалі ви зустрінете множину прикладів використання коментарів.

6.2. Абстракції даних

Для подання даних CLIPS використовує три основних **абстракції даних**: факти, об'єкти й глобальні змінні. У даному розділі докладно описана кожна із цих форм подання інформації.

6.2.1. Факти

Факти — одна з основних форм подання інформації в CLIPS. Факти є фундаментальним поняттям теорії експертних систем і призначені для використання в *правилах* системи. Кожен факт представляє фрагмент даних, поміщених у поточний *список фактів системи (робочу пам'ять)*.

Факт може бути доданий у поточний список фактів системи (за допомогою команди `assert`), вилучений з нього (команда `retract`), змінений (`modify`) або продубльований (`duplicate`) користувачем, у процесі інтерактивної роботи в системі, або із програми. Кількість фактів, які може містити список фактів, а також кількість інформації, що втримується в кожному факті, обмежено тільки вільною пам'яттю вашого комп'ютера. У випадку виконання користувачем спроби додати в систему факт, що точно відповідає вже існуючому, дана операція буде ігнорована, хоча подібне поведіння системи можна змінити.

Деякі команди, такі як `retract`, `modify` або `duplicate`, вимагають як параметр деякого вже існуючого факту. Факт може бути заданий за допомогою значень типу `fact-index` або `fact-address`. Після створення (або зміни) факт одержує унікальний індекс, називаний *індексом факту* (`fact-`

index). Індекс фактів починаються з 0 і збільшується на 1 при кожній додаванні або зміні факту. При виконанні команди `reset` або `clear` поточний індекс фактів обнулюється. Для визначення конкретного факту за допомогою типу `fact-address` необхідно одержати відповідне значення від функції, що повертає значення даного типу (наприклад, `assert`, `modify` або `duplicate`), або деякого правила.

Для зручності відображення фактів в CLIPS використовується поняття *ідентифікатора факту*. Ідентифікатор факту складається із символу `f`, що впливає за ним знака `-` і індексу факту. Наприклад, ідентифікатор `f-10` посилається на факт із індексом 10.

Для зберігання фактів використовується один із двох наступних форматів: упорядковані факти й неупорядковані факти або шаблони.

6.2.1.1. Упорядковані факти

Упорядкований факт складається зі значення типу `symbol` і наступної за ним послідовності з нуля або більше значень типу `symbol`. Факт полягає в круглі дужки, а значення в послідовності відокремлюються друг від друга пробілами. Перше поле впорядкованого факту визначає так назване *відношення*, або *зв'язок* факту. Наприклад, факт `(father-of jack bill)` показує, що батьком Джека є Білл. Нижче наведено кілька прикладів упорядкованих фактів:

Приклад 7.10. Упорядковані факти

```
(the pump is on)
(altitude is 10000 feet)
(grocery-list bread milk eggs)
```

Поля в упорядкованому факті можуть зберігати дані будь-якого примітивного типу CLIPS, за винятком першого поля, тип якого повинен бути `symbol`. Наступні слова зарезервовані й не можуть бути використані як перше поле: `test`, `and`, `or`, `not`, `declare`, `logical`, `object`, `exist` й `forall`.

6.2.1.2. Неупорядковані факти

Тому що впорядкований факт зберігає інформацію, використовуючи строго задані позиції даних, то для доступу до необхідної інформації користувач повинен знати не тільки які дані збережені у факті, але і яке поле містить ці дані. *Неупорядковані факти (або шаблони)* надають користувачеві можливість задавати абстрактну структуру факту шляхом призначення імені кожному полю. Для створення шаблонів, які згодом будуть застосовуватися для доступу до полів факту по імені, використовується конструктор `deftemplate`. Конструктор `deftemplate`, по суті, аналогічний визначенням записів або структур у таких мовах програмування, як Pascal або C.

Конструктор `deftemplate` задає ім'я шаблону й визначає послідовність із нуля або більше полів неупорядкованого факту, названих *слотами*. Слот складається з імені, заданого значенням типу `symbol`, і наступної за ним списку полів. Як і факт, слот по обидва боки обмежується круглими дужками. На відміну від упорядкованих фактів слот неупорядкованого факту може жорстко визначати тип своїх значень. Крім того, слоту можуть бути задані значення за замовчуванням.

Слоти не можуть бути використані в упорядкованих фактах, а в неупорядкованих файлах, у свою чергу, не можна посылатися на дані, використовуючи порядок слотів.

CLIPS відрізняє неупорядковані факти від упорядкованих по першому полю факту. Перше поле фактів будь-якого типу повинне бути значенням типу `symbol`. Якщо це значення відповідає імені деякого шаблону, то факт є неупорядкованим. Як й упорядковані факти, неупорядковані обмежуються дужками.

Приклад 6.11. Неупорядковані факти

```
(client (name "Joe Brown") (id
X9345A))
(point-mass (x-velocity 100) (y-
velocity -200))
(class (teacher "Martha Jones") (#-
students 30) (Room "37A"))
(grocery-list (#-of-items 3) (items bread milk
eggs))
```

Порядок слотів у неупорядкованому факті не важливий. Наприклад, всі наведені нижче факти вважаються ідентичними:

```
(class (teacher "Martha Jones") (#-
students 30) (Room "37A"))
(class (#-students 30) (teacher "Martha
Jones") (Room "37A"))
(class (Room "37A") (#-students 30)
(teacher "Martha Jones"))
```

На відміну від фактів, наведених вище, упорядковані факти з наступного приклада не є ідентичними:

```
(class "Martha Jones" 30 "37A")
(class 30 "Martha Jones" "37A")
(class "37A" 30 "Martha Jones")
```

Очевидними перевагами застосування шаблонів є більше висока читабельність і незалежність слотів від порядку їхнього визначення.

Так само як й упорядковані факти, шаблони можна додавати в список фактів і видаляти з нього. Крім того, існує можливість модифікації й дублювання шаблонів.

6.2.1.3. Ініціалізація фактів

Конструктор `deffacts` дозволяє створювати набір фактів, ініціюючи базу знань CLIPS, при кожному очищенні системи. При виконанні команди `reset` поточний список фактів CLIPS очищається, а потім у нього додаються всі факти, задані конструкторами `deffacts`. CLIPS містить один визначений системний конструктор `deffacts`, що виконує додавання в систему факту `initial-fact`.

6.2.2. Об'єкти

Об'єкт CLIPS складається із двох основних частин - властивостей об'єкта і його поведінки. Клас являє собою своєрідний шаблон, що визначає загальні властивості й поведінки об'єктів - екземплярів цього класу. Об'єкти можуть належати класам, певним користувачем, або деяким системним класам (наприклад, класам, що реалізують подання об'єктів у вигляді примітивних даних). Нижче наведені кілька прикладів об'єктів і назви їхніх класів:

Приклад 7.12. Об'єкти і їхні класи

Об'єкти (вид, відображуваний на екран)	Класи
<code>Rolls-Royce</code>	<code>SYMBOL</code>
<code>"Rolls-Royce"</code>	<code>STRING</code>
<code>8.0</code>	<code>FLOAT</code>
<code>8</code>	<code>INTEGER</code>
<code>(8.0 Rolls-Royce 8 [Rolls-Royce])</code>	<code>MULTIFIELD</code>
<code><Pointer- OOCF61AB></code>	<code>EXTERNAL-ADDRESS</code>
<code>[Rolls-Royce]</code>	<code>CAR</code>

Об'єкти CLIPS розділяються на дві важливі категорії: *об'єкти, що зберігають примітивні типи даних*, і *об'єкти класів, певних користувачем*. Об'єкти цих двох типів відрізняються способом свого створення й видалення, наборами властивостей і навіть методом використання.

Об'єкти примітивних типів неявно створюються й віддаляються системою CLIPS у місцях, де це необхідно. По посиланню на такий об'єкт можна одержати значення, що зберігається в ньому, що відповідає типу. Об'єкти примітивних типів не мають слотів й, як правило, не мають імен. Класи, що визначають ці об'єкти, є визначеними класами CLIPS. Функціональність визначених класів, що визначають об'єкти примітивних типів, подібна функціональності класів, певних користувачем, за винятком того, що до таких класів не можна приєднати оброблювачі повідомлень. Це робить не дуже зручним використання таких класів в об'єкто-орієнтованому програмуванні. Основним призначенням об'єктів примітивних типів є використання їх у визначенні родових функцій. Родові функції застосовують ці об'єкти як свої аргументи й, по заданому наборі, у момент виклику, визначають, який саме метод родової функції

повинен бути викликаний.

Для посилання на об'єкт класу, певного користувачем, необхідно використати ім'я або адресу об'єкта. Такі об'єкти явно створюються або віддаляються за допомогою повідомлень або спеціальних системних функцій. Властивості об'єкта класу, певного користувачем, визначаються набором *слотів*, заданих при визначенні класу. Слот об'єкта має ім'я й може містити просте або складене значення. Наприклад, об'єкт `Rolls-Royce` є об'єктом створеного користувачем класу `CAR`. Один зі слотів такого об'єкта може, наприклад, містити ціну автомобіля зі значенням `75 000`. Поведінки об'єктів визначається наявністю тих або інших *оброблювачів повідомлень*, приєднаних до відповідного класу.

Всі об'єкти одного класу мають однакові набори слотів, але можуть містити в цих слотах різні значення. Однак, якщо два об'єкти мають однакові набори слотів, це ще не означає, що вони належать одному класу, тому що два абсолютно різних класи, теоретично, можуть мати однакові набори слотів.

Основна різниця між слотами об'єкта й неупорядкованого факту полягає в можливості *спадкування*. Спадкування дозволяє використати в класі деякі властивості й поведінки, певні в іншому класі. `COOL` (об'єкто-орієнтована частина мови CLIPS) підтримує *множинне спадкування*, при якому клас може успадковувати слоти й оброблювачі повідомлень безпосередньо від декількох класів.

6.2.2.1. Ініціалізація об'єктів

Конструктор `definstances` дозволяє створювати набір об'єктів, що додаються в базу знань CLIPS при кожному очищенні системи. При виконанні команди `reset` середовище CLIPS очищається, а потім у список об'єктів додаються всі об'єкти, задані конструкторами `definstances`. CLIPS містить один визначений системний конструктор `definstances`, що викликає додавання в систему об'єкта `initial-object`.

6.2.3. Глобальні змінні

Конструктор `defglobal` призначений для визначення *глобальних змінних*. Доступ до таких змінних можна одержати з будь-якого місця середовища CLIPS, а значення, які вони містять, не залежать ні від яких інших конструкцій мови. На відміну від цього, деякі конструктори (наприклад, `defrule` або `deffunction`) дозволяють створювати локальні змінні. Ці локальні змінні доступні тільки усередині тіла відповідного правила або функції. Глобальні змінні CLIPS подібні глобальним змінним, що зустрічаються в таких традиційних процедурних мовах, як **C** або **Ada**. Однак, на

відміну від них, глобальні змінні CLIPS є слабо типізованими. Вони здатні зберігати значення будь-якого типу.

6.3. Подання знань

CLIPS підтримує як *евристичну*, так і *процедурну парадигму подання знань*. Обидві ці парадигми описані в даному розділі.

6.3.1. Евристичні знання

Одним з основних методів подання знань в CLIPS є *правила*. Правила використовуються для подання евристик або емпіричних правил, що визначають дії, які необхідно виконати у випадку виникнення деякої ситуації. розробник експертної системи створює набір правил, які, працюючи разом, вирішують поставлену задачу. Правила складаються з *передумов* і *наслідку*. Передумови називаються також *Якщо-частиною* правила або *LHS* правила (left-hand side). Наслідок називається *Т-частиною* правила або *RHS* правила (right-hand side).

Передумови правила являють собою набір *умов* (або *умовних елементів*), які повинні задовольнитися, для того щоб правило виконалося. Передумови правил задовольняються залежно від наявності або відсутності деяких заданих фактів у списку фактів або деяких створених об'єктів, що є екземплярами класів, певних користувачем. Один з найпоширеніших типів умовних виражень в CLIPS — *зразки* (patterns). Зразки складаються з набору *обмежень*, які використовуються для визначення того, чи задовольняє деякий факт або об'єкт умовному елементу. Інакше кажучи, зразок задає деяку маску для фактів або об'єктів. Процес зіставлення зразків фактам або об'єктам називається *зіставленням зразків* (pattern-matching). CLIPS надає механізм, називаний *механізмом логічного висновку* (inference engine), що автоматично зіставляє зразки з поточним списком фактів і певних об'єктів і шукає правила, які застосовні в даний момент.

Наслідок правила представляється набором деяких дій, які потрібно виконати, у випадку якщо правило застосовне до поточної ситуації. Таким чином, дії, задані в наслідку правила, виконуються по команді механізму логічного висновку, якщо всі передумови правила задоволені. У випадку, якщо в цей момент застосовно більше одного правила, механізм логічного висновку використовує поточну *стратегію дозволу конфліктів* (conflict resolution strategy), що визначає, яке саме правило буде виконано. Після цього CLIPS виконує дії, описані внаслідок обраного правила (які можуть вплинути на список застосовних правил), і приступає до вибору наступного правила. Цей процес триває доти, поки список застосовних правил не

спорожніє.

У більшості випадків правила CLIPS можна представити у вигляді операторів IF-THEN, використовуваних у процедурних мовах програмування, наприклад, таких як Ada або C. Однак умовні вираження IF-THEN у процедурних мовах перевіряються тільки тоді, коли потік керування програми безпосередньо попадає на дане вираження шляхом послідовного перебору виражень й операторів, що становлять програму. В CLIPS, на відміну від цього, механізм логічного висновку створює й постійно модифікує список правил, умови яких у цей момент задоволені. Ці правила запускаються на виконання механізмом логічного висновку. Із цієї сторони правила схожі на оброблювачі повідомлень, що є присутнім у таких мовах, як, наприклад, Ada або Smalltalk.

6.3.2. Процедурні знання

Крім евристичної, CLIPS підтримує й *процедурну парадигму подання знань*, використовувану в більшості мов програмування. Конструктори deffunction й defgeneric дозволяють користувачеві визначати нові виконувані конструкції безпосередньо в середовищі CLIPS, що повертають деякі значення або виконують якісь корисні дії. Виклик цих нових функцій нічим не відрізняється від виклику убудованих функцій CLIPS. Оброблювачі повідомлень дозволяють користувачеві визначати поводження об'єктів, за допомогою завдання тієї або іншої реакції на повідомлення. Функції, родові функції й оброблювачі повідомлень являють собою шматки коду, заданого користувачем і виконуваного, якщо буде потреба, інтерпретатором CLIPS. Крім того, механізм модулів (конструктор defmodule) дозволяє розбивати базу знань CLIPS на окремі змістовні частини.

6.3.2.1. Функції

Конструктор deffunction дозволяє створювати нові *функції* безпосередньо в CLIPS. Більше ранні версії CLIPS дозволяли використати тільки зовнішні користувацькі функції, написані на якій-небудь мові програмування (найчастіше Сі) і приєднані до середовища CLIPS.

Тіло функції, певної за допомогою конструктора deffunction, являє собою послідовність дій, подібну використовуваної в правій частині правил. Задані користувачем дії виконуються при виклику відповідної функції. Значення, що повертає функцією, є результатом обчислення останньої дії.

6.3.2.1.1. Родові функції

Родові функції, так само як і звичайні функції, можуть бути створені безпосередньо в CLIPS.

Спосіб виклику таких функцій також нічим не відрізняється від способу виклику звичайних функцій. Однак родові функції набагато могутніші звичайних, тому що вони здатні перевантажуватися. Завдяки механізму перевантаження родова функція може виконувати різні дії залежно від типу й числа аргументів. Звичайно родова функція складається з декількох компонентів, названих *методами*. Кожен метод містить різний набір аргументів родової функції.

Наприклад, можна перевантажити системну функцію + (арифметичне додавання) для виконання операції конкатенації двох рядків. Однак після цього функція + усе ще зможе виконувати арифметичне додавання. У даному прикладі в родової функції + існує два методи: перший метод явно визначений користувачем для конкатенації двох рядків, другий являє собою неявний виклик стандартної функції, що виконує арифметичне додавання. Значення, повернуте родовою функцією, є значенням, отриманим у результаті обчислення останньої дії в застосовуваному методі.

6.3.2.1.2. Оброблювачі повідомлень

Як уже згадувалося, об'єкт CLIPS складається із двох основних частин — властивостей об'єкта і його поводження. Властивості об'єктів визначаються в термінах слотів. Поводження об'єкта обумовлюється *оброблювачами повідомлень*, які є приєднаною до класу послідовністю дій із заданим ім'ям. Будь-які маніпуляції з об'єктом можна виконати тільки за допомогою повідомлень. Наприклад, у випадку вже згаданого об'єкта Rolls-Royce, що є екземпляром користувальницького класу car, для того щоб запустити двигун, користувач повинен послати об'єкту повідомлення start-engine за допомогою функції send. То, яким образом об'єкт Rolls-Royce прореагує на це повідомлення, залежить від визначення оброблювача повідомлення start-engine, пов'язаного із класом car. Призначення оброблювачів повідомлень у принципі еквівалентно призначенню будь-якої функції — повернення результату або виконання якихось корисних дій.

6.3.2.1.3. Модулі

Модулі дозволяють розбивати базу знань на окремі змістовні частини. Кожен викликуваний конструктор міститься в певний модуль. Програміст має можливість контролювати можливість доступу й видимість конструкторів у тих або інших модулях. Крім того, для модулів можна встановлювати видимість певних фактів або об'єктів. Модулі можна використати для контролю або зміни потоку виконання правил.

6.4. Об'єкто-орієнтовані можливості CLIPS

Даний розділ дає короткий огляд елементів мови **CLIPS Object-Oriented Language (COOL)** - убудованої мови CLIPS, що надає об'єкто-орієнтовані можливості.

6.4.1. Відмінності COOL від інших об'єкто-орієнтованих мов

У так званих "чистих" об'єкто-орієнтованих мовах абсолютно всі програмні елементи є об'єктами, і будь-які дії над ними виконуються за допомогою посилки повідомлень. В CLIPS об'єктами є тільки об'єкти класів, певних користувачем, і об'єкти, що представляють дані примітивних типів CLIPS. З об'єктами, що представляють дані примітивних типів, можна маніпулювати за допомогою повідомлень, а для об'єктів класів, певних користувачем, це є єдиною можливим способом роботи з об'єктом. Наприклад, в "чистих" об'єкто-орієнтованих мовах для додавання двох чисел першому з них передається повідомлення add й як аргумент передається друге. В CLIPS для цього досить просто викликати функцію + й як аргументи передати їй два числа. Однак ви можете визначити відповідний оброблювач повідомлення add для класу number і працювати із числами в стилі "чистих" систем ООП.

Робота з усіма програмними елементами CLIPS, що не є об'єктами, виконується не в об'єкто-орієнтованому стилі, а за допомогою викликів відповідних функцій. Наприклад, для висновку на екран визначення правила використовується функція ppdefrule, що потрібне правило передається як параметр, а не посилає повідомлення print, тому що правило не є об'єктом.

6.4.2. Основні можливості ООП

Будь-яка об'єкто-орієнтована система повинна мати наступні п'ятьма характеристики: абстрактністю, інкапсуляцією, спадкуванням, поліморфізмом і динамічним зв'язуванням. **Абстракція** — це спосіб подання даних на певному рівні деякої конкретної проблемної області. **Інкапсуляція** — це процес, що дозволяє приховувати деталі реалізації об'єкта за допомогою деякого певного для цього класу зовнішнього інтерфейсу. **Спадкування** дозволяє визначати класи, що використовують визначення інших класів і віх, що володіють всіма, і деякими своїми властивостями, якщо це необхідно. **Поліморфізм** — властивість, завдяки якій різні об'єкти по-різному реагують на ті самі повідомлення. **Динамічне зв'язування** є можливістю вибирати певний оброблювач повідомлення об'єкта під час виконання

програми. Розглянемо тепер, як CLIPS реалізує всі ці основні властивості системи ООП.

Побудова нового класу реалізує можливість абстрактного подання нового типу даних. Слоти й оброблювачі повідомлень цього класу визначають властивості й поведження цілої групи об'єктів, що належать цьому класу.

Інкапсуляція реалізується в CLIPS вимогою обов'язково використати повідомлення при роботі з об'єктами певних користувачем класів. Оброблювачі повідомлень класу являють собою доступний користувачеві інтерфейс, що приховує реалізацію класу.

COOL підтримує множинне спадкування. Це означає, що деякий клас може мати всі властивості зазначеного одного або більше *суперкласу*. Для встановлення лінійного порядку спадкування властивостей класів при множинному спадкуванні COOL використовує *список передування класів* (class precedence list), побудований з використанням ієрархії спадкування. Об'єкт, що представляє собою екземпляр нового класу, успадковує всі властивості (слоти) і поведження (оброблювачі повідомлень) кожного класу зі списку передування класів. Слово "передування" позначає, що властивості й поведження класу, що перебуває ближче до початку списку, перевизначають конфліктуючі визначення раніше, що зустрілися класів.

Різні COOL-об'єкти можуть реагувати на те саме повідомлення зовсім по-різному. Це реалізує

властивість поліморфізму. На практиці це виконується приєднанням до різних класів оброблювачів того самого повідомлення, але з різними послідовностями виконуваних дій.

CLIPS також підтримує можливість динамічного зв'язування, реалізовану за допомогою функції send, призначеної для посилки повідомлень об'єкту. Виклик цієї функції здійснюється саме в процесі виконання програми, таким чином, визначення оброблювача, що виконується, в той або інший момент також відбувається в процесі виконання програми. Наприклад, функція send може одержувати як параметр змінну, котра в різні моменти часу посилається на різні об'єкти, при цьому можуть викликатися зовсім різні оброблювачі.

6.4.3. Запити й набори об'єктів

На додаток до можливості використати об'єкти в процесі зіставлення зразків правил, COOL підтримує гнучку систему запитів, що дозволяє використати задані користувачем критерії для вибірки деякого набору об'єктів і виконання над ним певних дій. Запити дозволяють поєднувати в набори об'єкти самих різних класів. Запити можна використати, наприклад, для перевірки існування того або іншого набору об'єктів, виконання дій над набором або збереження посилання на набір для наступного використання.

Тема 7. CLIPS. Факти

Для функціонування будь-якої експертної системи критично важливим є наявність бази знань. Про це говорить навіть той факт, що останнім часом всі частіше термін "система, заснована на знаннях" (knowledge-base system) уживається як синонім терміна "експертна система". Як правило, у будь-якій експертній системі знання представляються фактами й правилами, заданими на деякій мові опису знань. CLIPS не є виключенням і надає можливості для придбання, зберігання й обробки фактів і правил. Дана лекція присвячена способам роботи з фактами в системі CLIPS. Робота із правилами буде описана в наступній темі.

7.1. Факти в CLIPS

Факти — одна з основних форм подання даних в CLIPS (існує також можливість подання даних у вигляді об'єктів і глобальних змінних, але про це мова йтиме пізніше). Кожен факт являє собою певний набір даних, що зберігає в поточному списку фактів - робочій пам'яті системи. Список фактів являє собою універсальне сховище фактів й є частиною бази знань. Об'єм списку фактів обмежений тільки пам'яттю вашого комп'ютера. Список фактів зберігається в оперативній пам'яті комп'ютера, але CLIPS надає можливість зберігати поточний список у файл і завантажувати список з раніше збереженого файлу.

У системі CLIPS фактом є список неподільних (або атомарних) значень примітивних типів даних. CLIPS підтримує два типи фактів - *упорядковані факти* (ordered facts) і *неупорядковані факти* або *шаблони* (non-ordered facts або template facts). Посилатися на дані, що втримуються у факті, можна або використовуючи строго задану позицію значення в списку даних для впорядкованих фактів, або вказуючи ім'я значення для шаблонів.

Факти можна додавати, видаляти, змінювати й дублювати, уводячи відповідні команди із клавіатури, або із програми. Всі відповідні команди будуть описані в даній лекції.

Після додавання факту в список фактів йому привласнюється цілий унікальний ідентифікатор, називаний *індексом факту* (fact-index). Індекс першого факту дорівнює нулю, надалі індекс збільшується на одиницю при додаванні кожного нового факту. CLIPS надає команди, що очищають поточний список фактів або всю базу знань, ці команди привласнюють поточному значенню індексу 0.

Деякі команди, наприклад зміни, видалення або дублювання фактів, вимагають вказівки певного факту. Факт можна задати або індексом факту, або його адресою. Адреса факту являє собою змінн-показник, що зберігає індекс факту. Процес створення адрес фактів

буде описаний нижче.

Упорядковані факти складаються з поля, що обов'язково є даним типу symbol і наступної за ним, можливо порожньої, послідовності полів, розділених пробілами. Обмеження факту служать круглі дужки.

Визначення 7.1. Упорядкований факт (дане_типу_symbol [поле]*)

Перше поле факту визначає так назване відношення, або зв'язок факту (relation). Термін "зв'язок" означає, що даний факт належить деякому певному конструктору або неявно оголошеному шаблону. Докладніше мова про це піде нижче. Приведемо кілька прикладів фактів:

Приклад 7.1 Упорядковані факти

```
(duck is bird)
(schoolboys is Bob Mike)
(Nuke did report)
(altitude is 1000 feet)
```

Кількість полів у факті не обмежено. Поля у факті можуть зберігати дані будь-якого примітивного типу CLIPS, за винятком першого поля, що обов'язково повинне бути типу symbol. Наступні слова зарезервовані й не можуть бути використані як перше поле: **test**, **and**, **or**, **not**, **declare**, **logical**, **object**, **exist** й **forall**. Ці слова можуть використатися як імена слотів шаблонів, хоча це не рекомендується.

Тому що впорядкований факт для подання інформації використовує строго задані позиції даних, то для доступу до неї користувач повинен знати не тільки які дані збережені у факті, але і яке поле містить ці дані. Неупорядковані факти (або шаблони) надають користувачеві можливість задавати абстрактну структуру факту шляхом призначення імені кожному полю. Для створення шаблонів, які згодом будуть застосовуватися для доступу до полів факту по імені, використовується конструктор `deftemplate`. Конструктор `deftemplate` аналогічний визначенням записів або структур у таких мовах програмування, як Pascal або C.

Конструктор `deftemplate` задає ім'я шаблону й визначає послідовність із нуля або більше полів неупорядкованого факту, названих також *слотами*. Слот складається з імені, заданого значенням типу symbol, і наступної за ним, можливо порожнього, списку полів. Як і факт, слот по обидва боки обмежується круглими дужками. На відміну від упорядкованих фактів слот неупорядкованого факту може жорстко визначати тип своїх значень. Крім того, слоту можуть бути задані значення за замовчуванням.

Слоти не можуть бути використані в упорядкованих фактах, а в неупорядкованих

фактах, у свою чергу, не можна посилалися на дані, використовуючи порядок слотів.

CLIPS розрізняє неупорядковані факти від упорядкованих по першому полю факту. Перше поле фактів будь-якого типу є значенням типу symbol. Якщо це значення відповідає імені деякого шаблону, то факт -упорядкований. Визначення неупорядкованого факту, як й упорядкованого, обмежується круглими дужками.

Нижче наведено кілька прикладів неупорядкованих фактів:

Приклад 7.2. Неупорядковані факти

```
(client (name "Joe Brown") (id X9345A))
(point-mass (x-velocity 100) (y-velocity -200))
(class (teacher "Martha Jones")
(#-students 30) (Room "37A"))
(grocery-list (#-of-items 3)
(items bread milk eggs))
```

Порядок слотів у неупорядкованому факті не важливий. Наприклад, всі наведені нижче факти вважаються ідентичними:

```
(class (teacher "Martha Jones") (#-students 30) (Room "37A"))
(class (#-students 30) (teacher "Martha Jones") (Room "37A"))
(class (Room "37A") (#-students 30) (teacher "Martha Jones"))
```

На відміну від фактів, наведених вище, упорядковані факти з наступного прикладу не є ідентичними:

```
(class "Martha Jones" 30 "37A")
(class 30 "Martha Jones" "37A")
(class "37A" 30 "Martha Jones")
```

З неупорядкованими фактами можна виконувати ті ж операції, що й з упорядкованими.

Далі розглянемо конструктори, операції й функції, які надає CLIPS для роботи з фактами.

7.2. Робота з фактами

CLIPS надає досить багатий набір можливостей для роботи з фактами за допомогою відповідних конструкторів, операцій і функцій. Ці можливості включають створення шаблонів за допомогою конструктора deftemplate, створення, зміна, видалення, пошук фактів, перегляд, збереження й завантаження списку фактів, визначення списку визначених фактів за допомогою конструктора deffacts і багато чого іншого.

7.2.1. Конструктор deftemplate

Для створення неупорядкованих фактів в CLIPS передбачений спеціальний конструктор deftemplate. Його використання приводить до появи в поточній базі знань

системи інформації про шаблон факту, за допомогою якого в систему надалі можна буде додавати факти, що відповідають даному шаблону. Таким чином, конструктор deftemplate аналогічний операторам record й struct таких процедурних мов програмування як Pascal або C.

Приведемо простий приклад використання конструктора deftemplate:

Приклад 7.3. Застосування конструктора deftemplate (deftemplate MyObject

```
(slot name)
(slot location)
(slot weight)
(multislot contents))
```

Як і всі конструктори CLIPS, конструктор deftemplate не повертає ніякого значення. При введенні даної команди в CLIPS ви повинні побачити результат, наведений на рис. 7.1.

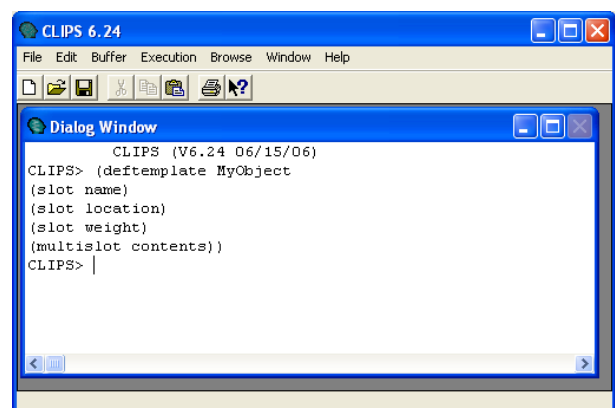


Рис. 7.1. Використання конструктора deftemplate

Подібна реакція середовища говорить про вдале додавання визначення шаблону в систему. Для перегляду всіх певних у поточній базі знань шаблонів можна скористатися командою get-deftemplate-list, мова про яку піде нижче, або спеціальним інструментом Deftemplate Manager (Менеджер шаблонів), доступним в Windows-версії середовища CLIPS. Для запуску менеджера шаблонів скористайтесь меню Browse і виберіть пункт Deftemplate Manager (мал. 7.2).

Менеджер шаблонів дозволяє в окремому вікні переглядати список всіх шаблонів, доступних у поточній базі знань, видалити обраний шаблон і відобразити всієї його властивості (наприклад, такі як імена й типи слотів). Зовнішній вигляд менеджера шаблонів представлений на мал. 7.3.

Після виконаної нами операції в поточній базі знань перебуває два шаблони, про що повідомляється в заголовку вікна менеджера (Deftemplate Manager — 2 Items). Перший шаблон є визначеним шаблоном initial-fact. Він не має слотів і завжди додається при запуску середовища. Його не можна видалити за допомогою менеджера, або переглянути його визначення. Призначення й приклади використання факту initial-fact будуть розглянуті нижче. Другим шаблоном є тільки що доданий шаблон MyObject. Менеджер шаблонів дозволяє вивести в головне вікно

середовища його визначення за допомогою кнопки **Pprint** або видалити його із середовища за допомогою кнопки **Remove**. На мал. 7.4 наведений результат послідовних операцій висновку інформації про визначення шаблону й видаленні його з поточної бази знань.

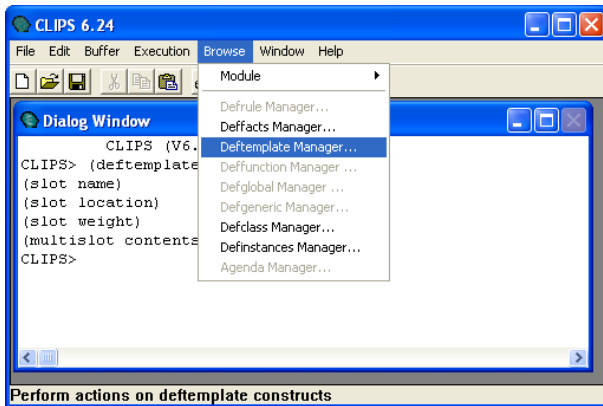


Рис. 7.2. Запуск менеджера шаблонів

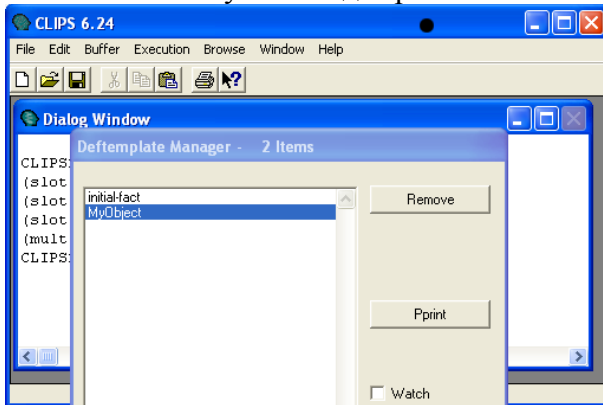


Рис. 7.3. Вікно менеджера шаблонів

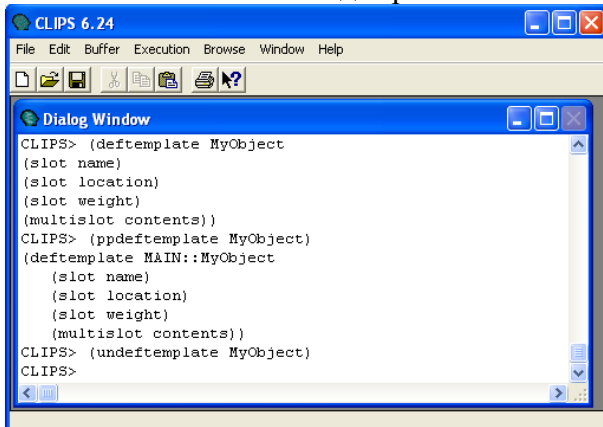


Рис. 7.4. Одержання інформації й видалення шаблону

Прапорець **Watch** дозволяє включати/виключати режим відображення повідомлень про використання шаблонів для кожного присутнього в системі шаблону в головному вікні середовища CLIPS. Якщо цей режим включений, користувач буде одержувати повідомлення при додаванні й видаленні неупорядкованих фактів, що використовують даний шаблон.

У випадку, якщо при додаванні нового шаблону за

допомогою конструктора `deftemplate` відбулася помилка, користувач одержить відповідне попередження.

Перевизначення вже існуючого шаблону приводить до виключення попереднього визначення. Шаблон не може бути перевизначений доти, поки він використовується (наприклад, фактом або правилом). Шаблон може мати будь-яку кількість простих або складених слотів. CLIPS відрізняє прості й складені слоти в шаблоні. Наприклад, буде помилкою зберігати значення складеного слоту в простий слот.

Розглянемо повний синтаксис конструктора `deftemplate`:

Визначення 8.2. Синтаксис конструктора `deftemplate` (`deftemplate` <імені-шаблону>[<необов'язков-коментарі>] [<определение-слота>*])

<определение-слота> ::= <определение-простого-слота>|<определение-составного-слота>

<определение-простого-слота> ::= (slot <ім'я-поля> <атрибута-шаблону>)

<определение-составного-слота> ::= (multislot <ім'я-поля> <атрибута-шаблону>)

<атрибута-шаблону> ::= <атрибуту-значенню-по-умовчання>|<атрибута-обмеження>

<атрибуту-значенню-по-умовчання> ::= (default ? DERIVE I ?NONE |<Вираз>)| (default-dynamic <Вираз>)

Помітимо ще раз, що імена шаблонів і слотів повинні бути значеннями типу `symbol`, крім того, на імена шаблонів поширюється заборона на використання деяких слів, зарезервованих системою, перерахованих вище.

Коментарі є необов'язковими й, як правило, описують призначення шаблону. Коментарі необхідно містити в лапки. Крім даного типу коментарів у конструкторі `deftemplate` також застосовні звичайні коментарі CLIPS, що починаються із символу `;`. Відмінність цих коментарів полягає в тім, що коментарі, що починаються із символу `;`, повністю ігноруються системою CLIPS, а коментарі, що впливають після імені шаблону й ув'язнені в лапки, зберігаються в базі знань системи. Ці коментарі доступні при перегляді визначення шаблону.

Визначимо в середовищі CLIPS наступний шаблон:

Приклад 8.4. Застосування конструктора `deftemplate` (`deftemplate` **MyObject** "Template for storage name and location" ; Slots for storage name and location (slot name) ;slot for name of object (slot location) ; slot for location of object

Після вдалого додавання шаблону в систему, за допомогою менеджера шаблонів, виведіть у головне

вікно CLIPS інформацію про визначення шаблону MyObject. Якщо перераховані дії були виконані без помилок, то на екрані повинні з'явитися повідомлення, ідентичні показаним на мал. 8.5.

Зверніть увагу, що коментарі *"Template for storage name and location"* збережені в пам'яті системи й відображаються разом з визначенням шаблону. На жаль, що ця версія CLIPS не сприймає символи кирилиці навіть як коментарі, тому всі коментарі прийде давати англійською мовою.

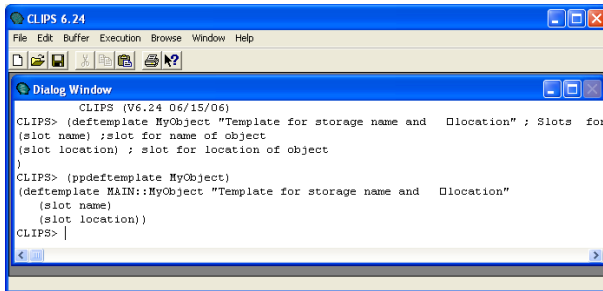


Рис. 8.5. Використання коментарів у конструкторі deftemplate

Крім ключового слова slot, що визначає простий слот, припустимо також застосування ключового слова multislot, для визначення складеного слоту. Простий слот, або слот, призначений для зберігання одиниці інформації одного із примітивних типів даних CLIPS. Складений слот здатн зберігати список подібн одиниц інформації необмежен об'єм. Для доступу до конкретних даних, що зберігається в складеному слоті, використовуються спеціальні групові символи й функції, приклади й правила використання яких будуть наведені нижче.

При створенні шаблону за допомогою конструктора deftemplate кожному полю можна призначити певні атрибути, що задають значення за замовчуванням або обмеження на значення слоту. Розглянемо ці атрибути детальніше.

<Атрибут-значення-за-замовченням> визначає значення, що буде використано у випадку, якщо при створенні факту не задане конкретне значення слоту. В CLIPS існує два способи визначення значення за замовчуванням, тому в конструкторі deftemplate передбачено два різних атрибути, що задає значення за замовчуванням: default й default-dynamic.

Атрибут default визначає статичне значення за замовчуванням. З його допомогою задається вираження, що обчислюється один раз при конструюванні шаблону. Результат обчислень зберігається разом із шаблоном. Цей результат привласнюється відповідному слоту в момент оголошення нового факту. У випадку якщо як значення за замовчуванням використається ключове слово ?DERIVE, те це значення буде витягнутися з обмежень, заданих для даного слоту. За замовчуванням для всіх слотів установлений атрибут default ?DERIVE.

В випадку якщо в місце вираз для значення за замовчуванням використається ключове слово ?NONE, те значення поля обов'язково повинне бути явно задане в момент виконання операції додавання факту. Додавання факту без визначення значень полів з атрибутом default ?NONE викличе помилку.

Атрибут default-dynamic призначений для установки динамічного значення за замовчуванням. Цей атрибут визначає вираз, що обчислюється щораз при додаванні факту по даному шаблоні. Результат обчислень привласнюється відповідному слоту.

Простий слот може мати тільки одне значення за замовчуванням. У складеного слоту може бути визначена будь-яка кількість значень за замовчуванням (кількість значень за замовчуванням повинне відповідати кількості даних, що зберігають у складеному слоті).

Нижче наведений приклад використання атрибута, що встановлює значення за замовчуванням:

Приклад 8.5. Використання атрибутів значення за замовчуванням

```
(deftemplate foo
  (slot w (default ?NONE))
  (slot x (default ?DERIVE))
  (slot y (default (gensym*)))
  (slot z (default-dynamic
    (gensym*))))
```

У конструкторі deftemplate підтримується перевірка статичних і динамічних обмежень.

Статична перевірка виконується під час використання визначення шаблону деякою командою або конструктором. Наприклад, для запису значень у слоти шаблону. Інакше кажучи, статична перевірка виконується до запуску програми. При невідповідності використовуваних значень із установленими обмеженнями користувачеві виводиться відповідне попередження про помилку.

Посилання на індекс факту в командах на зміну значення факту або його дублювання не зв'язує факт із відповідним шаблоном явно. Це робить статичну перевірку неоднозначною. Тому в командах, що використовують індекс факту, статична перевірка не виконується.

Статична перевірка обмежень включена за замовчуванням. Цю установку середовища CLIPS можна змінити за допомогою функції set-static-constraint-checking.

Крім статичної, CLIPS також підтримує динамічну перевірку обмежень. Якщо режим динамічної перевірки обмежень включений, то всі нові факти, створені з використанням деякого шаблону й певні значення, що мають,

перевіряються в момент їхнього додавання в список фактів.

У випадку якщо порушення заданих обмежень відбудеться в момент виконання динамічної перевірки в процесі виконання програми, то виконання програми припиняється й користувачеві буде видане відповідне повідомлення.

За замовчуванням в CLIPS відключений режим динамічної перевірки обмежень. Це середовище установки можна змінити за допомогою функції `set-dynamic-constraint-checking`.

Крім описаних вище функцій для зміни станів режимів статичної й динамічної перевірки обмежень, користувачам Windows-версії середовища CLIPS доступний візуальний спосіб налаштування цих установок. Для цього необхідно відкрити діалогове вікно **Execution Options**, вибравши пункт **Options** з меню **Execution**. Зовнішній вигляд цього діалогового вікна наведений на мал. 7.6.

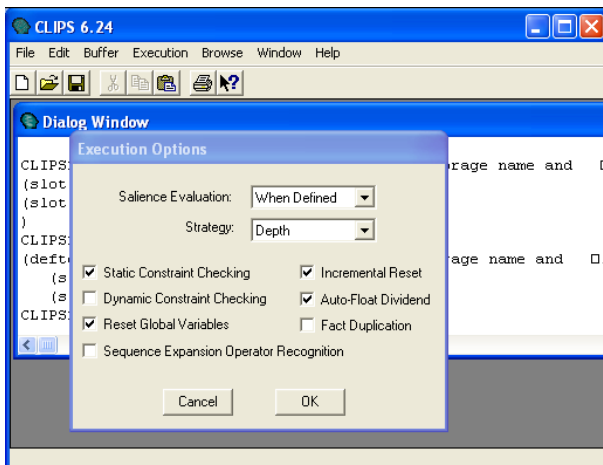


Рис. 7.6. Діалогове вікно **Execution Options**

Для включення або вимикання необхідних режимів перевірки обмежень атрибутів виставите у відповідне положення прапорці **Static Constraint Checking** й **Dynamic Constraint Checking** і натисніть кнопку **OK**.

Нижче наведений приклад використання атрибутів обмеження типу:

Приклад 7.6. Використання атрибутів обмеження

```
(deftemplate object
  (slot name
    (type SYMBOL)
    (default ?DERIVE))
  (slot location
    (type SYMBOL)
    (default ?DERIVE)))
```

Для повноти картини варто також згадати про неявно створюваних шаблонах. При використанні факту або посилання на впорядкований факт (наприклад, у

правилі) CLIPS неявно створює відповідний шаблон з одним складеним слотом. Ім'я неявно створеного складеного слоту не відображається при перегляді фактів. Неявно створеним шаблоном можна маніпулювати й порівнювати його з будь-яким тотожним, певним користувачем шаблоном, незважаючи на те, що він не має відображуваної форми.

7.2.2. Конструктор *deffacts*

Крім конструктора `deftemplates`, CLIPS надає конструктор `deffacts`, також призначений для роботи з фактами. Даний конструктор дозволяє визначати список фактів, які будуть автоматично додаватися щораз після виконання команди `reset`, що очищає поточний список фактів. Факти, додані за допомогою конструктора `deffacts`, можуть використатися й віддалятися так само, як і будь-які інші факти, додані в базу знань користувачем або програмою, за допомогою команди `assert`.

Визначення 7.3. Синтаксис конструктора `deffacts`

```
(deffacts <імена-списків-фактів>
  [<необов'язкові-коментарі>]
  [<факт>*])
```

Додавання конструктора `deffacts` з ім'ям уже існуючого конструктора приведе до видалення попереднього конструктора, навіть якщо новий конструктор містить помилки. У середовищі CLIPS можлива наявність декількох конструкторів `deffacts` одночасно й будь-яке число фактів у них (як упорядкованих, так і неупорядкованих). Факти всіх створених користувачем конструкторів `deffacts` будуть додані при ініціалізації системи.

Всі зауваження із приводу використання коментарів у конструкторі `deftemplate` застосовні й до конструктора `deffacts`.

У поля факту можуть бути включені динамічні вирази, значення яких будуть обчислюватися при додаванні цих фактів у поточну базу знань CLIPS.

Приклад 7.7. Використання конструктора `deffacts`

```
(deffacts startup "Refrigerator
Status"
```

```
(refrigerator light on)
(refrigerator door open)
(refrigerator temp (+ 5 10 15))
```

Зверніть увагу, що третій факт містить вираз, у даному прикладі суму трьох констант, але як вираз, ініціюючого значення факту, можуть використатися й більше складні вирази, наприклад, виклики функцій CLIPS або функцій, певних користувачем.

Перевірити роботу конструктора `deffacts` можна скориставшись діалогом **Watch Options**. Для цього виберіть пункт **Watch** меню **Execution** або використайте комбінацію клавіш `<Ctrl>+<W>`. У діалоговому вікні **Watch Options** включите режим перегляду зміни списку фактів, поставивши галочку

в поле **Facts**, як показано на мал. 8.7.

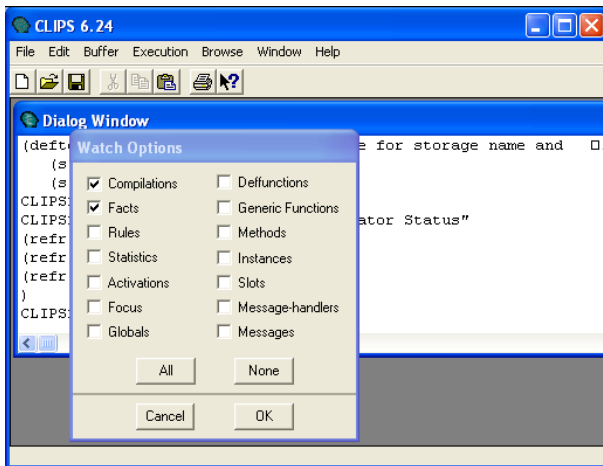


Рис. 7.7. Діалогове вікно Watch Options

Після цього натисніть кнопку OK й уведіть в CLIPS наведений вище конструктор deffacts. Потім у меню **Execution** виберіть пункт **Reset** (комбінація клавіш <Ctrl>+<E>). Якщо приклад був набраний правильно, то на екрані повинні з'явитися повідомлення, аналогічні наведеному на мал. 7.8.

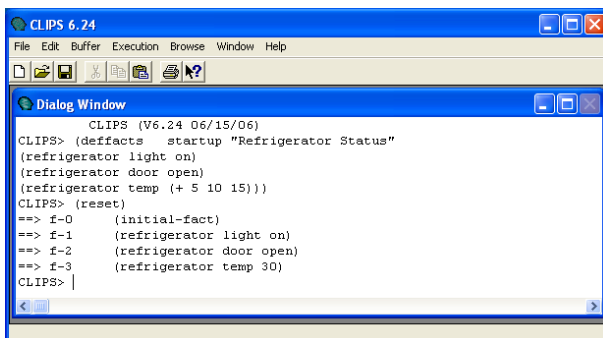


Рис. 7.8. Перегляд процесу додавання файлів

Так само, як і для конструкторів deftemplate, CLIPS надає візуальний інструмент для маніпуляції з певними в цей момент у системі конструкторами deffacts -- **Deffacts Manager** (Менеджер визначених фактів). Для запуску **Deffacts Manager** у меню **Browse** виберіть пункт **Deffacts Manager**. Зовнішній вигляд менеджера наведений на мал. 7.9.

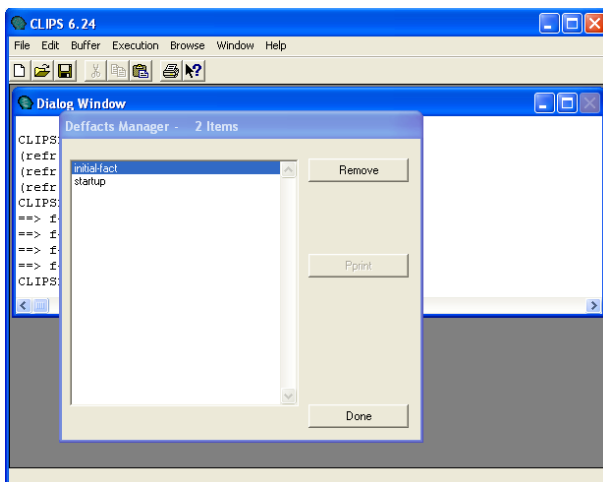


Рис. 7.9. Вікно менеджера визначених фактів

Менеджер відображає всі уведені на сучасний момент у систему конструктори deffacts. У нашому випадку це initial-fact, мова про яке піде нижче, і тільки що доданий нами startup. Менеджер дозволяє виводити в основне вікно CLIPS інформацію про визначення існуючих у цей момент у системі конструкторів deffacts за допомогою кнопки **Pprint** (крім deffacts initial-fact) і видаляти будь-який існуючий конструктор. Приклад висновку інформації про визначення конструктора deffacts startup наведений на мал. 8.10. Зверніть увагу, що коментарі, введені після імені конструктора, зберігаються й виводяться на екран так само, як у конструкторі deftemplate.

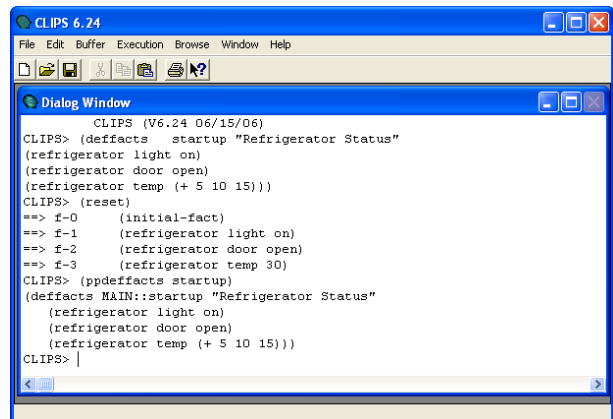


Рис. 7.10. Одержання інформації про певного конструктора

Під час запуску й після виконання команди **clear** CLIPS автоматично конструює наступні визначені шаблони й факти:

Визначення 7.4. Визначені шаблони й факти
(deftemplate initial-fact)
(deffacts initial-fact
(initial-fact))

Визначений факт initial-fact шаблону initial-fact надає зручний спосіб для запуску програм мовою CLIPS — правила, що не мають умовних елементів, автоматично перетворюються в правила з умовою, що перевіряє наявність факту initial-fact. Факт initial-fact можна обробляти так само, як і всі інші факти CLIPS, додані користувачем або програмою за допомогою команди **assert**. Приклад використання факту initial-fact буде наведений у наступній главі, відразу після першого знайомства із правилами CLIPS.

7.2.3. Функція assert

Функція **assert** - одна з найбільше часто застосованих команд у системі CLIPS. Без використання цієї команди не можна написати навіть найпростішу експертну систему й запустити її на виконання в середовищі CLIPS. Функції **Assert**, **retract** й **modify** - три робітники конячки, використовувані більшістю правил.

Функція `assert` дозволяє додавати факти в список фактів поточної бази знань. Кожним викликом цієї функції можна додати довільне число фактів. У випадку якщо був включений режим перегляду зміни списку фактів, то відповідне інформаційне повідомлення буде відображатися у вікні CLIPS при додаванні кожного факту.

Визначення 7.5. Синтаксис команди `assert`

(assert <факт>+)

При використанні команди `assert` необхідно пам'ятати, що перше поле факту обов'язково повинне бути значенням типу `symbol`. У випадку вдалого додавання фактів у базу знань, функція повертає адресу останнього доданого факту. Якщо під час додавання деякого факту відбулася помилка, команда припиняє свою роботу й повертає значення `FALSE`.

Слотам неупорядкованого факту, значення яких не задані, будуть привласнені значення за замовчуванням.

Приклад 7.8. Використання функції `assert`

```
(clear)
(assert (color red))
(assert (color blue)
      (value (+ 3 4)))
(deftemplate status
  (slot temp)
  (slot pressure
   (default low)))
(assert (status (temp high)))
```

Команда `clear` очищає поточний список фактів (а також всі певні конструктори, які вже були й ще буде розглянуті нижче). На відміну від `reset`, команда `clear` не додає в список фактів `initial-fact`. Цю команду також можна виконати, вибравши пункт **Clear CLIPS** у меню **Execution**. При виборі даної команди на екрані з'являється діалогове вікно, представлене на мал. 7.11. Це вікно запитує підтвердження користувача на очищення поточної бази знань.

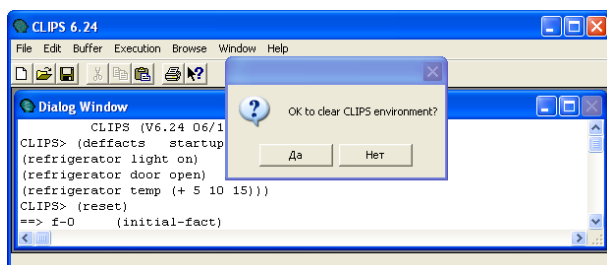


Рис. 7.11. Підтвердження очищення середовища CLIPS

У випадку, якщо команда була набрана із клавіатури, ніякого підтвердження на виконання цієї операції система не запитує. Якщо ви недавно почали працювати в середовищі CLIPS, то для очищення системи краще використати меню, тому що втрата всіх поточних даних з бази знань може

виявитися досить хворобливою.

Включите режим перегляду зміни списку фактів і наберіть наведений вище приклад. Після цього виконаєте команду (`facts`). Якщо при виконанні цих дій не було допущено помилок, то ви повинні одержати результат, ідентичний зображеному на мал. 7.12.

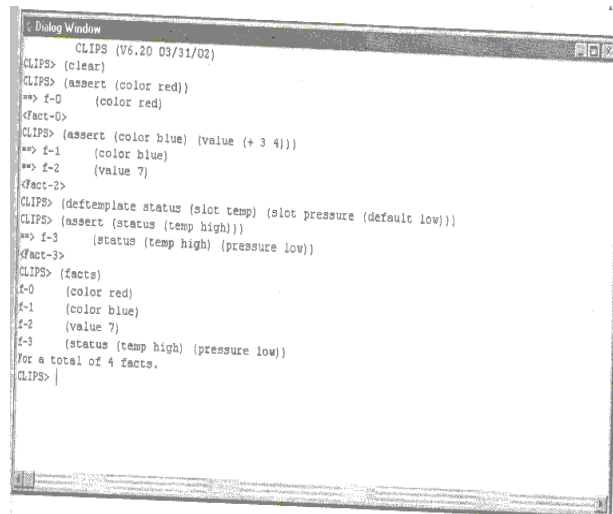


Рис. 7.12. Додавання фактів

Зверніть увагу, що при ініціалізації факту `value` використалося вираз, а слот `pressure` неупорядкованого факту `status` одержав значення за замовчуванням `low`.

За замовчуванням CLIPS не дозволяє додавати в список фактів два однакових факти. Наприклад, спроба додати два факти `color red` приведе до помилки й функція `assert` поверне значення `FALSE`. Дану установку системи можна змінити за допомогою функції `set-fact-duplication`. Крім того, користувачам Windows-версії CLIPS доступний ще один спосіб налаштування. Для цього необхідно відкрити діалогове вікно **Execution Options**, вибравши пункт **Options** з меню **Execution**, установити прапорець **Fact Duplication**. Зовнішній вигляд цього діалогового вікна наведений на мал. 7.6.

7.2.4. Функція *retract*

Після додавання факту в базу знань рано або пізно встане питання про те, як його відтіля видалити. Для видалення фактів з поточного списку фактів у системі CLIPS передбачена функція `retract`. Кожним викликом цієї функції можна видалити довільне число фактів. Видалення деякого факту може стати причиною видалення інших фактів, які логічно пов'язані із що видаляє. Крім того, видалення факту викликає видалення правил із *плану рішення поточної задачі*, активованих видаляють фактом, що, але про це мова йтиме в наступних главах. У випадку якщо був включений режим перегляду зміни списку фактів, то відповідне інформаційне повідомлення буде відображатися у вікні CLIPS при видаленні

кожного факту.

Визначення 7.6. Синтаксис команди retract

(retract <визначення-факту>+ \ *)

Аргумент <визначення-факту> може бути або змінної, пов'язаної з адресою факту за допомогою правила (ця можливість буде описана в наступній главі), або індексом факту без префікса (наприклад, 3 для факту з індексом f-3), або вираженням, що обчислює цей індекс (наприклад, (+ 1 2) для факту з індексом f-3). Якщо як аргумент функції retract використався символ *, то з поточної бази знань системи будуть вилучені всі факти. Функція retract не має значення, що повертає.

Для демонстрації роботи функції retract скористаємося ще одним візуальним інструментом, не описаним раніше. Він призначений для перегляду вмісту списку фактів у реальному часі. Цей інструмент доступний тільки користувачам Windows-версії системи CLIPS. Для того щоб активізувати перегляд списку фактів, поставте прапорець поруч із пунктом **Facts Window** меню **Windows**, як показано на мал. 8.13. Зовнішній вигляд інструмента перегляду списку фактів показаний на тім же малюнку. Відразу після запуску CLIPS цей список порожній.

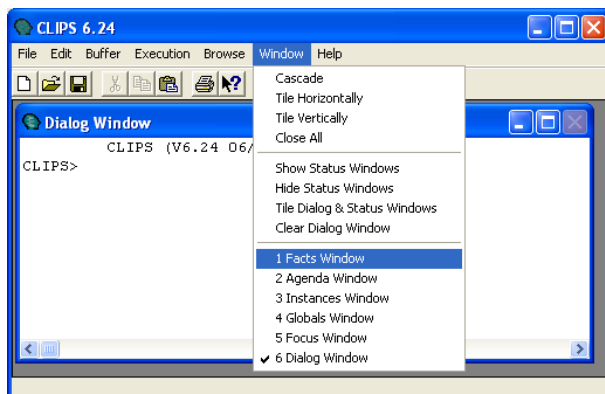


Рис. 7.13. Список фактів

Включите режим перегляду зміни списку фактів за допомогою діалогового вікна **Watch Options** і додайте в список фактів наступні факти:

Приклад 7.9. Додавання фактів

(assert (a) (b) (c) (d) (e) (f))

Зверніть увагу, що у вікні перегляду фактів тепер відображаються всі 6 доданих фактів.

Приклад 7.10. Видалення фактів

(retract 0 (+ 0 2) (+ 0 2 2))

Ця команда видалить всі факти з парними індексами, використовуючи індекс факту безпосередньо (перший аргумент) і вираження, що обчислює індекс факту (другий і третій аргумент). Якщо перераховані команди були виконані правильно, то результат повинен відповідати мал. 7.14.

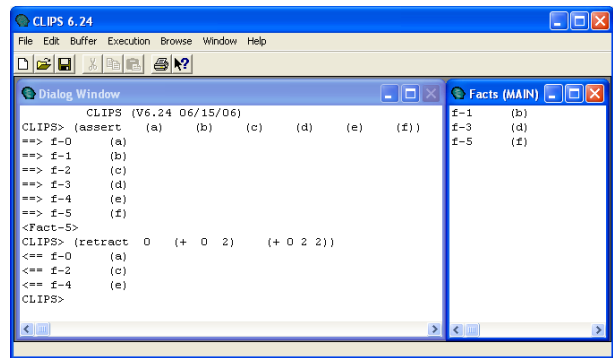


Рис. 7.14. Результат додавання й видалення фактів

У випадку, якщо факт із зазначеним індексом не буде знайдений, CLIPS видасть відповідне повідомлення про помилку.

Виконайте команду:

Приклад 7.11. Видалення всіх фактів

(retract *)

Після виконання даної команди список фактів буде очищено повністю й вікно відображення поточного стану списку фактів стане ідентично зображеному на мал. 7.13.

Необхідно помітити, що функція retract не робить ніякого впливу на індекс наступних доданих фактів, тобто цей індекс не обнулюється. Якщо після видалення всіх уведених фактів додати в систему який-небудь факт, то він одержить індекс f-б, незважаючи на те, що список фактів у цей момент порожній.

7.2. 8. Функція modify

Використовуючи функції assert й retract, можна виконувати більшість необхідних для функціонування правил дій. У тому числі й зміни існуючого факту. Наприклад, якщо в список фактів раніше був доданий факт (temperature is low), що одержав індекс 0, то змінити його значення можна, наприклад, у такий спосіб:

Приклад 8.12. Зміна існуючого факту

(clear)

(assert (temperature is low))

(retract 0)

(assert (temperature is high))

Для зміни впорядкованих фактів доступний тільки цей спосіб. Для спрощення операції зміни неупорядкованих фактів CLIPS надає функцію modify, що дозволяє змінювати значення слотів таких фактів. Modify просто спрощує процес зміни факту, але її внутрішня реалізація еквівалентна викликам пар функцій retract й assert. За один виклик modify дозволяє змінювати тільки один факт. У випадку вдалого виконання функція повертає новий індекс

модифікованого факту. Якщо в процесі виконання відбулася яка-небудь помилка, то користувачеві виводиться відповідне попередження й функція повертає значення FALSE.

Визначення 7.7. Синтаксис команди **modify** (**modify** <визначення-факту> <нове-значення-слота>+)

Аргументом <визначення-факту> може бути або змінна, пов'язана з адресою факту за допомогою правила, або індекс факту без префікса (наприклад, 3 для факту з індексом f-3). Після визначення факту треба список з одного або більше нових значень слотів зазначеного шаблона. Для використання наведеного вище приклада його необхідно переробити в такий спосіб:

Приклад 8.13. Зміна існуючого неупорядкованого факту

```
(deftemplate temperature (slot value) )
(assert (temperature (value low)) )
(modify 0 (value high) )
```

Якщо **включити** режим перегляду зміни списку фактів і виконати **наведені** вище команди, те отриманий результат повинен відповідати мал. 7.15.

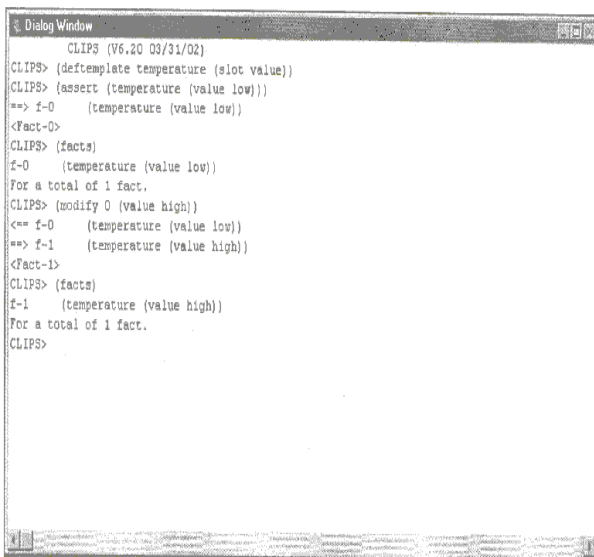


Рис. 7.15. Результат зміни існуючого неупорядкованого факту

Зверніть увагу на рух фактів у базі знань CLIPS при виконанні функції **modify** - спочатку віддаляється старий факт із індексом f-0, а потім додається новий факт із індексом f-1, ідентичний попередній, але з новим значенням заданого слоту.

Якщо в шаблоні заданого факту відсутній слот, значення якого потрібно змінити, CLIPS виведе відповідне повідомлення про помилку. Якщо заданий факт відсутній у списку фактів, користувач також одержить відповідне попередження.

7.2.6. Функція **duplicate**

Крім функції **modify**, в CLIPS існує ще одна дуже корисна функція, що спрощує роботу з фактами, — функція **duplicate**. Ця функція створює новий неупорядкований факт заданого шаблона й копіює в нього певну користувачем групу полів уже існуючого факту того ж шаблона. По діях, які виконує функція **duplicate**, аналогічна **modify**, за винятком того, що вона не видаляє старий факт зі списку фактів. Одним викликом функції **duplicate** можна створити одну копію деякого заданого факту. Як і функція **modify**, **duplicate**, у випадку вдалого виконання, повертає індекс нового факту, а у випадку невдачі — значення FALSE.

Визначення 7.8. Синтаксис команди **duplicate** (**duplicate** <визначення-факту> <нове-значення-слота>+)

Аргумент <визначення-факту> може бути або змінної, пов'язаної з адресою факту за допомогою правила, або індексом факту без префікса. Після визначення факту треба список з одного або більше нових значень слотів зазначеного шаблона. Продемонструємо роботу даної функції на наступному прикладі:

Приклад 8.14. Створення копії існуючого неупорядкованого факту

```
(deftemplate car
  (slot name)
  (slot producer)
  (slot type)
  (slot max-speed))
(assert ( car
  (name scorpio)
  (producer ford)
  (type sedan)
  (max-speed 180)))
(duplicate 0
  (type off-road)
  (max-speed 130))
```

У наведеному прикладі визначається шаблон, що описує властивості автомобіля, і додається факт - автомобіль Ford Scorpio з типом кузова седан і максимальна швидкість 180 (км/ч). Після цього за допомогою функції **duplicate** додається факт із інформацією про ще один автомобіль зі схожими характеристиками - це позашляховик Ford Scorpio з максимальною швидкістю 130 (км/ч). **Duplicate** просто полегшує нам життя, рятуючи від зайвого введення значень даних співпадаючих слотів.

У випадку, якщо додає з допомогою **duplicate** факт уже присутній у списку фактів, буде видана відповідна інформація про помилку й повернуте значення FALSE. Факт при цьому доданий не буде. Це поведінку можна змінити, дозволивши існування однакових фактів у базі знань.

7.2.7. Функція *assert-string*

Крім функції *assert*, CLIPS надає ще одну функцію, корисну при додаванні фактів, — *assert-string*. Ця функція приймає як єдиний аргумент символічний рядок, що є текстовим поданням факту (у тім виді, у якому ви набираєте його, наприклад, у функції *assert*), і додає його в список фактів. Функція *assert-string* може працювати як з упорядкованими, так і з неупорядкованими фактами. Одним викликом функції *assert-string* можна додати тільки один факт.

Визначення 7.9. Синтаксис команди *assert-string*
(*assert-string* <строков-вираження>)

Строкове вираження повинне бути укладене в лапки. Функція перетворить задане строкове вираження у факт CLIPS, розділяючи окремі слова на поля, з обліком певних у системі на сучасний момент шаблонів. Якщо в рядку необхідно записати внутрішнє строкове вираження, що представляє, скажемо, деяке поле, то для включення в строкове вираження символу лапок використовується *зворотна коса риса* (backslash). Наприклад, факт (*book-name* "CLIPS user Guide") можна додати в такий спосіб:

Приклад 7.15. Використання лапок усередині рядка

```
(assert-string "(book-name V'CLIPS User Guide\")(")
```

Для додавання, що втримується в поле символу зворотної косої риси використовуйте неї двічі. Якщо зворотна коса повинна втримуватися усередині підрядку, її необхідно використати чотири рази. Наприклад, для приміщення в поточний список факту (*a\b "c\d"*) необхідно викликати функцію *assert-string* з наступним строковим аргументом:

Приклад 7.16. Використання зворотної косої риси
(*assert-string* "(*a\b \"c\\\\d\"*)")

Якщо додавання факту пройшло вдало, функція повертає індекс тільки що доданого факту, у противному випадку функція повертає повідомлення про помилку й значення *FALSE*. Функція *assert-string* не дозволяє додавати факт у випадку, якщо такий факт уже присутній у базі знань (якщо ви ще не включили можливість присутності однакових фактів).

7.2.8. Функція *fact-existp*

У цьому розділі розглянемо дуже просту, але надзвичайно важливу функцію *fact-existp*. Ця функція визначає, є присутнім чи в цей момент факт, заданий індексом або змінної покажчиком, у базі знань системи. У випадку якщо факт присутній у списку фактів, функція

повертає значення *TRUE*, інакше — *FALSE*.

Визначення 7.10. Синтаксис команди *fact-existp*
(*fact-existp* <визначення-факту>)

Звичайно ця функція застосовується в правилах, описаних у наступній главі. Приведемо простий приклад використання даної функції:

Приклад 8.17. Використання функції *fact-existp*
(*clear*)
(*assert-string* "(*a\b \"c\\\\d\"*)")
(*fact-existp* 0)
(*retract* 0)
(*fact-existp* 0)

Не забудьте виконати команду *clear*, щоб доданий факт мав нульовий індекс. Після першого виклику функція *fact-exist* поверне значення *TRUE*, а після видалення факту з індексом 0 — *FALSE*.

7.2.9. Функції для роботи з неупорядкованими фактами

Для роботи з неупорядкованими фактами в CLIPS передбачений цілий ряд спеціальних функцій. До них **ставляться**: *fact-relation*, *fact-slot-names* й *fact-slot-value*. **Розглянемо** ці функції один по одному.

Функція *fact-relation* дозволяє **одержати зв'язок** (*relation*) існуючого факту із шаблоном. **Зв'язок** факту із шаблоном, **певним** за допомогою конструктора *deftemplate* або неявно створеним шаблоном, **визначається** по **першому полю** факту. Це поле завжди є простим полем і **використається** CLIPS як ім'я **шаблону**, з яким **зв'язаний** факт. Таким чином, функція *fact-relation* просто повертає перше поле факту, або значення *FALSE*, якщо **зазначений** факт не знайдений.

Визначення 8.11. Синтаксис команди *fact-relation*
(*fact-relation* <визначення-факту>)

Як визначення факту, як й в описані вище функціях, потрібно використати або змінну покажчик, що містить адресу факту, або індекс факту.

Приклад 8.18. Використання функції *fact-relation*
(*clear*)
(*assert* (*car* *Ford*))
(*fact-relation* 0)
(*retract* 0)
(*fact-relation* 0)

У першому випадку функція *fact-relation* поверне значення *car*, а в другому -*FALSE*.

Для одержання імен всіх слотів заданого факту в CLIPS призначена функція *fact-slot-names*.

Визначення 7.12. Синтаксис команди fact-slot-names

(fact- slot-names <визначення-факту>)

Дана функція повертає список імен слотів у складеному полі. Для впорядкованих фактів функція повертає значення implied (який мається на увазі), тому що, якщо ви **помнете**, CLIPS **представляє** впорядковані факти як неявно задані **неупорядковані** з одним **складеним** слотом. У випадку якщо заданий факт не знайдений, функція повертає значення FALSE.

Приклад 7.19. Використання функції fact-slot-names

```
(clear)
(deftemplate car
  (slot name)
  (slot producer)
  (slot type)
  (slot max-speed))
(assert ( car
  (name scorpio)
  (producer ford)
  (type sedan)
  (max-speed 180)))
(fact-slot-names 0)
```

Якщо **наведений** приклад був набраний без помилок, то функція fact-slot-names поверне значення (name producer type max-speed).

Останньою з розглянутих у даній лекції функцій для роботи з неупорядкованими фактами буде функція fact-slot-value.

Визначення 7.13. Синтаксис команди fact-slot-value

(fact-slot-value <визначення-факту> <им'я-слота >)

Дана функція дозволяє **одержувати** значення слоту деякого заданого факту. Якщо факт є **впорядкованим**, то для одержання значення неявно **певного складеного** слоту **використається** значення implied. У випадку якщо **зазначений** факт не існує, або ім'я слоту **зазначене** не вірно, функція повертає значення FALSE.

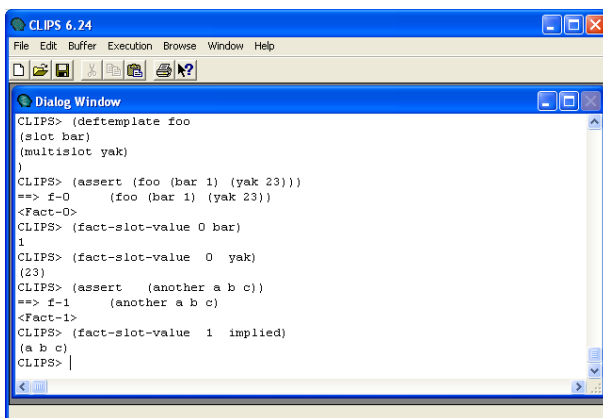


Рис. 7.16. Результат використання функції fact-slot-value

Виконаєте в середовищі CLIPS наступний

приклад:

Приклад 7.20. Використання функції fact-slot-value

```
(clear)
(deftemplate foo
  (slot bar)
  (multislot yak)
(assert (foo (bar 1) (yak 23)))
(fact-slot-value 0 bar)
(fact-slot-value 0 yak)
(assert (another a b c))
(fact-slot-value 1 implied)
```

Якщо попередній приклад був виконаний без помилок, то отриманий результат повинен відповідати наведеному на мал. 7.16.

7.2.10. Функції збереження й завантаження списку фактів

Як можна помітити, наповнення списку фактів в CLIPS досить кропіткє й тривале заняття. Якщо фактів досить багато, цей процес може розтягтися на кілька годин, або навіть днів. Тому що список фактів зберігається в оперативній пам'яті комп'ютера, теоретично, через збій комп'ютера або, наприклад, несподіваного відключення харчування, список фактів можна безповоротно втратити. Щоб цього не відбулося, а так само для того щоб зробити роботу з наповнення бази знань фактами більше зручної, CLIPS надає команди збереження й завантаження списку фактів у файл - save-facts й load-facts відповідно.

Визначення 7.14. Синтаксис команди save-facts

(save-facts <імені-файлу> [<межі-видимості> <списків-шаблонів >])
<межі-видимості> ::= visible|local

Команда save-facts зберігає факти з поточного списку фактів у текстовий файл. На **кожен** факт приділяється **один рядок**. Неупорядковані факти зберігаються разом з іменами слотів. У функції існує можливість обмежити **область** видимості фактів, що зберігають. Для цього **використається** аргумент <межі-видимості>. Він може приймати значення local або visible. У випадку якщо цей аргумент приймає значення visible, те зберігаються всі факти, що є **присутнім** у цей момент у системі. Якщо як аргумент **використається** ключове слово local, то зберігаються тільки факти з поточного модуля. За замовчуванням аргумент <межі-видимості> приймає значення local. Після аргументу <межі-видимості> може **впливати** список **певних** у системі шаблонів. У цьому випадку будуть збережені тільки ті факти, які пов'язані із **зазначеними** шаблонами.

Приклад 7.21. Використання функції save-facts

(clear)

```
(deftemplate template
(slot a)
(slot b))
(assert (template (a 1) (b
2)))
(assert (simple-fact1) (simple-
fact2))
(save-facts fl local template
simple-fact1)
```

Послідовність дій, наведена в даному прикладі, зберігає у файл fl, що перебуває в поточному каталозі, всі факти, видимі в поточному модулі й пов'язані із шаблонами template й simple-fact1 (як ви пам'ятаєте, після додавання факту simple-fact1 CLIPS визначає неявно створений шаблон simple-fact1). У результаті буде отриманий текстовий файл із наступним змістом:

Приклад 7.22. Зміст файлу fl

```
(template (a 1) (b 2) )
(simple-fact1)
```

У випадку успішного виконання, команда повертає значення true, а у випадку невдачі — відповідне повідомлення про помилку. Якщо зазначений файл уже існує, він буде перезаписаний.

Для завантаження збережених раніше файлів використається функція load-facts. Функція має наступний формат:

Визначення 7.15. Синтаксис команди load-facts

```
(load-facts <ім'я-файлу>)
```

Тут <ім'я-файлу> - ім'я текстового файлу, збереженого раніше за допомогою команди save-facts, що містить список фактів. Файл зі списком фактів можна також створити в будь-якому текстовому редакторі, якщо ви добре розібралися з поданням фактів в CLIPS. Для завантаження збереженого в попередньому прикладі файлу виконаєте:

Приклад 7.23. Використання функції load-facts

```
(load-facts fl)
```

У випадку успішного виконання команда повертає значення true, а у випадку невдачі — false і відповідне повідомлення про помилку. Зверніть увагу, що якщо у файлі втримуються факти, пов'язані з явно створеними за допомогою конструктора deftemplate шаблонами, то в момент завантаження всі необхідні шаблони повинні бути вже визначені в системі. Якщо ця умова не буде виконано, то завантаження фактів закінчиться невдало. На щастя, CLIPS також дозволяє й завантаження конструкторів з текстового файлу, але про це ми поговоримо в наступній главі, після розгляду конструктора defrule.

Тема 8. CLIPS. Правила

CLIPS підтримує евристичну й процедурну парадигму подання знань. Для подання знань у процедурній парадигмі CLIPS надає такі механізми, як глобальні змінні, функції й родові функції.

У цій темі ми розглянемо такий спосіб подання знань, як **правила**. Правила в CLIPS служать для подання евристик або так званих "емпіричних правил", які визначають набір дій, виконуваних при виникненні деякої ситуації. розробник експертної системи визначає набір правил, які разом працюють над рішенням деякої задачі.

Правила складаються з **передумов** і **нещастя**. Передумови називаються також "**ЯКЩО-частиною**" **правила**, **лівою частиною правила** або **LHS правила** (left-hand side of rule).

Наслідок називається **T-частиною правила**, **правою частиною правила** або **RHS правила** (right-hand side of rule).

Передумови правила являють собою набір умов (або умовних елементів), які повинні задовольнитися, для того щоб правило виконалося.

Передумови правил задовольняються залежно від наявності або відсутності деяких заданих фактів у списку фактів або деяких створених об'єктів, що є екземплярами класів, визначених користувачем.

Один з найпоширеніших типів умовних виражень в CLIPS — **зразки** (patterns).

Зразки складаються з набору обмежень, які використовуються для визначення того, чи задовольняє деякий факт або об'єкт умовному елементу. Інакше кажучи, зразок задає деяку маску для фактів або об'єктів.

Процес зіставлення зразків фактам або об'єктам називається **процесом зіставлення зразків** (pattern-matching).

CLIPS надає механізм, називаний **механізмом логічного висновку** (inference engine), що автоматично зіставляє зразки з поточним списком фактів і певних об'єктів у пошуках правил, які застосовні в цей момент.

Наслідок правила представляється набором деяких дій, які необхідно виконати, у випадку якщо правило застосовне до поточної ситуації.

Таким чином, дії, задані внаслідок правила, виконуються по команді механізму логічного висновку, якщо всі передумови правила задоволені.

У випадку якщо в цей момент застосовно більше одного правила, механізм логічного висновку використовує так названу **стратегію дозволу конфліктів** (conflict

resolution strategy), що визначає, яке саме правило буде виконано. Після цього CLIPS виконує дії, описані внаслідок обраного правила (які можуть вплинути на список застосовних правил), і приступає до вибору наступного правила. Цей процес триває доти, поки список застосовних правил не спорожніє.

Щоб краще зрозуміти сутність правил в CLIPS, їх можна представити у вигляді оператора **IF-THEN**, використовуюваного в процедурних мовах програмування, наприклад, таких як Ada або C. Однак умови вираження **IF-THEN** у процедурних мовах обчислюються тільки тоді, коли потік керування програми безпосередньо попадає на даний вираз шляхом послідовного перебору виразів й операторів, що становлять програму.

В CLIPS, на відміну від цього, механізм логічного висновку створює й постійно модифікує список правил, умови яких у цей момент задоволені. Ці правила запускаються на виконання механізмом логічного висновку. Із цієї сторони правила схожі на оброблювачі повідомлень, що є присутнім у таких мовах програмування, як, наприклад, Ada або Smalltalk.

Без правил не обійдеться жодна експертна система, так що правила й мова їхнього подання в експертній системі можна сміло назвати найважливішою частиною будь-якої експертної оболонки. Успіх експертної системи багато в чому визначається тим, наскільки вдалий спосіб подання знань у вигляді правил, і наскільки добре їм володіє розробник експертної системи. Вся дана глава присвячена правилам, їхньому синтаксису й способам побудови, їхньому функціонуванню й призначенню, а також прийомам їхнього застосування.

8.1. Створення правил. Конструктор defrule

Для додавання нових правил у базу знань CLIPS надає спеціальний конструктор defrule. У загальному виді синтаксис даного конструктора можна представити в такий спосіб:

Визначення 8.1. Синтаксис конструктора defrule (defrule)

```
<імені-правила>  
[<коментарі>]  
[<визначення-властивості-правила>]  
<передумови > ; ліва  
частина правила  
=>  
<наслідок> ; права  
частина правила
```

)

Ім'я правила повинне бути значенням типу symbol. Як ім'я правила не можна використати зарезервовані слова CLIPS, які були перераховані раніше. Повторне визначення існуючого правила приводить до видалення правила з тим же ім'ям, навіть якщо нове визначення містить помилки.

Коментарі є необов'язковими, і, як правило, описують призначення правила. Коментарі необхідно містити в лапки. Ці коментарі зберігаються й надалі можуть бути доступні при перегляді визначення правила.

Визначення правила може містити оголошення властивостей правила, яких треба безпосередньо після імені правила й коментарів. Більш докладно властивості правила будуть розглянуті нижче.

У довідковій системі й документації по CLIPS для позначення передумов правила найчастіше застосовується термін "LHS of rule", а для позначення наслідку "RHS of rule", тому надалі ми будемо використовувати аналогічну термінологію - ліва й права частина правила.

Ліва частина правила задається набором умовних елементів, що звичайно складається з умов, застосованих до деяких зразків. Заданий набір зразків використовується системою для зіставлення з наявними фактами й об'єктами.

Всі умови в лівій частині правила поєднуються за допомогою неявного логічного оператора and. Права частина правила містить список дій, виконуваних при активізації правила механізмом логічного висновку.

Для поділу правої й лівої частини правил використовується символ `=>`. Правило не має обмежень на кількість умовних елементів або дій. Єдиним обмеженням є вільна пам'ять вашого комп'ютера. Дії правила виконуються послідовно, але тоді й тільки тоді, коли всі умовні елементи в лівій частині цього правила задоволені.

Якщо в лівій частині правила не зазначений жоден умовний елемент, CLIPS автоматично підставляє умову-зразок **initial-fact** або **initial-object**. Таким чином, правило активізується щораз із появою в

базі знань факту initial-fact АБО Об'єкта initial-object.

Якщо в правій частині правила не визначене жодне дія, правило може бути активоване й виконане, але при цьому нічого не відбудеться.

Більш докладно синтаксис лівої й правої частини правил буде описаний далі в цій лекції, а поки, як демонстрація застосування правил, напишемо найпростішу CLIPS-програму, що за традицією поздоровкається з усім світом, відразу після свого народження. Ви напевно знайомі з текстом подібних програм для процедурних мов програмування, таких як, наприклад, С. Мовою CLIPS така програма буде виглядати в такий спосіб:

Приклад 8.1. Програма "Hello-World!"

```
(clear)
(defrule
Hello-World
"My FirstCLIPS Rule"
=>
(printout t crlf crlf)
(printout t *****
crlf)
(printout t "* HELLO WORLD!!! *"
crlf)
(printout t *****
crlf)
(printout t crlf crlf)
)
(reset) (run)
```

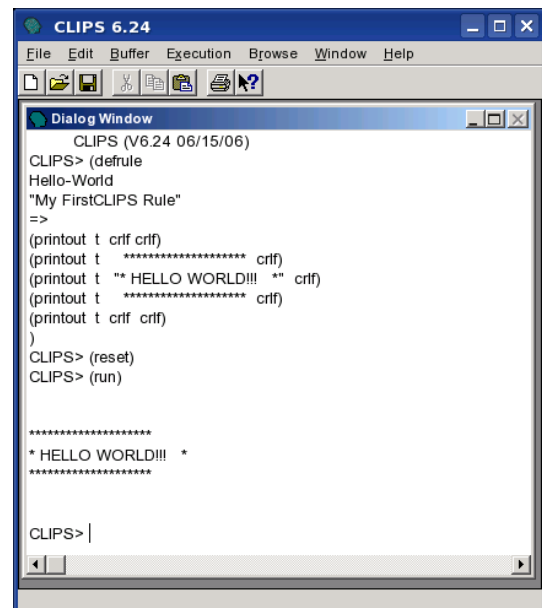


Рис. 8.1. Результат роботи програми "Hello-World!"

Тому що це перша наша програма мовою CLIPS, розберемо докладно всієї її дії й те, яким образом вони виконуються.

Функція **clear** повністю очищує систему, тобто видаляє всі правила, факти та інші об'єкти бази знань CLIPS, додані конструкторами, приводить систему в початковий стан, необхідний для кожної нової програми.

Потім, за допомогою конструктора `defrule` у систему додається нове правило з ім'ям `Hello-World` і відповідними коментарями. Ліва частина правила в цьому випадку відсутній, тому CLIPS автоматично формує передумови, що складаються з єдиного умовного вираження (`initial-fact`). Цей вираз є зразком найпростішого типу. При запуску програми на виконання механізм логічного висновку CLIPS буде шукати в списку фактів факт (`initial-fact`) і якщо він там буде знайдений — активізує правило. Права частина нашого правила складається з декількох викликів функції `printout`.

Зараз нам необхідно знати, що ця функція виводить текстовий вираз в один з потоків висновку. Параметр *t* задає стандартний потік висновку - екран. Він аналогічний, наприклад, стандартному потоку `cout` в C++. Виразення `clr` служить для переходу на новий рядок.

Функція `reset`, як уже згадувалося раніше, очищає список фактів і заносить у нього факт (`initial-fact`), що дуже важливо для нормального функціонування нашої програми.

І, нарешті, функція `run` запускає механізм логічного висновку й приводить нашу програму в дію.

Якщо описані вище дії були виконані правильно, то ви повинні побачити результат, аналогічний наведеному на мал. 8.1 — фразу "hello world!!!" у гарній рамочці із зірочок.

Щоб запустити нашу програму на виконання ще раз, досить викликати функції `reset` й `run`. Ці функції можна вводити із клавіатури, крім того, вони доступні в меню **Execution** і мають "гарячі" клавіші `<Ctrl>+<E>` й `<Ctrl>+<R>` відповідно.

CLIPS підтримує ряд функцій, команд і візуальних засобів, необхідних для ефективної роботи із правилами. Самі основні з них будуть розглянуті в даній лекції в міру необхідності.

Зараз же розглянемо візуальний інструмент, доступний користувачам Windows-версії середовища CLIPS — **Defrule Manager** (Менеджер правил). Для запуску менеджера правил у меню **Browse** виберіть пункт **Defrule Manager**. Зовнішній вигляд цього інструмента показаний на мал. 8.2.

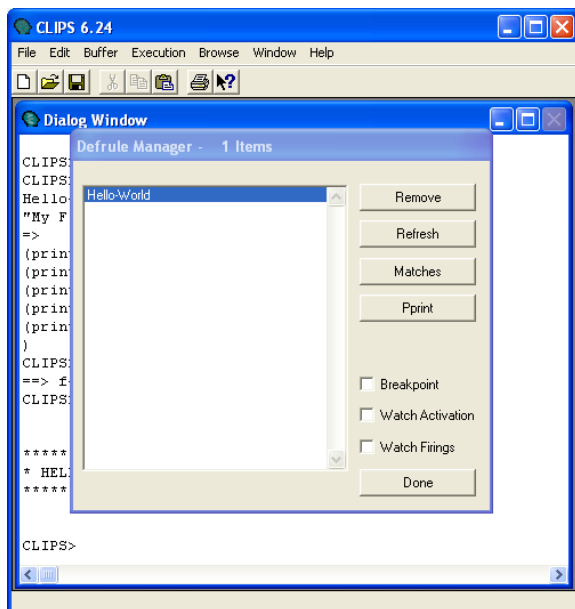


Рис. 8.2. Вікно менеджера правил

Менеджер відображає список правил, що є присутнім у системі в цей момент, і дозволяє виконувати над ними ряд операцій. Наприклад, за допомогою кнопки **Remove** можна видалити обране правило із системи, а за допомогою **Pprint** вивести у вікні CLIPS визначення

виділеного правила разом з уведеними коментарями. Загальна кількість правил відображається в заголовку вікна менеджера — **Defrule Manager — 1 Items**.

Як ви вже могли переконатися, розбираючи наведений вище приклад нескладної програми, досить важко створити навіть мінімально корисну програму мовою CLIPS, не уявляючи собі, що саме відбувається при виконанні вашої програми. При створенні експертних систем необхідно точно знати, як відбувається зіставлення зразків, заданих у лівій частині правила, яким саме образом вибирається правило для виконання й т.д.

8.2. Основний цикл виконання правил

Після того як у систему додані всі необхідні правила й приготовлені початкові списки фактів й об'єктів, CLIPS готовий виконувати правила.

У традиційних мовах програмування крапка входу, крапка зупинки й послідовність обчислень явно визначаються програмістом.

В CLIPS потік виконання програми зовсім не вимагає явного визначення. **Знання** (правила) і **дані** (факти й об'єкти) розділені, і механізм логічного висновку, надаваний CLIPS, застосовує дані до знань, формуючи **список застосовних правил**, після чого послідовно виконує їх. Цей процес називається **основним циклом виконання правил** (basic cycle of rule execution).

Розглянемо послідовність дій (кроків), виконуваних системою CLIPS у цьому циклі в момент виконання нашої програми:

1. Якщо були досягнуті межі виконання правил або не був установлений поточний фокус, виконання переривається. У противному випадку, для виконання вибирається перше правило модуля, на якому був установлений фокус. Якщо в поточному плані виконання немає вдоволених правил, то фокус переміщається по стеку фокусів і встановлюється на наступний модуль у списку. Якщо стік фокусів порожній, виконання припиняється. Інакше крок 1 виконується ще один раз.
2. Виконуються дії, описані в правій частині обраного правила. Використання функції `return` може міняти положення фокуса в стеці фокусів. Число запусків даного правила збільшується на одиницю, для визначення межі виконання правила.
3. У результаті виконання кроку 2 деякого правила можуть бути активовані або дезактивовані. Активовані правила (тобто правила, умови яких задовольняються в цей момент) містяться в план рішення задачі модуля, у якому вони визначені. Розміщення в плані визначається пріоритетом правила (*salience*) і поточною стратегією розв'язання конфліктів (ці поняття будуть описані

нижче). Дезактивовані правила віддаляються з поточного плану рішення задачі. Якщо для правила встановлений режим перегляду активацій, то користувач одержить відповідне інформаційне повідомлення при кожній активації або дезактивації правила (режим перегляду активацій можна встановити за допомогою діалогового вікна **Watch Options**. Для цього виберіть пункт **Watch** у меню **Execution** й установите прапорцець **Activations**).

4. Якщо встановлений режим динамічного пріоритету (dynamic salience), те для всіх правил з поточного плану рішення задачі обчислюються нові значення пріоритету. Після цього цикл повторюється із кроку 1.

8.3. Властивості правил

Для більше повного розуміння матеріалу, викладеного далі в цій лекції, необхідно розібратися з таким поняттям, як властивості правил. Властивості правил дозволяють задавати характеристики правил до опису лівої частини правила. Для завдання властивості правила використовується ключове слово `declare`. Одне правило може мати тільки одне визначення властивості, задане за допомогою `declare`.

Визначення 9.2. Синтаксис властивостей правил

```
<визначення-властивості-правила> ::=
(declare <властивості-правила>)
<властивості-правила> ::=
(salience <цілочисельне вираження>) :
(auto-focus TRUE :FALSE)
```

8.3.1. Властивість salience

Властивість правила `salience` дозволяє користувачеві призначати пріоритет для своїх правил. Пріоритет, що повідомляє, повинен бути виразом, що має цілочисельне значення з діапазону від -10 000 до +10 000.

Вираз, що представляє пріоритет правила, може використати глобальні змінні й функції. Однак намагайтеся не вказувати в цьому вираженні функцій, що мають побічну дію. У випадку якщо пріоритет правила явно не заданий, йому привласнюється значення за замовчуванням - 0.

Значення пріоритету може бути обчислене в одному із трьох випадків: при додаванні нового правила, при активації правила й на кожному кроці основного циклу виконання правил. Два останніх варіанти називаються **динамічним пріоритетом** (dynamic salience). За замовчуванням значення пріоритету обчислюється тільки під час додавання правила. Для зміни цієї установки можна використати команду `set-salience-evaluation`.

Крім того, користувачі Windows-версії середовища CLIPS можуть змінити це налаштування за допомогою

діалогового вікна **Execution Options**. Для цього виберіть пункт **Options** у меню **Execution**, у діалоговому вікні, що з'явилося, укажіть необхідний режим обчислення пріоритету за допомогою списку, що **розкривається**, **Salience Evaluation**, як показано на мал. 8.3.

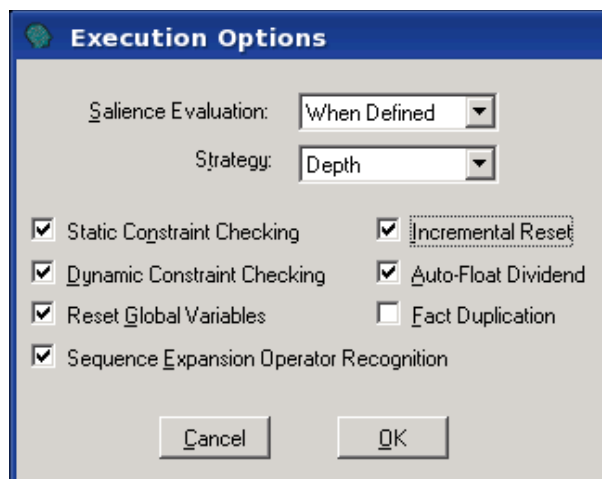


Рис. 8. 3. Установка способу обчислення пріоритетів правил

Кожний метод обчислення пріоритету містить у собі попередній (тобто якщо пріоритет обчислюється на кожному кроці основного циклу виконання правил, те він також обчислюється й при активації правила, а так само при його додаванні в систему).

8.3.2. Властивість auto-focus

Властивість **auto-focus** дозволяє автоматично виконуватися команді `focus` при кожній активації правила.

Якщо властивість `auto-focus` встановлена в значення `TRUE`, то команда `focus` у модулі, у якому визначене дане правило, автоматично виконується щораз при активації правила.

Якщо властивості `auto-focus` привласнене значення `FALSE`, то при активації правила не відбувається ніяких дій. За замовчуванням ця властивість встановлена в `FALSE`.

8.4. Стратегія дозволу конфліктів

План рішення задачі — це список всіх правил, що мають задоволені умови при якомусь стані списку фактів й об'єктів (і які ще не були виконані).

Кожен модуль має свій власний план рішення задачі. Виконання плану подібно стеку (верхнє правило плану завжди буде виконано першим).

Коли активується нове правило, воно

розміщається в плані рішення задачі керуючись наступними факторами:

- Тільки що активоване правило міститься вище всіх правил з меншим пріоритетом і нижче всіх правил з більшим пріоритетом.
- Серед правил з однаковим пріоритетом використовується поточна стратегія дозволу конфліктів для визначення розміщення серед інших правил з однаковим пріоритетом.
- Якщо правило активоване разом з декількома іншими правилами, додаванням або виключенням деякого факту й за допомогою кроків 1 й 2 не можна визначити порядок правила в плані рішення задачі, то правило довільним образом упорядковуються разом з іншими правилами, які були активовані. Помітьте, що в цьому випадку порядок, у якому правила були додані в систему, робить довільний ефект на дозволи конфлікту (який найвищою мірою залежить від поточної реалізації правил). Намагайтеся не використати довільне упорядкування правил при рішенні задач, у яких потрібні точні результати або пояснення отриманих рішень.

CLIPS підтримує сім різних стратегій дозволу конфліктів: *стратегія глибини* (depth strategy), *стратегія ширини* (breadth strategy), *стратегія спрощення* (simplicity strategy), *стратегія ускладнення* (complexity strategy), *LEX* (LEX strategy), *MEA* (MEA strategy) і *випадкова стратегія* (random strategy).

За замовчуванням в CLIPS встановлена стратегія глибини. Поточна стратегія може бути встановлена командою set-strategy (яка з поточний план рішення задачі, базуючись на новій стратегії). Крім того, користувачі Windows-версії середовища CLIPS можуть указати необхідну стратегію пошуку за допомогою діалогового вікна **Execution Options** (див. мал. 9.3). Для цього виберіть пункт **Options** у меню **Execution**, у діалоговому вікні, що з'явилася, виберіть необхідну стратегію за допомогою списку, що **розкривається, Strategy**.

8.4.1. Стратегія глибини

Тільки що активоване правило міститься вище всіх правил з таким же пріоритетом.

Наприклад, припустимо, що факт-а активував правило-1 і правило-2 і факт-б активував правило-3 і правило-4, тоді, якщо факт-а доданий перед фактом-б, у плані рішення задачі правило-3 і правило-4 будуть розташовуватися вище, ніж правило-1 і правило-2. Однак позиція правила-1 відносно правила-2 і правила-3 відносно правила-4 буде довільною.

8.4.2. Стратегія широчини

Тільки що активоване правило міститься нижче всіх правил з таким же пріоритетом.

Наприклад, допустимо, що факт-а активував правило-1 і правило-2 і факт-б активував правило-3 і правило-4, тоді, якщо факт-а доданий перед в, у плані рішення задачі правило-1 і правило-2 будуть розташовуватися вище, ніж правило-3 і правило-4. Однак позиція правила-1 відносно правила-2 і правила-3 відносно правила-4 буде довільною.

8.4.3. Стратегія спрощення

Між всіма правилами з однаковим пріоритетом тільки що активовані правила розміщуються вище всіх активованих правил з рівною або більшою **визначеністю** (specificity). Визначеність правила обчислюється за кількістю зіставлень, які потрібно зробити в лівій частині правила. Кожне зіставлення з константою або заздалегідь пов'язаною з фактом змінної додає до визначеності одиницю.

Кожний виклик функції в лівій частині правила, що є частиною умовних елементів :, = або test, також додає до визначеності одиницю. Логічні функції and, or й not не збільшують визначеність правила, але їхні аргументи можуть зробити це. Виклики функцій, зроблені усередині функцій, не збільшують визначеність правила.

Наприклад, впливаюче правило має визначеність, рівну 5.

Приклад 8.2. Обчислення визначеності правила

```
(defrule example
  (item ?x ?y ?x)
  (test (and (numberp ?
x) (> ?x (+ 10 ?
y)) (< ?x 100)))
=>
```

І порівняння заздалегідь зв'язаною змінною ?x з константою, і виклики функцій numberp, < й > додають одиницю до визначеності правила. У підсумку одержуємо визначеність, рівну 5. Виклики функцій and й + не збільшують визначеність правила.

8.4.4. Стратегія ускладнення

Між правилами з однаковим пріоритетом, тільки що активовані правила розміщуються вище всіх активованих правил з рівною або меншою визначеністю.

8.4.5. Стратегія LEX

Між правилами з однаковим пріоритетом тільки що активовані правила розміщуються з використанням однойменної стратегії, уперше використаної в системі

OPS5. Для визначення місця активованого правила в плані рішення задачі використовується "новизна" зразка, що активував правило. CLIPS маркірує кожен факт або об'єкт тимчасовим тегом для відображення відносної новизни кожного факту або об'єкта в системі.

Зразки, асоційовані з кожною активацією правила, сортуються по убутуванню тегів для визначення місця розташування правила. Активація правила, виконана більше новими зразками, розташовується перед активацією, здійсненою більше пізніми зразками. Для визначення порядку розміщення двох активацій правил, поодиночі рівняються відсортовані тимчасові теги для цих двох активацій, починаючи з найбільшого тимчасового тегу. Порівняння триває доти, поки не залишиться одна активація з найбільшим тимчасовим тегом. Ця активація розміщується вище всіх інших у плані рішення задачі.

Якщо активація деякого правила виконана більшим числом зразків, чим активація іншого правила й всі порівнювані тимчасові теги однакові, то активація з більшим числом тимчасових тегів поміщає перед активацією з меншим. Якщо дві активації мають однакову кількість тимчасових тегів й їхніх значень рівні, то правило з більшою визначеністю міститься перед активацією з меншою. На відміну від системи OPS5, умовний елемент `not` в CLIPS має псевдочасовий тег, що також використовується в даній стратегії дозволу конфліктів. Часовий тег умовного елемента `not` завжди менше, ніж часовий тег зразку.

Як приклад розглянемо наступні шість активацій правил, наведені в LEX-порядку (кома наприкінці рядка активації означає наявність логічного елемента `not`). Урахуйте, що тимчасові теги фактів не обов'язково дорівнюють індексу, але якщо індекс факту більше, то більше і його часовий тег. Для даного прикладу приймемо, що тимчасові теги дорівнюють індексам.

Приклад 9.3. Правила, відсортовані стратегією LEX

```
rule-6:      f-1, f-4
rule-5:      f-1, f-2, f-3,
rule-1:      f-1, f-2, f-3
rule-2:      f-3, f-1
rule-4:      f-1, f-2
rule-3:      f-2, f-1
```

У прикладі 8.4 показані ті ж активації з індексами фактів у тім порядку, у якому вони вирівнюються стратегією LEX.

Приклад 9.4. Порядок порівняння стратегією LEX

```
rule-6:      f-4, f-1
rule-5:      f-3, f-2, f-1,
rule-1:      f-3, f-2, f-1
```

```
rule-2:      f-3, f-1
rule-4:      f-2, f-1,
rule-3:      f-2, f-1
```

8.4.6. Стратегія MEA

Між правилами з однаковим пріоритетом тільки що активовані правила розміщуються з використанням однойменної стратегії, уперше використаної в системі OPS5.

Основна відмінність стратегії MEA від LEX полчає у тому, що в стратегії MEA не виробляється сортування зразків, що активували правило. Рівняються тільки тимчасові теги перших зразків двох активацій. Активація з більшим тегом міститься в план рішення задачі перед активацією з меншим. Якщо обидві активації мають однакові тимчасові теги, асоційовані з першим зразком, то для визначення розміщення активації в плані рішення задачі використовується стратегія LEX. Так само, як й у стратегії LEX, умовний елемент `not` має псевдочасовий тег.

Як приклад розглянемо наступні шість активацій, наведені в Mea-порядку (кома на кінці активації означає наявність логічного елемента `not`).

Приклад 9.5. Правила, відсортовані стратегією MEA

```
rule-2:      f-3, f-1
rule-3:      f-2, f-1
rule-6:      f-1, f-4
rule-5:      f-1, f-2, f-3,
rule-1:      f-1, f-2, f-3
rule-4:      f-1, f-2,
```

8.4.7. Випадкова стратегія

Кожній активації призначається випадкове число, що використовується для визначення місця розташування серед активацій з однаковим пріоритетом. Це випадкове число зберігається при зміні стратегій, таким чином, той же порядок відтворюється при наступній установці випадкової стратегії (серед активацій у плані рішення задачі, коли стратегія замінена на вихідну).

8.5. Синтаксис LHS правила

Цей розділ описує синтаксис, використовуваний у лівій частині правил. Ліва частина правил містить список *умовних елементів* (conditional elements або CEs), які повинні задовольнятися, для того щоб правило було поміщено в план рішення задачі. Існує вісім типів умовних елементів, використовуваних у лівій частині правил: *CEs-зразки*, *test CEs*, *and CEs*, *or CEs*, *not CEs*, *exists CEs*, *forall CEs* й *logical CEs*.

Зразки - найбільше часто використовуваний умовний елемент. Він містить обмеження, які служать для

визначення, чи задовольняє який-небудь елемент даних (факт або об'єкт) зразку.

Умова `test` використовується для оцінки вираження, як частини процесу зіставлення образів.

Умова `and` застосовується для визначення групи умов, кожне з якої повинне бути задоволене.

Умова `or` - для визначення однієї умови з деякої групи, що повинне бути задоволене.

Умова `not` - для визначення умови, що не повинне бути задоволене.

Умова `exists` - для перевірки наявності, принаймні одного, збігу факту (або об'єкта) з деяким заданим зразком.

І нарешті, умова `logical` дозволяє виконати додавання фактів і створення об'єктів у правій частині правила, пов'язаних з фактами й об'єктами, що збіглися із заданим зразком у лівій часті правила (підтримка вірогідності фактів у базі знань).

Синтаксис умовного елемента можна формалізувати в такий спосіб:

Визначення 8.3. Синтаксис умовного елемента

```
<умовн-елемент> ::= <pattern-CE> |  
<assigned-pattern-CE> | <not-CE> |  
<and-CE> | <or-CE> | <logical-CE> |  
<test-CE> | <exists-CE> | <forall-CE>
```

8.5.1. Зразок (pattern PC)

Цей умовний елемент складається зі списку обмежень полів, групових символів (wildcards) і змінних, які використовуються для пошуку множини фактів або об'єктів, які відповідають заданому зразку. Таким чином, зразок як би визначає маску, який повинні відповідати дані. Такий умовний елемент задовольняється будь-яким фактом або об'єктом, що відповідають заданим обмеженням.

Обмеження полів — це набір обмежень, які використовуються для перевірки простих полів або слотів об'єктів. Обмеження полів можуть складатися тільки з одного символічного обмеження, однак, кілька обмежень можна з'єднувати разом. На додаток до символічних обмежень, CLIPS підтримує три інших типи обмежень: *об'єднуючі обмеження*, *предикатні обмеження* й *обмеження, що повертають значення*.

Групові символи використовуються при зіставленні зразків у ситуації, коли просте поле або група полів можуть приймати будь-які значення.

Змінні застосовуються для зберігання значення поля, що може бути згодом використане в лівій частині правила для іншого умовного елемента або в правій частині, як аргумент дії.

Перше поле будь-якого зразка обов'язково повинне бути значенням типу `symbol` і не може приймати значення інших типів. CLIPS використає перше поле для визначення: чи є даний зразок упорядкованим фактом, шаблоном або об'єктом. Ключове слово `object` зарезервоване для створення зразків, призначених для зіставлення з об'єктами. Будь-яке інше значення типу `symbol` повинне відповідати імені шаблону, створеного за допомогою конструктора `deftemplate` або неявно створеного шаблону. Для завдання імен слотів також повинні використовуватися значення типу `symbol`.

У слотах простих полів зразків, призначених для об'єктів і шаблонів, може втримуватися тільки одне обмеження поля, і не можуть бути присутнім групові символи або змінні. У складених слотах може втримуватися будь-яка кількість обмежень поля.

Далі будуть показані синтаксис і приклади використання зразків.

Для забезпечення наочності прикладів у наступних лекціях будуть використовуватися факти й шаблони, наведені в прикладі 8.6.

Приклад 8.6. Необхідні для подальшої роботи шаблони й факти

```
(defacts data-facts  
  (data 1.0 blue "red")  
  (data 1 blue)  
  (data 1 blue red)  
  (data 1 blue RED)  
  (data 1 blue red 9.9))  
  
(deftemplate person  
  (slot name)  
  (slot age)  
  (multislot friends))  
  
(defacts people  
  (person (name Joe) (age 20))  
  (person (name Bob) (age 20))  
  (person (name Joe) (age 34))  
  (person (name Sue) (age 34))  
  (person (name Sue) (age 20)))
```

8.5.1.1. Символьні обмеження

Основні обмеження, що використовуються в зразках, — це обмеження, що визначають точну відповідність між полями факту й зразком. Ці обмеження називаються *символьними*. Символьне обмеження повністю складається з констант, таких як речовинні й цілі числа, значення типу `symbol`, рядки або імена об'єктів. Вони не можуть містити групових символів або змінних. Всі символічні обмеження при зіставленні зразків повинні точно збігатися по всіх зазначених полях, інакше факт не буде вважатися придатним доданому зразку.

Умовний елемент, що представляє собою зразок

для неупорядкованого факту, у якому присутні тільки символні обмеження, має наступний синтаксис:

Визначення 8.4. Синтаксис символних обмежень для неупорядкованого факту

`(<обмеження-1> ... <обмеження-n>)`

Умовний елемент, що представляє собою зразок для шаблону, у якому присутні тільки символні обмеження, виглядає так:

Визначення 8.5. Синтаксис символних обмежень для шаблону

`(<ім'я-шаблону>
(<ім'я-слота-1> <обмеження-1>)
...
(<ім'я-слота-n> <обмеження-n>))`

Розглянемо приклад правил, що використовують як зразок — зразок фактів (як упорядкованих, так і шаблонів) із символними обмеженнями. Для нормальної роботи цього приклада необхідно ввести в CLIPS всі конструктори визначених фактів і шаблонів, представлені в цій лекції.

Після цього варто виконати команду `reset` для ініціалізації списку фактів. Для перевірки правильності виконаних операцій відкрийте вікно **Facts**. Якщо всі описані дії були виконані без помилок, ви повинні побачити результат, наведений на рис. 8.4.

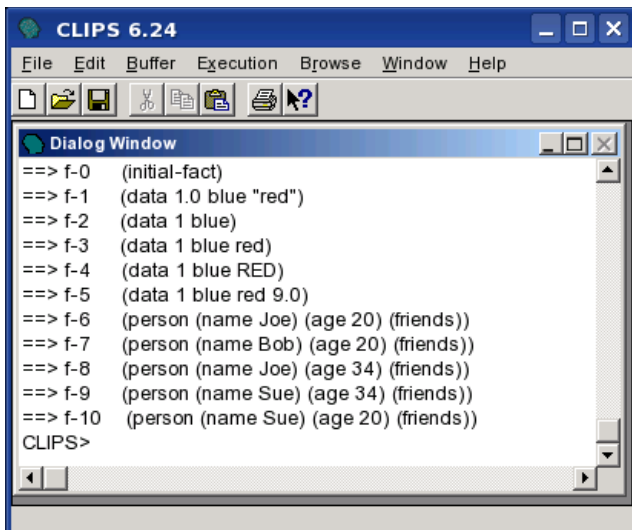


Рис. 8.4. Список необхідних фактів

Для нормальної роботи прикладів не забувайте виконувати команду `reset` перед кожним запуском правил. Уведіть в CLIPS визначення наступних правил:

Приклад 8.7. Правила із символними обмеженнями

```
(defrule Find-data  
  (data 1 blue red)  
  =>
```

```
(printout t crlf "Found data  
  (data 1 blue red)" crlf))  
(defrule Find-Bob-20  
  (person (name Bob) (age 20))  
  =>  
  (printout t crlf "Found Bob-20  
  (person (name Bob) (age 20))"  
    crlf))  
(defrule Find-Bob-30  
  (person (name Bob) (age 30))  
  =>  
  (printout t crlf "Found Bob-30  
  (person (name Bob) (age 30))"  
    crlf))
```

Виконайте команди `reset` й `run`. Ви повинні одержати результат, наведений на мал. 8.5.

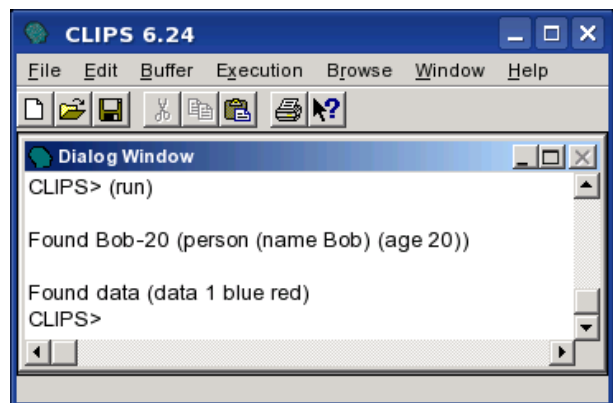


Рис. 8.5. Виконання правил із символними обмеженнями

Як ми бачимо, були активовані й виконані два правила: `Find-data` й `Find-Bob-20`. Це відбулося тому, що зразки, задані в лівій частині цих правил, знайшли в списку фактів дані, повністю відповідаючи заданим символним обмеженням.

8.5.1.2. Групові символи для простих і складових полів

В CLIPS є два різних групових символи, які використовуються для зіставлення полів у зразках. CLIPS інтерпретує ці групові символи як місце для підстановки деяких частин даних, що задовольняють зразкам. Груповий символ для простого поля записується за допомогою знака `?`, що відповідає одному будь-якому значенню, збереженому в заданому полі. Груповий символ складеного поля записується за допомогою знака `$?` і відповідає, можливо, порожньої послідовності полів, збереженої в складеному полі. Групові символи для простих і складових полів можуть комбінуватися в будь-якій послідовності. Не можна використати груповий символ складеного поля для простих полів. За замовчуванням не заданий у зразку простий слот шаблону або об'єкта зіставляється з неявно заданим груповим символом для простого поля. Аналогічно не заданий у зразку складений слот зіставляється з неявно заданим груповим символом для складеного поля.

Умовний елемент, що представляє собою зразок для неупорядкованого факту, у якому присутні тільки символні обмеження й групові символи, буде мати такий вигляд:

Визначення 8.6. Синтаксис обмежень для неупорядкованого факту
(<обмеження-1> ... <обмеження-n>)
<обмеження> ::= <символьн-
обмеження> : ? : \$?

Відповідно для шаблону зразок прийме вид:

Визначення 8.7. Синтаксис обмежень для шаблону
(<ім'я шаблону>
(<ім'я-слота-1> <обмеження-1>
...
(<ім'я-слота-n> <обмеження-n>))

Як приклад можна привести наступне правило:

Приклад 8.8. Правило Find-data

```
(defrule Find-data
  (data ? blue red $?) =>)
```

У нашому списку фактів присутні два факти, що підходять заданому шаблону й здатні активувати дане правило:

Приклад 8.9. Факти, що активують правило Find-data

```
(data 1 blue red)
(data 1 blue red 9.9))
```

Розглянемо ще одне правило:

Приклад 8.10. Правило match-all-persons

```
(defrule match-all-persons
  (person)
  =>)
```

Оскільки person є шаблоном, а в зразку даного правила не визначений жоден слот шаблону, CLIPS автоматично поставить у відповідність кожному простому слоту груповий символ для простого поля, а складеному слоту - символ для складеного. Таким чином, правило перетвориться в наступне:

Приклад 9.11. Перетворене правило match-all-persons

```
(defrule match-all-persons
  (person
   (name ?)
   (age ?)
   (friends $?))
  =>)
```

Це правило буде активувати всі факти шаблону person.

Групові символи для складеного поля можна комбінувати із символними обмеженнями, що приводить до одержання могутніших можливостей зіставлення зразків. Зразок, що зіставляється з усіма фактами, що мають значення yellow у будь-якому полі (включаючи перший), може бути записаний так:

Приклад 9.12. Зразок зі значенням yellow у будь-якому полі

```
(data $? yellow $?)
```

От кілька фактів, що відповідають цьому зразку:

Приклад 9.13. Факти зі значенням yellow у будь-якому полі

```
(data yellow blue red green)
(data yellow red)
(data red yellow)
(data yellow)
(data yellow data yellow)
```

Останній факт буде відповідати зразку двічі, тому що yellow є присутнім у ньому двічі. Використання групового символу для складеного поля дозволяє створювати набагато більше загальні зразки, чим ті, які можна сформулювати за допомогою групових символів для простого поля. Однак подібна спільність приводить до того, що процес зіставлення зразків, що використовують групові символи, іноді займає набагато більше часу, чим аналогічний процес зі зразками, що використовують тільки групові символи для простих полів.

8.5.1.3. Змінні, пов'язані із простими й складеними полями

Групові символи замінюють будь-які поля зразка й можуть приймати які завгодно значення цих полів. Значення поля може бути пов'язане зі змінними для наступного зіставлення, відображення й інших дій. Це виконується за допомогою застосування імені змінної наступної безпосередньо після групового символу.

Таким чином, синтаксис обмеження, застосовуваного в зразку, прийме наступний вид:

Визначення 8.8. Синтаксис обмежень

```
<обмеження> ::= <символьн-обмеження> >
|
? :
$? :
<змінна-простого-поля> |
<змінна-складеного-поля>
<змінна-простого-поля> ::= ?
<ім'я-змінної>
<змінна-складеного-поля> ::= $?
<ім'я-змінної>
```

Ім'я змінної повинне бути значенням типу

symbol й обов'язково починається з букви. В імені змінної не дозволяється використати лапки, тобто рядок не може використовуватися як ім'я змінної або її частина.

Правила зіставлення зразків при використанні змінних в обмеженнях зразка аналогічні правилам, що використовуються для групових символів. У момент першої появи імені змінної вона поводить себе так само, як і відповідний груповий символ. У цей момент CLIPS зв'язує значення поля із заданою змінною. Цей зв'язок буде діяти тільки в рамках правила, у якому вона виникла. Кожне правило має свій власний список імен змінних зі значеннями, пов'язаними з ними, ці змінні локальні для правил.

Зв'язані змінні можуть бути використані в зовнішніх функціях. Символ \$ має особливе значення в лівій частині правил - цей оператор відображає, що деяка, можливо порожня, послідовність полів вимагає зіставлення. У правій частині правила символ \$ ставиться перед змінною для позначення того, що перед використанням змінної як аргумент функції необхідно розкрити послідовність полів, що втримуються в змінній. Таким чином, при використанні змінних як параметри функцій (як у лівій, так і правій частині правил) перед ім'ям змінне, утримуюче значення складеного поля, не повинен стояти символ \$ (за винятком випадків, коли потрібно розкрити послідовність полів). При використанні змінне, утримуюче значення складеного поля, в інших випадках, перед її ім'ям повинен стояти символ \$. Не можна застосовувати змінну складеного поля при операціях із простим полем зразка шаблону або об'єкта.

Як приклад уведіть у середовище CLIPS наступне правило:

Приклад 8.14. Правило Find-data

```
(defrule Find-data
  (data ? blue ?x $?y) =>
  (printout t "Found data (data ?
  blue " ?x " " ?y )" crlf))
```

Виконайте команди reset й run. Якщо правило було уведено в систему без помилок, то на екрані з'явиться наступний результат:

Приклад 8.15. Результат роботи правила Find-data

```
Found data (data ? blue red (6.9))
Found data (data ? blue RED ())
Found data (data ? blue red ())
Found data (data ? blue red ())
```

Зразку, заданому в правилі, задовольняють чотири факти з індексами 1, 3, 4, 5. У результаті активації правило виводить на екран властивості фактів, що активували правило.

Розглянемо наступне правило:

Приклад 8.16. Модифіковане правило Find-data

```
(defrule Find-data
  (data ?x $?y ?z) =>
  (printout t "x=" ?x " y=" ?y
    " z=" ?z crlf))
```

Заданому зразку задовольняють всі факти data, але зверніть увагу, яким образом зв'язуються значення зі змінної в у різних випадках:

Приклад 8.17. Результат роботи модифікованого правила Find-data

```
x=1.0 y=(blue) z=red
x=1 b=() z=blue
x=1 y=(blue) z=red
x=1 y=(blue) z=RED
x=1 y=(blue red) z=6.9
```

Після того як відбулося зв'язування змінної зі значенням, всі посилання на цю змінну повертають значення, з яким змінна була зв'язана. Це дійсно як для змінних, пов'язаних зі складеними полями, так і для змінних, пов'язаних із простими полями. Крім того, припустимі посилання між зразками в одному правилі.

Приклад 9.18. Правило Find-2-Coeval-Person

```
(defrule Find-2-Coeval-Person
  (person (name ?x) (age ?z))
  (person (name ?y) (age &z))
  =>
  (printout t "name=" ?x "
  name=" ?y " age=" ?z crlf))
```

Наведене вище правило Find-2-Coeval-person виведе на екран усілякі пари імен людей (всі перестановки) однакового віку. Як навчити це правило не виводити еквівалентні за змістом або безглузді пари однакових імен (Bob-Bob), ми побачимо далі.

8.5.1.4. Єднальні обмеження

CLIPS надає 3 єднальні обмеження, призначених для об'єднання окремих обмежень і змінних у єдине ціле: & (логічне И), | (логічне АБО) і ~ (логічне НЕ). Обмеження & задовольняється, якщо два сусідніх обмеження задовольняються. Обмеження | задовольняється, якщо кожне із двох сусідніх обмежень задовольняється. Обмеження ~ задовольняється, якщо наступне за ним обмеження не задовольняється. Єднальні обмеження можуть комбінуватися майже довільним образом й у будь-якій кількості. Обмеження ~ має найвищий пріоритет, далі впливають & й |. У випадку однакового пріоритету обмеження обчислюється ліворуч праворуч. Існує одне виключення із правил

пріоритету, що застосовується при зв'язуванні змінних. Якщо перше обмеження - це змінна й за нею треба &, то змінна є окремим обмеженням. Обмеження `?x&red|blue` обчислюється як `?x& (red|blue)`, у той час як за правилами пріоритету воно повинне було обчислюватися як `(?x&red) | blue`.

Зв'язані обмеження мають наступний синтаксис:

Визначення 8.9. Синтаксис єднальних обмежень

```
<елемент-1>& <елемент-2> ... & елемент
-n>
<елемент-1>| <елемент-2> ... | елемент
-n>
~ <елемент>
```

Тут <елемент> повинен бути змінною, пов'язаною із простим або складовим полем, обмеженням або зв'язаним обмеженням.

Таким чином, визначення обмежень, наведені в попередніх розділах, можна розширити так:

Визначення 8.10. Синтаксис обмежень

```
<обмеження> ::= ? :
                $? :
                <зв'язане-обмеження>
<зв'язане-обмеження> ::= <просто-
обмеження> |
                <просто-обмеження>&<зв'язане-
обмеження> |
                <просто-обмеження>|<зв'язане-
обмеження> |
<просто-обмеження> ::= <елемент>| ~
<елемент>
<елемент> ::= <константа>|
                <проста-змінна>|
                <складова-змінна>
```

Обмеження & звичайно служить тільки для об'єднання з іншими обмеженнями або зв'язування змінних. Помітьте, що єднальні обмеження можуть використати зв'язані змінні й у той же час самі робити зв'язування змінної зі значенням якогось поля. Якщо ім'я змінної зустрілося в перший раз, то для обмеження будуть використовуватися інші члени умовного елемента, а змінна буде пов'язана з відповідним значенням поля. Якщо змінна вже була зв'язана, то її значення працює як додаткове обмеження для даного поля.

Як приклад приведемо поліпшений варіант правила Find-2-coeval-Person з попереднього розділу.

Приклад 8.19. Поліпшене правило Find-2-Coeval-Person

```
(defrule Find-2-Coeval-Person
  (person (name ?x) (age ?z))
  (person (name ?y&~?x) (age
&z))
```

```
=>
(printout t "name=" ?x "
name=" ?y " age=" ?z crlf))
```

Обмеження `?y&~?x` забороняє виводити безглузді пари однакових імен (Bob-Bob). Однак дане правило усе ще виводить еквівалентні за змістом пари імен (наприклад, Bob-Sue й Sue-Bob).

8.5.1.5. Предикатні обмеження

Іноді необхідно обмежити поле, ґрунтуючись на істинності деякого логічного вираження. CLIPS дозволяє використати предикатні обмеження. Предикатні обмеження дозволяють викликати предикатні функції (функції, які повертають значення FALSE при не відповідності умовам і не-FALSE, якщо значення задовольняє умовам) протягом процесу зіставлення зразків. Якщо предикатна функція повертає значення не-FALSE, обмеження задовольняється. Якщо предикатна функція повертає значення FALSE, то обмеження не задовольняється. Предикатні обмеження записуються за допомогою двокрапки й наступного за ним виклику відповідної предикатної функції. Звичайно предикатні обмеження використовуються спільно з єднальними обмеженнями й при зв'язуванні змінних (тобто якщо ви маєте змінну, котру потрібно зв'язати з деяким полем і хочете одночасно неї протестувати, об'єднаєте її із предикатним обмеженням).

Предикатні обмеження мають наступний синтаксис:

Визначення 8.11. Синтаксис предикатного обмеження

```
:<виклик-функції>
```

Таким чином, визначення поняття "елемент", наведене в попередньому розділі, можна розширити в такий спосіб:

Визначення 8.12. Синтаксис поняття "елемент"

```
<елемент> ::= <константа>|
                <проста-змінна>|
                <складова-змінна>|
                :<викликів-функції>
```

CLIPS надає кілька готових предикатних функцій. Крім цього, користувач також може створювати свої власні предикатні функції.

Приклад 9.20. Ще один варіант правила Find-data

```
(defrule Find-data
  (data ?x&: (floatp ?x)&:{> ?x 0)
  $?y ?z&:(stringp ?z) )
=>
  (printout t "x=" ?x " y=" ?y " z="
?z crlf ) )
```

Вище наведений ще один варіант правила Find-data. У цьому випадку шукається факт неявно

створеного шаблона data, перше поле якого - речовинне число більше нуля, а останнє - рядок. У нашому списку фактів такому правилу задовольняє тільки факт із індексом 1 -

```
(data 1.0 blue "red") .
```

8.5.1.6. Обмеження, що повертають значення

В обмеженнях можливе використання значень, повернутих деякими функціями (у тому числі й зовнішніми). Виклик функції записується за допомогою знака = і зазначеної за ним функцією.

Функція порівняння також використовує знак =. Різниця між ними може бути визначена по контексту.

Значення, що повертає, повинне бути одним із простих типів даних CLIPS. Це значення, повернуте функцією, поєднується зі зразком так, ніби воно було символьним обмеженням. Помітьте, що функція обчислюється при кожному зіставленні зразків, а не один раз при визначенні правила.

Обмеження, що повертають значення, мають наступний синтаксис:

Визначення 9.13. Синтаксис обмеження, що повертає значення

```
=<виклик-функції>
```

Визначення поняття "елемент", наведені в попередньому розділі, приймуть такий вид:

Визначення 9.14. Синтаксис поняття "елемент"

```
<елемент> ::= <константа> |
```

```
<проста-змінна> |  
<складова-змінна> |  
:<виклик-функції>  
=<виклик-функції>
```

Правило із прикладу 9.21 виводить на екран такі факти data, у яких значення другого поля у два рази більше, ніж значення першого. У нашому випадку це факти (data 1 2) та (data 2 4).

Приклад 9.21. Використання обмеження, що повертає значення

```
(assert (data 1 2)  
        (data 2 3)  
        (data 2 4))  
(defrule Find-data  
  (data ?x ?y(&(* 2 ?x)))  
  =>  
  (printout t "x=" ?x "  
y=" ?y crlf))
```

8.5.1.7. Зіставлення зразків з об'єктами

У всіх наведених вище прикладах зразки зіставлялися з фактами зі списку фактів. Крім

цього, зразки можна зіставляти з екземплярами об'єктів - екземплярів, певних користувачем класів мовою COOL. Такі зразки називаються зразками об'єктів. Зразки можуть зіставлятися з об'єктами, специфікація яких визначена до створення зразка і які перебувають у границях видимості поточного модуля. Любою клас, що має об'єкти, що відповідають зразку, не може бути вилучений або змінений, поки не буде вилучений зразок. Навіть якщо правило вилучене за допомогою дій, виконуваних у власній правій частині, клас, пов'язаний зі зразком, не може бути змінений доти, поки права частина правила не закінчить роботу.

При створенні або видаленні об'єкта всі зразки, що підходять цьому об'єкту, оновлюються. Однак у випадку зміни слоту об'єкта оновлюються тільки ті зразки, які явно зіставляються по цьому слоті. Таким зразком можна використати логічні залежності для обробки змін деяких слотів.

Зміна неактивних слотів або об'єктів неактивних класів не робить ніякого впливу на правила.

Визначення 9.15. Синтаксис зразків об'єктів

```
<зразок об'єкта> ::= (object  
<атрибута-обмеження>  
<атрибута-обмеження> ::= (is-a  
<обмеження> |  
                           (name <обмеження> |  
                             (slot <обмеження> )
```

Обмеження is-a (ε) використовується для визначення обмежень класу, таких як "чи є цей об'єкт екземпляром заданого класу?".

Обмеження is-a також визначає, чи є об'єкт екземпляром класу, що є спадкоємцем класу, заданого в обмеженні, у випадку якщо це не буде явно заборонено зразком.

Обмеження name використовується для визначення конкретного об'єкта із заданим ім'ям. Ім'я, задане в даному обмеженні, повинне бути значенням типу instance-name, а не значенням типу symbol, як звичайно.

Обмеження для складових полів (такі як \$?) не можуть використатися з обмеженнями is-a й name. Ці обмеження застосовуються в роботі зі слотами об'єктів так само, як і при роботі зі слотами шаблонів. Як й у випадку зразків для шаблонів, імена слотів для зразка об'єкта повинні бути значеннями типу symbol.

8.5.2. Автоматичне додавання і перегрупування умовних елементів

У деяких ситуаціях CLIPS автоматично додає додаткові зразки до лівої частини правил (звичайно для поліпшення алгоритму зіставлення зразків, використовуваного системою CLIPS). Існує два зразки,

застосовуваних CLIPS за замовчуванням: зразок факту initial-object і зразок об'єкта initial-object.

Нижче приводиться визначення цих даних:

Визначення 9.24. Синтаксис визначеного факту й об'єкта

```
(initial-fact)
(object (is-a INITIAL-OBJECT) (name [initial-object]))
```

8.5.2.1. Безумовні правила

Якщо правило не містить умовних елементів у своїй лівій частині, то до передумов правила автоматично додається зразок initial-fact (конфігурацію CLIPS можна настроїти таким чином, щоб замість зразка факту додавався зразок об'єкта initial-object). Наприклад, правило, що впливає із прикладу 9.46, буде перетворено так, як показано в прикладі 9.47:

Приклад 9.46. Правило без умов

```
(defrule example
=> )
```

Приклад 9.47. Перетворене правило без умов

```
(defrule example
(initial-fact)
=>)
```

8.5.2.2. Використання елементів test та not перед and

Умовні елементи test й not, що коштують перед and, додають зразок initial-fact або initial-object безпосередньо перед собою. Зразок initial-fact додається, якщо в першому умовному елементі використовується зразок факту. Зразок initial-object додається, якщо в першому умовному елементі використовується зразок об'єкта. Якщо в першому умовному елементі немає зразків, то тип зразка, що додає, визначається по наступному умовному елементі таким же методом. Якщо у всьому поточному умовному вираженні немає зразків, то система використає визначений факт initial-fact (хоча конфігурацію CLIPS можна настроїти таким чином, щоб замість зразка initial-fact додавався зразок initial-object).

Наприклад правила, що впливають із прикладу 9.48, будуть змінені так, як у прикладі 9.49.

Приклад 9.48. Правила з умовами test й not перед and

```
(defrule example1
(test (> 80 (startup-value)))
=>
(defrule example2
(test (> 80 (startup-value)))
(object (is-a MACHINE))
=>
(defrule example3
(machine ?x)
(not (and (not (part ?x ?y))
(inventoried ?x)))
=>)
```

Приклад 9.49. Перетворені правила з умовами test й not перед and

```
(defrule example1
(initial-fact)
(test (> 80 (startup-value)))
=>
(defrule example2
(object (is-a INITIAL-OBJECT) (name [initial-object]))
(test (> 80 (startup-value)))
(object (is-a MACHINE))
=>
(defrule example3
(machine ?x)
(not (and (initial-fact)
(not (part ?x ?y))
(inventoried ?x)))
=>)
```

8.5.2.3. Використання елемента not перед test

Якщо відразу перед умовним елементом test використався умовний елемент not, то CLIPS автоматично переміщає умовний елемент not на місце першої умови безпосередньо наступного за test.

Наприклад, правило із прикладу 9.50 зміниться на еквівалентне (приклад 9.51):

Приклад 9.50. Правило з умовами not перед елементом test

```
(defrule example
(a ?x)
(not (b ?x))
```

```
(test (> ?x 5) )
=>)
```

Приклад 9.51. Перетворене правило з умовами not перед test

```
(defrule example
  (a ?x)
  (test (> ?x 5))
  (not (b ?x) )
  =>)
```

8.5.2.4. Використання елемента not перед or

Якщо відразу перед умовним елементом or використався умовний елемент not, то CLIPS автоматично заміняє комбінацію not/or на еквівалентну комбінацію and/not.

Наприклад, що впливає правило (приклад 9.52) будуть змінено так, як показано в прикладі 9.53.

Приклад 9.52. Правило з умовами not перед or

```
(defrule example
  (a ?x)
  (not (or (b ?x)
  (c ?x)))
  =>)
```

Приклад 9.53. Перетворене правило з умовами not перед or

```
(defrule example
  (a ?x)
  (and (not (b ?
  x))
  (not (c
  ?x) ) )
  =>)
```

8.5.2.5. Зауваження про автоматичне додавання й перегрупування умовних елементів

У завершення опису синтаксису лівої частини правил CLIPS оборотна увага на наступні важливі особливості:

- Повна версія лівої частини правила містить неявний умовний елемент and.
- Перетворення умовних елементів forall й exists до еквівалентних виражень за допомогою not й and виконується перед додаванням відповідних зразків у ліву частину правила.
- Умовний елемент test звичайно не використовується як перший елемент в умові and.
- Команди, що виводять інформацію про умовні елементи в лівій частині правила,

відображають інформацію про визначення правила у вигляді, у якому неї задав користувач. Інформація про перегрупування й додавання зразків initial-fact й initial-object не виводиться.

8.6. Команди й функції для роботи із правилами

Після того як ми повністю розібралися з поданням правил в CLIPS, розглянули внутрішні алгоритми обробки правил, стратегії дозволу конфліктів і синтаксис лівої частини правил, можна сміло переходити до вивчення функцій і команд, надаваних CLIPS для роботи із правилами.

8.6.1. Перегляд і видалення існуючих правил

Після створення правил за допомогою конструктора defrule цілком природно виникає бажання зробити що-небудь із уже існуючим правилом. CLIPS підтримує множина різних команд, що оперують із правилами. У даному розділі ми розглянемо найбільше часто використовувані команди: ppdefrule, list-defrules й undefrule.

За допомогою команди ppdefrule можна переглянути визначення правила в тому вигляді, у якому воно було створено за допомогою конструктора defrule.

Визначення 9.25. Синтаксис команди ppdefrule (ppdefrule <ім'я-правила>)

Для того щоб одержати повний список правил, присутніх в CLIPS у цей момент, використовується команда list-def rules.

Визначення 9.26. Синтаксис команди list-defrules (list-defrules <ім'я-модуля>)

Повний синтаксис цієї команди містить необов'язковий аргумент <ім'я-модуля>. Якщо даний аргумент не заданий, то буде виведений список правил, певних у поточному модулі. У випадку явного завдання модуля буде список правил, що належать конкретному модулю. Даний аргумент може приймати значення *. У цьому випадку на екран буде виведений список всіх правил із всіх модулів.

Для видалення правила використовується команда undefrule.

Визначення 9.27. Синтаксис команди undefrule (undefrule <ім'я-правила>)

Як параметр команда undefrule приймає ім'я правила, яке потрібно видалити. Якщо як ім'я правила був заданий символ *, то будуть видалені всі правила.

Для демонстрації роботи команд, наведених у цьому й наступному розділах, будемо використати наступні правила:

Приклад 9.54. Необхідні для подальшої роботи правила

```
(defrule Make
  (a)
  (b)
  =>
  (assert (c)))

(defrule Make
  (c)
  (or (a)
  ( b))

  =>
  (assert (d)))

(defrule Make
  (d)
  (or (a)
  (b)
  (c))
  =>
  (assert (e)))
```

Введемо ці правила в середовище CLIPS, а потім виконаєте наступну послідовність команд:

Використання команд ppdefrule, list-defrules й undefrule

```
(ppdefrule Make)
(list-defrules)
(undefrule Make)
(list-defrules)
(undefrule *)
(list-defrules)
```

Якщо наведені вище дії були виконані правильно, то отриманий результат повинен відповідати мал. 8.6.

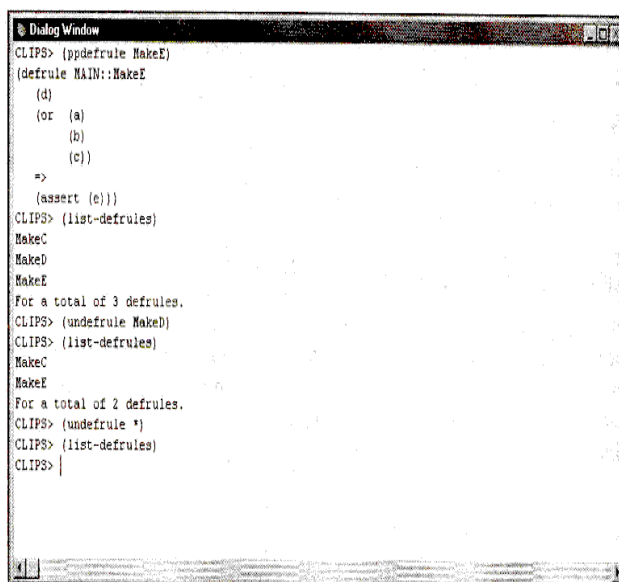


Рис. 8.6. Результат застосування команд ppdefrule, list-defrules й undefrule

Як уже згадувалося в лекції 8 користувачам Windows-версії CLIPS доступний інструмент за назвою **Defrule Manager** (Менеджер правил). Якщо в цей момент у середовищі CLIPS відсутні правила, то пункт **Defrule Manager** меню **Browse** не буде доступний. Якщо ви повторно заведете наведені вище правила й відкриєте менеджер правил, то повинні будете побачити результат, наведений на мал. 9.7. Менеджер відображає список всіх правил, доступних у цей момент. Загальна кількість правил відображається в заголовку вікна менеджера, у цей момент це **Defrule Manager — 3 Items**. За допомогою кнопок **Remove** й **Pprint** можна видаляти й виводити визначення обраного правила відповідно. Вся інформація, одержувана від менеджера правил, відображається безпосередньо в головному вікні CLIPS.

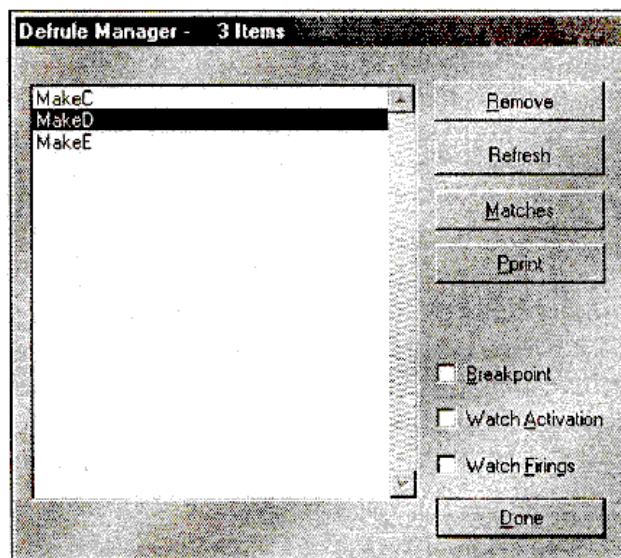


Рис. 8.7. Перегляд списку правил за допомогою менеджера правил

CLIPS не містить спеціальних команд для зміни існуючих правил. Щоб змінити існуюче правило,

користувачеві необхідно заново визначити таке правило за допомогою конструктора `def rule`. При цьому існує визначення правила буде автоматично вилучено із системи, навіть якщо новий конструктор містив помилки, і нове правило додане не було.

8.6.2. Збереження правил

Як ви вже встигли переконатися, створювати правила конструктором `defrule` щораз, у міру необхідності використовуючи для цього середовище CLIPS, досить незручно. Для полегшення участі користувача CLIPS дозволяє завантажувати конструктори правил (як, втім, і всі інші конструктори) з текстового файлу. Для цього використовується наступна команда:

Визначення 9.28. Синтаксис команди `load` (`load <ім'я-файлу>`)

Ім'я файлу повинне бути рядком, тобто полягати в лапці. Ім'я файлу може містити повний шлях до файлу. У противному випадку система буде шукати файл у поточному каталозі. Для створення файлу в принципі можна використати будь-який ASCII-редактор, але краще застосовувати убудований редактор, надаваний середовищем CLIPS. Убудований редактор підтримує кілька додаткових функцій, надзвичайно корисних при розробці програм. По-перше, він здатний перевіряти синтаксис функцій, баланс відкриваючих і закриваючих дужок, допомагає в розміщенні й видаленні коментарів і т.д. Якщо ви будете використати убудований редактор для створення серйозної експертної системи, ви по достоїнству оціните ці можливості. По-друге, убудований редактор дозволяє швидко завантажувати в середовище окремі конструктори й команди. Ця можливість допомагає перевіряти й тестувати велику експертну систему. І, нарешті, по-третє, редактор надає допомогу по середовищу й мові, що буває надзвичайно корисної, навіть при наявності великого досвіду роботи в CLIPS. За замовчуванням файли, створені в убудованому редакторі CLIPS, одержують розширення `clp`. Для початку роботи з редактором просто виберіть пункт **New** меню **File**.

Створимо в CLIPS файл `example1.CLP` із трьома наведеними вище правилами. Після чого очистите CLIPS за допомогою команди `clear` і виконаєте команду (`load "example1.CLP"`). Отриманий результат повинен відповідати мал. 8.8.

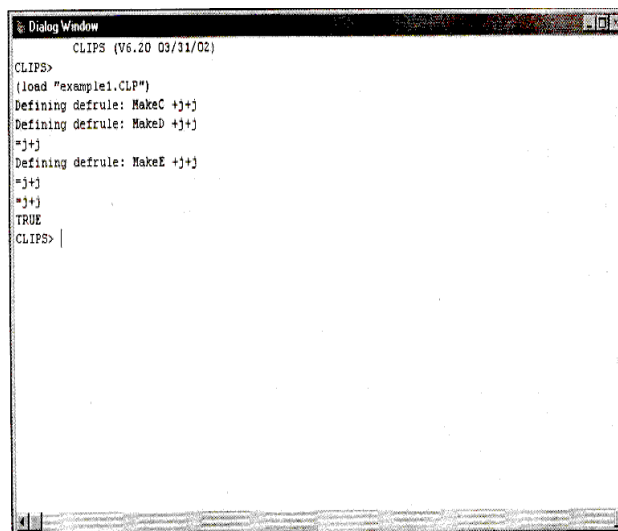


Рис. 9.8. Результат завантаження файлу `example1.CLP`

Команда `load` відображає процес завантаження кожного конструктора. У випадку успішного завантаження всіх певних у файлі конструкторів команда повертає значення `TRUE`, у противному випадку — інформацію про помилку. У випадку якщо була знайдена помилка, процес завантаження файлу припиняється.

CLIPS підтримує також команду `load*`. Ця команда повністю ідентична `load` за винятком того, що вона не відображає процесу завантаження конструкторів.

Визначення 9.29. Синтаксис команди `load*` (`load* <ім'я-файлу>`)

CLIPS надає також команду `save`, що дозволяє зберігати в текстовий файл всі конструктори, певні в цей момент у системі. Синтаксис цієї команди ідентичний синтаксису команд `load` й `load*`.

Визначення 9.30. Синтаксис команди `save` (`save <ім'я-файлу>`)

Текстовий формат не єдиний спосіб зберігання конструкторів CLIPS. Команди `bsave` й `bload` дозволяють зберігати й завантажувати конструктори у двійковому виді. Двійкові файли завантажуються набагато швидше, ніж текстові, але займають більше місця (тому що крім конструкторів вони зберігають повну інформацію про поточний стан середовища). Ще однією незручністю використання двійкових файлів є те, що створювати їх можна тільки безпосередньо в середовищі CLIPS.

Більшість описаних вище команд для роботи з файлами (`a save`, `load`, `bsave` й `bload`) доступні в меню **File** Windows-версії середовища CLIPS. Це команди **Load**, **Save**, **Save Binary** й **Load Binary** відповідно. Усі використовують стандартні Windows-діалоги для вибору файлів.

8.6.3. Запуск і зупинка програми

Як було замічено, для запуску CLIPS-програм використовується команда `run`.

Визначення 8.31. Синтаксис команди `run` (`run <цілочисельний-вираз>`)

Цілочисельне вираження є обов'язковим аргументом команди `run`. У найпростішому випадку як цей аргумент можна використати будь-яку цілую константу. Якщо даний аргумент заданий і він позитивний, то CLIPS запустить на виконання задане число правил із плану рішення задачі. Якщо дане число більше числа правил у плані рішення задачі, то буде запущені всі правила. У випадку якщо аргумент не заданий або є негативним, план рішення задачі також буде виконаний повністю.

В Windows-версії CLIPS у меню **Execution** доступні дві версії команди `run` — **Run** й **Step**. Перша команда використає версію команди `run` без аргументів і запускає всі правила із плану рішення задачі. Програма **Step** дозволяє трасувати програму й виконувати задане число правил. За замовчуванням це число дорівнює 1, але цю установку середовища можна змінити за допомогою діалогового вікна **Preferences**. Для запуску цього діалогового вікна виберіть пункт **Preferences** меню **Execution**. Загальний вид діалогового вікна показаний на мал. 8.9. Кількість правил, запущених за один крок трасування, відображається в полі **Step Rule Firing Increment**.

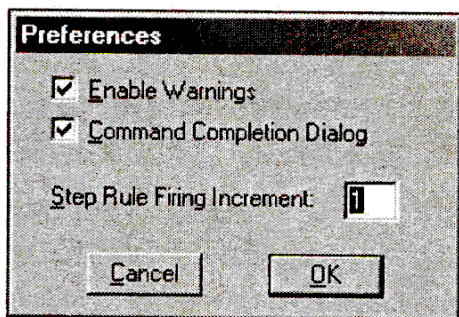


Рис. 8.9. Діалогове вікно **Preferences**

Крім виконання програми по кроках, CLIPS дозволяє установку точок зупину (breakpoints) на окремих правилах.

Визначення 9.32. Синтаксис команди `set-break` (`set-break <ім'я-правила>`)

Якщо крапка останова визначена для заданого правила, то виконання програми припиниться перед запуском цього правила. Крапка останова не зупиняє правило, якщо це перше правило в плані рішення задачі.

Видалити крапки останова можна за допомогою команди `remove-break`.

Визначення 8.33. Синтаксис команди `remove-break` (`remove-break <ім'я-правила>`)

У випадку виконання команди `remove-break` без параметрів CLIPS видалить всі певні раніше крапки останова.

Установлювати й знімати крапки останова також можна за допомогою менеджера правил, зовнішній вигляд якого представлений на мал. 8.7. Для цього виберіть правило й установите прапорець **Breakpoint**.

У випадку якщо виконання програми необхідно зупинити, **використайте** команду `halt` без аргументів.

Визначення 8.34. Синтаксис команди `halt` (`halt`)

Діалогове вікно **Watch Options** (пункт **Watch** меню **Execution**) дозволяє встановити прапорець **Statistics**, як показано на мал. 8.10.

У цьому випадку після виконання кожної команди `run` CLIPS буде виводити статистичну інформацію про кількість запущених правил, повному й середньому часі виконання правил, кількості доданих фактів і т.д.

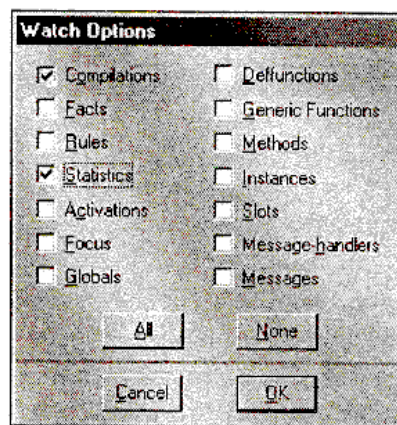


Рис. 8.10. Установка режиму з висновком статистичної інформації

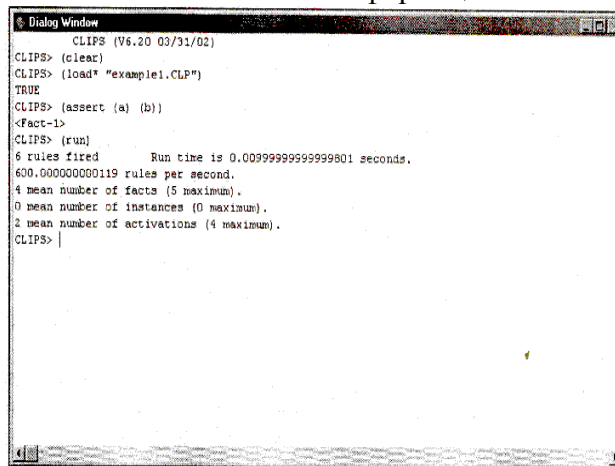


Рис. 8.11. Одержання статистичної інформації

Якщо ви встановите прапорець **Statistics**, завантажимо файл `example 1.CLP`, додамо факти (a) і (b) і запустимо програму, то побачимо результати,

представлені на мал. 8.11.

8.6.4. Перегляд плану рішення задачі

План рішення задачі (agenda) можна переглядати різними способами. Найпростіший з них - команда agenda, набрана в головному вікні CLIPS. Очистите CLIPS, завантажте файл example1.CLP, додайте факти a, b, c і d і викличте команду agenda. Отриманий результат повинен відповідати наведеному на мал. 8.12.

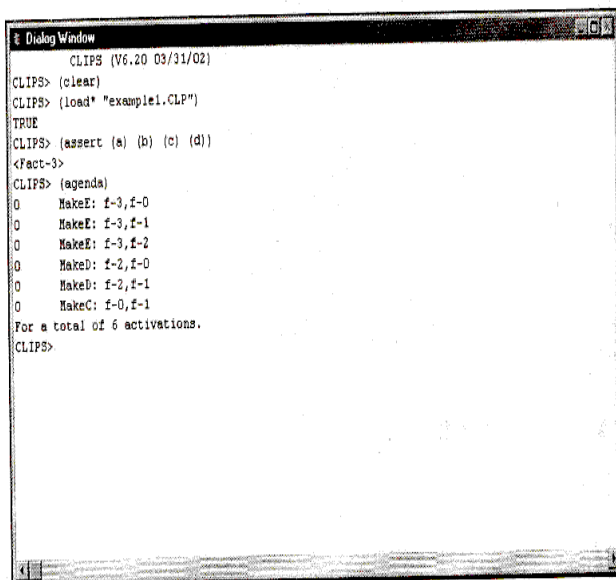


Рис. 8.12. Перегляд плану рішення задачі

План рішення задачі містить 6 активацій правил. По команді agenda всі ці активації будуть виведені на екран разом із пріоритетом правил (ліворуч від імені правила) і списком даних, що активували правило (праворуч від імені правила). Порядок правил у плані рішення задачі сильно залежить від обраної стратегії дозволу конфліктів і пріоритету правил.

Крім цього, Windows-версія CLIPS дозволяє виводити план рішення задачі в окремому вікні — **Agenda**. Для того щоб зробити вікно видимим, скористайтесь пунктом **Agenda Window** меню **Window**. Зовнішній вигляд цього вікна показаний на мал. 8.13, його вміст повністю відповідає інформації, одержуваної за допомогою команди agenda. Даний інструмент надзвичайно корисний при налагодженні програм або для спостереження за зміною плану рішення задачі в процесі виконання програми.

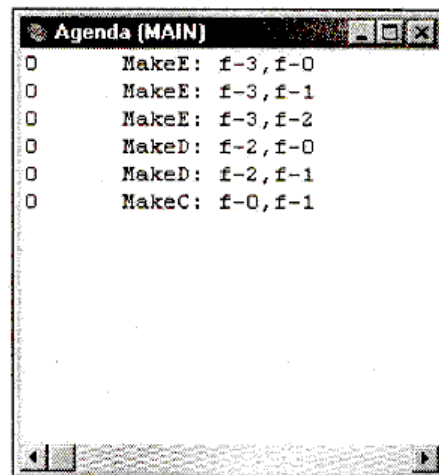


Рис. 8.13. Вікно Agenda

Крім вікна **Agenda**, що дозволяє тільки перегляд, CLIPS надає ще один зручний візуальний інструмент — **Agenda Manager** (Менеджер плану рішення задачі), що дозволяє якщо буде потреба коректувати план рішення задачі. Для виклику менеджера плану рішення задачі виберіть пункт **Agenda Manager** меню **Browse**. Зовнішній вигляд цього інструмента наведений на мал. 8.14. З його допомогою можна видаляти із плану рішення задачі окремі активації правил або запускати правила в деякому довільному порядку.

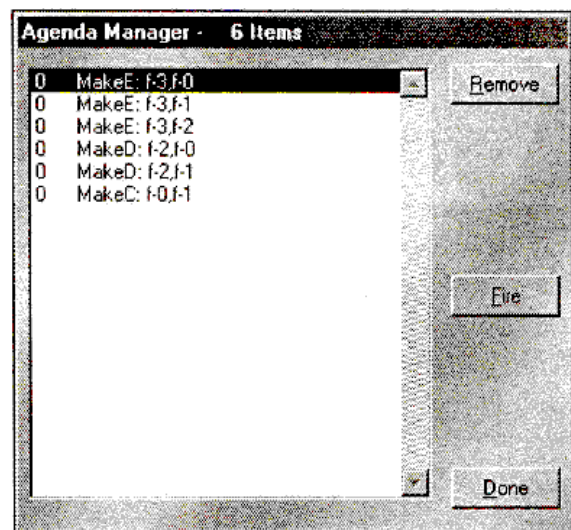


Рис. 8.14. Вікно менеджера плану рішення задачі

За допомогою діалогового вікна **Watch Options** (див. мал. 8.10) або менеджера правил можна задавати режим відображення активацій й/або запуску правил. У цьому випадку користувач буде одержувати відповідне інформаційне повідомлення при додаванні правила в план рішення задачі або при видаленні правила з нього, а також при кожному запуску правила.

8.6.5. Перегляд даних, здатних активувати правило

CLIPS надає можливість переглядати списки наборів даних (фактів або об'єктів), здатних активувати задане правило.

Визначення 8.35. Синтаксис команди matches (matches <ім'я-правила>)

Команда matches виводить інформацію про всі можливі набори даних, здатних активувати це правило. Подивіться на результати виконання даної команди для правил Make й Make (наявність фактів a, b і з обов'язково), наведені на мал. 9.15 й 9.16 відповідно.

На цьому ми закінчимо вивчення синтаксису правил CLIPS й основних команд і функцій для роботи з ними.

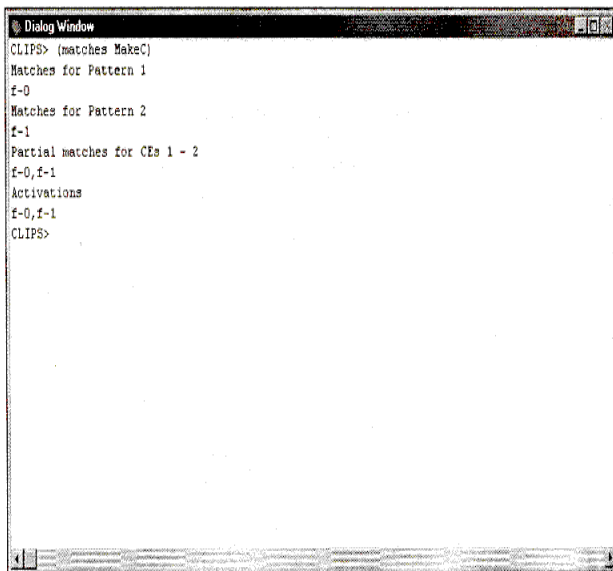


Рис. 8.15. Дані, що активують правило Make

Глава 6. Правила

253

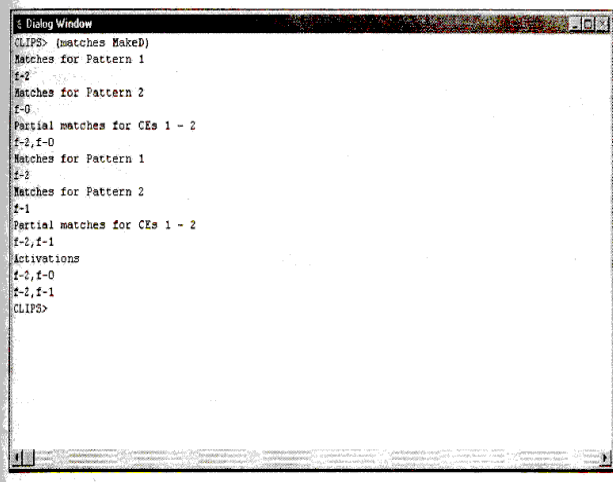


Рис. 8.16. Приведемо кілька прикладів використання зразків об'єктів.

Приклад 9.22. Використання зразків об'єктів

```
(defrule example-1
  (object (is-a MyObj1 | MyObj2) )
  =>)
```

```
(defrule example-2
  (object (is-a ?x) )
  (object (is-a ~?x) )
  =>)
(defrule example-3
  (object (width ?x&(> ?x 20)))
  =>)
(defrule example-4
  (object (width ?x) (height ?x))
  =>)
```

Перше правило задовольняє будь-який об'єкт класу MyObj1 або MyObj2. Друге правило активується будь-якою парою об'єктів, що належить різним класам. Третє правило виконується у випадку, якщо буде знайдений об'єкт активного класу, що містить активний слот width, значення якого більше 20. Останній наведений приклад задовольняється будь-яким об'єктом активного класу, що містить активні слоти width й height, значення яких повинні бути рівні.

8.6.6. Адреса зразка

Деякі дії в правій частині правил, такі як retract й unmake-instance, оперують із фактами або об'єктами, що беруть участь у лівій частині. Для того щоб визначити, який факт або об'єкт буде змінюватися, необхідно привласнити змінної адресу конкретного факту або об'єкта. Присвоювання адрес відбувається в лівій частині правила й отримане значення називається *адресою зразка* (pattern-address).

Визначення 8.16. Синтаксис адреси зразка

<адреса-зразка> ::= ?<ім'я-змінної> <-> <зразок>

Стрілка вліво (<-) - необхідна частина синтаксису. Змінна, пов'язана з адресою факту або об'єкта, може рівнятися з інший змінної або використатися зовнішньою функцією. Змінна, пов'язана з адресою факту або об'єкта, може бути також використана для наступного обмеження полів у зразку умовного вираження. Однак не можна зв'язувати змінну в умовному вираженні not.

Як приклад приведемо просте правило, що видаляє всі факти data.

Приклад 9.23. Правило del-data-facts

```
(defrule del-data-facts
  ?data-facts <- (data
  $?)
  =>
  (retract ?data-facts))
```

8.6.7. Умовний елемент test

Умовний елемент test надає можливість накладення додаткових обмежень на слоти фактів або об'єктів. Елемент test задовольняється, якщо викликана в ньому функція повертає значення не-false. Як й у випадку предикатних обмежень зразка в умовному елементі test, можна використати змінні, уже зв'язані зі своїми значеннями. У середині елемента test можуть бути виконані різні логічні операції, наприклад порівняння змінних.

Визначення 9.17. Синтаксис умовного елемента test

```
<умовний-елемент-test > ::=  
(test <виклик-функції>)
```

Виразження test обчислюється щораз при задоволенні інших умовних елементів. Це означає, що умовний елемент test буде обчислений більше одного разу, якщо оброблюване виразження може бути задоволене більш ніж однією групою даних. Використання умовного елемента test може стати причиною автоматичного додавання правила деяких умовних виразень. Крім того, CLIPS може автоматично з умовні елементи test.

Наведене нижче правило знаходить пару фактів data, причому різниця між значеннями перших полів цих фактів повинна бути більше або рівної 3.

Приклад 9.24. Застосування умовного елемента test

```
(defrule example  
  (data ?x)  
  (data ?y)  
  (test (>= (abs (- ?y ?x) )  
           3))  
  =>)
```

Умовний елемент test може привести до автоматичного додавання зразків initial-fact або initial-object у ліву частину правила. Тому не забувайте використати команду reset (яка створює initial-fact й initial-object), щоб бути впевненим у коректній роботі умовного елемента test.

8.6.8. Умовний елемент or

Умовний елемент or дозволяє активувати правило кожним з декількох заданих умовних елементів. Якщо який-небудь із умовних елементів, об'єднаних за допомогою or, удоволений, то й все виразження or вважається вдоволеним. У цьому випадку, якщо всі інші умовні елементи, що входять у ліву частину правила (але не входні в or), також задоволені, правило буде активовано. Умовний елемент or може поєднувати будь-яка кількість елементів.

Правило буде активовано для кожного виразження в умовному елементі or, що було задоволено. Таким чином, умовний елемент or

робить ефект, ідентичний написанню декількох правил зі схожими посилками й наслідками.

Визначення 9.18. Синтаксис умовного елемента or

```
<умовний-елемент-or > ::= (or  
<умовний-елемент>+)
```

Приклад 9.25. Застосування умовного елемента or

```
(defrule system-fault  
  (error-status unknown) (or (temp  
    high)  
    (valve broken)  
    (pump off))  
  =>  
  (printout t "The system has  
    a fault." crlf))
```

Дане правило повідомить про поломку системи, якщо в списку фактів буде присутній факт error-status unknown й один з фактів temp high, valve broken або pump off. У випадку якщо будуть присутні два із цих трьох фактів, наприклад temp high й pump off, те повідомлення буде виведено два рази.

Помітьте, що наведений приклад - точний еквівалент наступних трьох окремих правил:

Приклад 9.26. Еквівалент правила system-fault

```
(defrule system-fault-1  
  (error-status unknown)  
  (pump off)  
  =>  
  (printout t "The system has a  
    fault." crlf))  
(defrule system-fault-2  
  (error-status unknown) (valve  
    broken)  
  =>  
  (printout t "The system has a  
    fault." crlf))  
(defrule system-fault-3  
  (error-status unknown) (temp  
    high)  
  =>  
  (printout t "The system has a  
    fault." crlf))
```

8.6.9. Умовний елемент and

Всі умовні елементи в лівій частині правил CLIPS об'єднані неявним умовним елементом and. Це означає, що всі умовні елементи, задані в лівій частині, повинні задовольнитися, для того щоб правило було активовано. За допомогою явного застосування умовного елемента and можна змішувати різні умови and й or і групувати елементи так, як цього вимагає логіка правил. Умова and задовольняється, тільки якщо всі умови усередині явного and задоволені. У випадку, якщо решта умов у лівій

частині правила також щирі, правило буде активовано. Елемент and може поєднувати будь-яке число умовних елементів.

Визначення 9.19. Синтаксис умовного елемента and

<умовний-елемент-and> ::= (and <умовн-елемент>+)

Приклад 9.27. Застосування умовного елемента and

```
(defrule system-flow
  (error-status confirmed)
  (or (and (temp high)
           (valve closed))
  (and (temp low)
        (valve open)))
  =>
  (printout t "The system is having
a flow problem." crlf))
```

Якщо умовний елемент and містить умовні елементи test або not як перший елемент, то перед ними автоматично додається зразок initial-fact або initial-object. Пам'ятаємо, що ліва частина будь-якого правила містить неявний елемент and, тому наведене в прикладі 9.28 правило буде автоматично перетворений (див. приклад 9.29).

Приклад 9.28. Правило nothing-to-schedule

```
(defrule nothing-to-schedule
  (not (schedule ?))
  =>
  (printout t "Nothing to
schedule." crlf))
```

Приклад 9.29. Перетворене правило nothing-to-schedule

```
(defrule nothing-to-schedule
  (and (initial-fact)
        (not (schedule ?)))
  =>
  (printout t "Nothing to
schedule." crlf))
```

8.6.10. Умовний елемент not

Іноді важливіше відсутність інформації, а не її присутність, тобто виникають ситуації, коли необхідно запустити правило, якщо зразок або інший умовний елемент не задовольняється (наприклад, факт не існує). Умовний елемент not надає цю можливість. Елемент not задовольняється, тільки якщо умовний елемент, що він містить, не задовольняється.

Визначення 9.20. Синтаксис умовного елемента not

<умовний-елемент-not> ::= (not <умовний-елемент>)

Умовний елемент not може заперечувати тільки одне вираження. Кілька умовних елементів потрібно заперечувати за допомогою декількох елементів not. Ретельно стежите за комбінаціями not з or або and; результат не завжди очевидний!

Приклад 9.30. Застосування умовного елемента not

```
(defrule high-flow-rate
  (temp high)
  (valve open)
  (not (error-status
confirmed))
  =>
  (printout t "Recommend
closing of valve due to
high temp" crlf))
```

У логічному елементі not можна використати зв'язані змінні, так само як й в інших умовних елементах:

Приклад 9.31. Правило check-value

```
(defrule check-valve
  (check-status ?valve)
  (not (valve-broken ?
valve))
  =>
  (printout t "Device " ?
valve " is OK" crlf))
```

За допомогою умовного елемента not можна, нарешті, довести до досконалості наше правило Find-2-coeval-Person.

Якщо ви пам'ятаєте, це правило виводить усілякі пари персон однакового віку. Щоб дане правило не виводило еквівалентні за змістом пари імен (наприклад, Bob-Sue й Sue-Bob), перетворимо нашу програму наступним образом:

Приклад 9.32. Поліпшене правило Find-2-coeval-Person

```
(deftemplate person
  (slot name)
  (slot age))
(deftemplate person-pair
  (slot name1)
  (slot name2)
  (slot age))
(deffacts people
  (person (name Joe) (age
20))
  (person (name Bob) (age
20))
  (person (name Joe) (age
34))
  (person (name Sue) (age
```

```

34))
(person (name Sue) (age
20)))
(defrule Find-2-Coeval-Person
(person (name ?x) (age ?z))
(person (name ?ys~?x) (age ?z))
(not (person-pair (name1 ?
x) (name2 ?y) (age ?z)))
(not (person-pair (name1 ?
y) (name2 ?x) (age ?z)))
=>
(printout t "name=" ?x " name=" ?
y " age=" ?z crlf)
(assert (person-pair (name1 ?x)
(name2 ?y) (age ?z))))

```

Зверніть увагу на зроблені зміни. По-перше, за допомогою конструктора `deftemplate` був Доданий додатковий шаблон `person-pair`. У фактах, що відповідають даному шаблону, буде зберігатися інформація про вже знайдені пари ровесників.

Крім того, було сильно змінено й саме правило. У його лівій частині було додано дві умови:

```

(not (person-pair (name1 ?x) (name2 ?
y) (age ?z)))
(not (person-pair (name1 ?y) (name2 ?
x) (age ?z)))

```

Ці умовні елементи перевіряють наявність фактів типу `person-pair` й, тим самим відслідковують, чи була вже оброблена дана пара або її перестановка. Якщо ці факти відсутні, то це означає, що обробка ще не була виконана. У цьому випадку правило активується, і виконуються дії, описані в правій частині правила. А саме виводиться на екран повідомлення про знайдену пару ровесників і додається відповідний факт `person-pair`, що затверджує, що дана пара вже була оброблена.

Для запуску програми виконайте команди `reset` й `run`. Програма виведе на екран наступну інформацію:

Приклад 9.33. Результат роботи правила Find-2-Coeval-Person

```

name=Sue name=Bob age=20
name=Sue name=Joe age=20
name=Sue name=Joe age=34
name=Bob name=Joe age=20

```

Якщо ви уважно подивитеся на отриманий результат і вихідні дані, то виявите, що це саме те, що нам було потрібно. Це список усіляких ровесників без повторень і з виключенням того факту, що всі люди є ровесниками самі собі. Тепер наше правило досягло повної досконалості! Зверніть увагу на той факт, що якщо ви повторно спробуєте виконати команду `run`, те нічого не побачите. Це відбувається тому, що в списку фактів утримується інформація про всі оброблені пари, що залишилася після першого запуску. Для того щоб повторно запускати даний приклад, виконуйте команду `reset` перед кожною командою `run`.

Умовний елемент `not`, так само як й `test`, може привести до автоматичного додавання зразків `initial-fact` або `initial-object` у лівій частині правил. Тому не забувайте використати команду `reset` (яка створює `initial-fact` й `initial-object`), щоб бути впевненим у коректній роботі умовного елемента `not`.

В умовний елемент `not`, що містить елемент `test`, автоматично перетвориться в елемент `not`, що містить `and` з `initial-fact` і вихідним елементом `test`. Наприклад, що впливає умовний елемент із прикладу 9.34 перетвориться в елемент із прикладу 9.35.

Приклад 9.34. Умовний елемент not, що містить елемент test

```

(not (test (> ?time-1 ?time-
2)))

```

Приклад 9.35. Перетворений умовний елемент not, L утримуючий елемент test

```

(not (and (initial-fact)
(test (> ?time-
1 ?time-2))))

```

Помітьте, що найбільш простим і правильним способом запису даного **виразу** буде:

```

(test (not (> ?time-1 ?time-2) ) ).

```

8.6.11. Умовний елемент exists

Умовний елемент `exists` дозволяє визначити, чи існує хоча б один набір даних (фактів або об'єктів), які задовольняють умовним елементам, заданим усередині елемента `exists`.

Визначення 9.21. Синтаксис умовного елемента exists

```

<умовний-елемент-exists> ::=
(exists <умовн-елемент>+)

```

CLIPS автоматично замінює `exists` двома послідовними умовними елементами `not`. Наприклад, що впливає правило (приклад 9.36) буде перетворено в правило із прикладу 9.37.

Приклад 9.36. Правило example

```

(defrule example
(exists (a ?x) (b ?
x))=>

```

Приклад 9.37. Перетворене правило example

```

(defrule example
(not (not (and (a ?x) (b ?
x) ) ) )=>

```

Тому що внутрішній спосіб реалізації `exists` використовує умовний елемент `not`, то для `exists` справедливі всі зауваження й обмеження, наведені в попередніх лекціях.

Розглянемо наступний приклад:

Приклад 9.38. Використання умовного елемента exists

```
(deftemplate hero
  (multislot name)
  (slot status (default
    unoccupied)))

(deffacts goal-and-heroes
  (goal save-the-world)
  (hero (name Death Defying
    Man))
  (hero (name Stupendous
    Man))
  (hero (name Incredible
    Man)))

(defrule save-the-world
  (goal save-the-world)
  (exists (hero (status
    unoccupied)))
  =>
  (printout t "The day is
    saved." crlf))
```

Дана програма визначає шаблон - героя, що має складене поле з ім'ям героя й простої поле, що містить статус "не зайнятий" за замовчуванням. Конструктор deffacts визначає трьох нічим не зайнятих героїв і поточну мету - порятунок миру. Правило перевіряє, є в цей момент ця мета, і у випадку позитивної відповіді перевіряє, якщо чи який-небудь ще не зайнятий герой. Якщо всі умовні елементи правила задоволені, воно повідомляє, що мир урятований. Зверніть увагу: незважаючи на те, що в нас всі три герої не зайняті, правило буде активовано тільки один раз.

Тому що спосіб реалізації exists використовує умовний елемент not, те умовний елемент exists може привести до автоматичного додавання зразків initial-fact або initial-object у ліву частину правила. Тому не забувайте використати команду reset (яка створює initial-fact й initial-object), щоб бути впевненим у коректній роботі умовного елемента exists.

8.6.12. Умовний елемент forall

Умовний елемент forall дозволяє визначити, що деяка задана умова виконується для всіх заданих умовних елементів.

Визначення 9.22. Синтаксис умовного елемента forall

```
<умовний-елемент forall> ::= (forall
  <умовний-елемент> <умовний-елемент>+)
```

CLIPS автоматично заміняє f про rail комбінацію умовних елементів not й and. Наприклад, що впливає правило (приклад 9.39) буде перетворено так, як показано в прикладі 9.40.

Приклад 9.39. Правило example

```
(defrule example
  (forall (a ?x) (b ?x)
    (c ?x))=>)
```

Приклад 9.40. Перетворене правило example

```
(defrule example
  (not (and (a ?x)
    (not (and
      (b ?x)
      (c ?x)))))=>)
```

Розглянемо наступний приклад. Правило all-students-passed визначає, чи пройшли всі студенти читання, правопис й арифметику, використовуючи умову forall:

Приклад 9.41. Правило all-students-passed

```
(defrule all-students-passed
  (forall (student ?name)
    (reading ?name)
    (writing ?name)
    (arithmetic ?
      name))
  =>
  (printout t "All students
    passed." crlf))
```

Помітьте, що дане правило задовольняється, поки немає жодного студента. При додаванні факту (student Bob) правило перестав задовольнятися, тому що немає фактів, що підтверджують, що Bob пройшов всі необхідні предмети. Правило не почне задовольнятися й після додавання фактів (reading Bob) і (writing Bob). А от після додавання факту (arithmetic Bob) правило буде активоване й зможе вивести на екран відповідний запис. Якщо додати факт (student John), правило знову перестане задовольнятися, тому що один зі студентів (John) не пройшов всі необхідні предмети. Використовуючи умовний елемент exists, ви без праці зможете змінити це правило так, щоб воно не виконувалося у випадку відсутності студентів.

Тому що реалізація forall використовує умовний елемент not, те forall, так само як й not, test й exists, може привести до автоматичного додавання зразків initial-fact або initial-object у ліву частину правила. Не забувайте використати команду reset для коректної роботи цього умовного елемента.

8.6.13. Умовний елемент logical

Умовний елемент logical надає механізм підтримки вірогідності для створених правилом даних (фактів або об'єктів), що задовольняють зразкам. Дані, створені в правій частині правила, можуть мати логічну залежність від даних, що задовольнили зразки в лівій частині правила. Така залежність називається *логічною підтримкою*. Дані можуть залежати від групи даних

або декількох груп даних, що задовольнили одне або кілька правил. Якщо віддаляються дані, які підтримують деякі інші дані, то залежні дані також автоматично віддаляються.

Якщо деякі дані створені без логічної підтримки (наприклад, за допомогою конструкторів `def facts`, `def instance` або команди `assert`, уведеної користувачем або викликаній в правій частині правила), то вважається, що вони мають безумовну підтримку. Безумовна підтримка видаляє всі присутні в цей момент умовні підтримки цих даних (але не видаляє самі дані). Подальша логічна підтримка для даних з безумовною підтримкою ігнорується. Видалення правила, що викликало логічну підтримку для даних, видаляє логічну підтримку, згенеровану цим правилом (але не видаляє дані, якщо в них ще є логічна підтримка, згенерована іншим правилом).

Визначення 9.23. Синтаксис умовного елемента `logical`

`<умовний-елемент-logical> ::= (logical <умовний-елемент>+)`

Умовний елемент `logical` групує зразки, так само як це робить `and`. Дана властивість можна використати при об'єднанні елементів `and`, `or` й `not`. Однак тільки перші *n* зразків правила можуть використатися в умовному елементі `logical`. Наприклад, що впливає правило записане вірно:

Приклад 9.42. Правильний варіант використання умовного елемента `logical`

```
(defrule ok
  (logical (a))
  (logical (b))
  (c)
  =>
  (assert (d)))
```

А таке оголошення правил неприпустиме:

Приклад 9.43. Неправильні варіанти використання умовного елемента `logical`

```
(defrule not-ok-1
  (logical (a))
  (b)
  (logical (c))
  =>
  (assert (d)))

(defrule not-ok-2
  (a)
  (logical (b))
  (logical (c))
  =>
  (assert (d)))

(defrule not-ok-3
```

```
(or (a)
(logical (b)))
(logical (c))
=>
(assert (d)))
```

Розглянемо наступний приклад. Включите перегляд списку фактів за допомогою пункту **Facts Window** меню **Windows**, це допоможе стежити за тим, що відбувається в момент виконання програми.

Приклад 9.44. Використання умовного елемента `logical`

```
(clear)
(reset)
(defrule example
  (logical (a))
  (b)
  =>
  (assert (c)))

(assert (a) (b) )
(run)
(retract 2)
(retract 1)
```

По команді `run` правило `example`, активоване фактами `a` й `b`, додає новий факт `із`, що має логічну підтримку (залежить) від факту `a`.

Після видалення факту `b` за допомогою команди `(retract 2)` нічого особливого не відбувається, але якщо ми видалимо факт `a`, те побачимо, що це відразу приведе до видалення пов'язаного з ним факту `с`.

Як згадувалося, умовний елемент `logical` може бути використаний для створення даних, які будуть логічно пов'язані зі змінами деяких окремих слотів об'єкта, а не від усього об'єкта цілком. Дану можливість можна використати тільки при роботі з об'єктом. При роботі із шаблонами фактів дану можливість використати не можна, тому що зміни слоту факту фактично приводить до видалення старого факту й додавання нового зі зміненими слотами й індексом.

На відміну від фактів зміни слотів об'єкта виконуються без видалення об'єкта. Це поводження ілюструється наведеним нижче приміром:

Приклад 9.45. Використання умовного елемента `logical` з об'єктами

```
(clear)
(defclass A (is-a USER)
  (role concrete)
  (pattern-match reactive)
  (slot foo (create-accessor write))
  (slot bar (create-accessor write)))

(defrule match-A
  (logical (object (is-a A)
```

```
(foo ?)))  
=>  
(assert (new-fact)))  
(make-instance a of A)  
(run)  
(send [a] put-foo 100)
```

Після виконання команди run правило match-A додає факт new-fact, логічно пов'язаний з конкретним значенням слоту foo об'єкта a. При зміні значення даного слоту факт new-fact автоматично віддаляється зі списку фактів.

Додаток 1. Приклади програм в системі CLIPS

Д1.1. Приклад простої бази знань

```
; This is a very simple example of a
CLIPS knowledge base,
; just using the pattern matching to
create new knowledge.

; To run this example:
; 1) start CLIPSWin from Explorer
; 2) load the KB with (load
"emhl.txt")
; 3) initialize with (reset)
; 4) run the KB with (run)
; 5) to view the facts generated,
enter (facts)
; The current fact is for a patient
with symptoms of measles.
; To change this, edit the deffacts at
the end of this file.

; Before we can define rules and
facts, we have to define the
; "variables" used with deftemplate,
which defines the concept
; name, as well as the names of all
attributes which could be
; given values.

; A "Patient" may have values for
fever, spots, rash, sore_throat
; and inoculated.

(deftemplate Patient
  (slot fever)
  (slot spots)
  (slot rash)
  (slot sore_throat)
  (slot inoculated))

; These deftemplates represent
conclusions which we may assign
; values to as a result of the
inference.

(deftemplate Diagnosis
  (slot diagnosis))

(deftemplate Treatment
  (slot treatment))

; In its simplest form, a rule just
has a right side which is a
; "template" that the inference engine
will try to match to some
; fact. In this case, it matches a
Patient with specific values
; for fever, spots, and inoculated
```

```
(the other values don't matter).
```

```
; The left side of the => are actions
to take if the RHS matches a
; fact. In this case, we assert a new
fact (Diagnosis (diagnosis measles))
; and printout the string "Measles
diagnosed" to the terminal (t),
; followed by a return (crlf).
```

```
(defrule Measles
  (Patient (fever high) (spots yes)
(innoculated no))
=>
  (assert (Diagnosis (diagnosis
measles)))
  (printout t "Measles diagnosed"
crlf))
```

```
; We can also combine template
matches, using the standard
connectives
; of and, or, not. Note that the
syntax of CLIPS is prefix-oriented.
```

```
(defrule Allergy1
  (and (Patient (spots yes))
        (or (not (Patient (fever
high)))
              (Patient (inoculated
yes))))))
=>
  (assert (Diagnosis (diagnosis
allergy)))
  (printout t "Allergy diagnosed from
spots and lack of measles" crlf))
```

```
(defrule Allergy2
  (Patient (rash yes))
=>
  (assert (Diagnosis (diagnosis
allergy)))
  (printout t "Allergy diagnosed from
rash" crlf))
```

```
(defrule Flu
  (Patient (sore_throat yes) (fever
mild | high))
=>
  (assert (Diagnosis (diagnosis
flu)))
  (printout t "Flu diagnosed" crlf))
```

```
; Rules for recommending treatments on
the basis of
; Diagnosis facts created.
```

```
(defrule Penicillin
```

```

(Diagnosis (diagnosis measles))
=>
(assert (Treatment (treatment
pennicillin)))
(printout t "Penicillin prescribed"
crlf))

(defrule Allergy_pills
  (Diagnosis (diagnosis allergy))
  =>
  (assert (Treatment (treatment
allergy_shot)))
  (printout t "Allergy shot
prescribed" crlf))

(defrule Bed_rest
  (Diagnosis (diagnosis flu))
  =>
  (assert (Treatment (treatment
bed_rest))))
(printout t "Bed rest prescribed"
crlf))

; Facts are created with deffacts (the
can also
; be directly asserted while in
CLIPS). The list
; consists of a name, and a list of
facts.

(deffacts Symptoms
  (Patient (fever low)
            (spots yes)
            (rash no)
            (sore_throat no)
            (innoculated no)))

```

Д1.2. База знань з використанням властивості salience

```

; This is a very simple example of a
CLIPS knowledge base,
; just using the pattern matching to
create new knowledge.

; We have modified this to use
salience measure to order
; rules (necessary for the spots -->
allergy rule).

; A "Patient" may have values for
fever, spots, rash, sore_throat
; and inoculated.

(deftemplate Patient
  (slot fever)
  (slot spots)
  (slot rash)
  (slot sore_throat)

```

```

(slot inoculated))

; These deftemplates represent
conclusions which we may assign
; values to as a result of the
inference.

(deftemplate Diagnosis
  (slot diagnosis))

(deftemplate Treatment
  (slot treatment))

; Rules for determining diagnosis on
the basis of patient symptoms

; Salience added to give this rule
priority

(defrule Measles
  (declare (salience 100))
  (Patient (fever high) (spots yes)
(innoculated no))
  =>
  (assert (Diagnosis (diagnosis
measles)))
  (printout t "Measles diagnosed"
crlf))

; Modified to only fire if no measles

(defrule Allergy1
  (and (Patient (spots yes))
        (not (Diagnosis (diagnosis
measles)))))
  =>
  (assert (Diagnosis (diagnosis
allergy)))
  (printout t "Allergy diagnosed from
spots and lack of measles" crlf))

(defrule Allergy2
  (Patient (rash yes))
  =>
  (assert (Diagnosis (diagnosis
allergy)))
  (printout t "Allergy diagnosed from
rash" crlf))

(defrule Flu
  (Patient (sore_throat yes) (fever
mild | high))
  =>
  (assert (Diagnosis (diagnosis
flu)))
  (printout t "Flu diagnosed" crlf))

; Rules for recommending treatments on
the basis of
; Diagnosis facts created.

```

```

(defrule Penicillin
  (Diagnosis (diagnosis measles))
  =>
  (assert (Treatment (treatment
pennicillin))))
  (printout t "Penicillin prescribed"
crlf))

(defrule Allergy_pills
  (Diagnosis (diagnosis allergy))
  =>
  (assert (Treatment (treatment
allergy_shot))))
  (printout t "Allergy shot
prescribed" crlf))

(defrule Bed_rest
  (Diagnosis (diagnosis flu))
  =>
  (assert (Treatment (treatment
bed_rest))))
  (printout t "Bed rest prescribed"
crlf))

; Facts are created with deffacts (the
can also
; be directly asserted while in
CLIPS). The list
; consists of a name, and a list of
facts.

(deffacts Symptoms
  (Patient (fever high)
            (spots yes)
            (rash no)
            (sore_throat no)
            (innoculated no)))

```

Д1.3. Приклад бази знань з використанням порівняння за зразком

```

; This is a very simple example of a
CLIPS knowledge base,
; just using the pattern matching to
create new knowledge.

```

```

; We have modified it to bind facts to
patients with a given
; name (a new property of patient).

```

```

(deftemplate Patient
  (slot name)
  (slot fever)
  (slot spots)
  (slot rash)
  (slot sore_throat)
  (slot inoculated))

```

```

; These deftemplates represent
conclusions which we may assign
; values to as a result of the
inference.

```

```

(deftemplate Diagnosis
  (slot name)
  (slot diagnosis))

```

```

(deftemplate Treatment
  (slot name)
  (slot treatment))

```

```

; Rules for determining diagnosis on
the basis of patient symptoms

```

```

; In the following, ?n is a variable,
which gets "bound" to the value
; corresponding to the attribute (in
this case the "name" attribute)
; in the matching fact. That value
then gets used on the rest of the
; rule, including the assert and the
printout.

```

```

(defrule Measles
  (declare (salience 100))
  (Patient (name ?n) (fever high)
            (spots yes) (innoculated no))
  =>
  (assert (Diagnosis (name ?n)
(diagnosis measles)))
  (printout t "Measles diagnosed for
" ?n crlf))

```

```

; Modified to only fire if no measles

```

```

(defrule Allergy1
  (and (Patient (name ?n) (spots
yes))
       (not (Diagnosis (name ?n)
(diagnosis measles))))
  =>
  (assert (Diagnosis (name ?n)
(diagnosis allergy)))
  (printout t "Allergy diagnosed for
" ?n " from spots and lack of measles"
crlf))

```

```

(defrule Allergy2
  (Patient (name ?n) (rash yes))
  =>
  (assert (Diagnosis (name ?n)
(diagnosis allergy)))
  (printout t "Allergy diagnosed from
rash for" ?n crlf))

```

```

(defrule Flu
  (Patient (name ?n) (sore_throat
yes) (fever mild | high))
  =>

```

```

(assert (Diagnosis (name ?n)
(diagnosis flu)))
(printout t "Flu diagnosed for " ?n
crlf))

; Rules for recommending treatments on
the basis of
; Diagnosis facts created.

(defrule Penicillin
  (Diagnosis (name ?n) (diagnosis
measles))
=>
  (assert (Treatment (name ?n)
(treatment penicillin)))
  (printout t "Penicillin prescribed
for " ?n crlf))

(defrule Allergy_pills
  (Diagnosis (name ?n) (diagnosis
allergy))
=>
  (assert (Treatment (name ?n)
(treatment allergy_shot)))
  (printout t "Allergy shot
prescribed for " ?n crlf))

(defrule Bed_rest
  (Diagnosis (name ?n) (diagnosis
flu))
=>
  (assert (Treatment (name ?n)
(treatment bed_rest)))
  (printout t "Bed rest prescribed
for " ?n crlf))

; Facts are created with deffacts (the
can also
; be directly asserted while in
CLIPS). The list
; consists of a name, and a list of
facts.

(deffacts Symptoms
  (Patient (name "Fred")
    (fever high)
    (spots yes)
    (rash no)
    (sore_throat no)
    (innoculated no))
  (Patient (name "Barney")
    (fever mild)
    (spots yes)
    (rash no)
    (sore_throat no)
    (innoculated no))
  (Patient (name "Wilma")
    (fever high)
    (spots no)
    (rash no)
    (sore_throat yes))

```

```

(innoculated no)))

```

Д1.4. Приклад бази знань з використанням чисельних виразів

```

; This is a very simple example of a
CLIPS knowledge base,
; just using the pattern matching to
create new knowledge.

```

```

; We have modified this to use binding
and arithmetic operations
; to conclude the level of fever from
temperature.

```

```

; A "Patient" may have values for
temperature, spots, rash, sore_throat
; and inoculated.

```

```

(deftemplate Patient
  (slot temperature)
  (slot spots)
  (slot rash)
  (slot sore_throat)
  (slot inoculated))

```

```

; These deftemplates represent
conclusions which we may assign
; values to as a result of the
inference.

```

```

(deftemplate Diagnosis
  (slot diagnosis))

```

```

(deftemplate Treatment
  (slot treatment))

```

```

; We add another template for fever,
which is now a conclusion
; instead of a patient property

```

```

(deftemplate Fever
  (slot level))

```

```

; Rules for concluding fever from
temperature.

```

```

; Note that these rules find the
patient temperature, and then bind
; it to ?t. The next part uses the
test keyword to evaluate the
; conditional expression as true or
false.

```

```

(defrule Fever1
  (Patient (temperature ?t))
  (test (>= ?t 101))
=>
  (assert (Fever (level high)))

```

```

    (printout t "High fever diagnosed"
    crlf))

(defrule Fever2
  (Patient (temperature ?t))
  (test (and (< ?t 101) (> ?t 98.6)))
  =>
  (assert (Fever (level mild)))
  (printout t "Mild fever diagnosed"
  crlf))

; Rules for determining diagnosis on
the basis of patient symptoms
; Saliency added to give this rule
priority

(defrule Measles
  (declare (saliency 100))
  (Patient (spots yes) (innoculated
no))
  (Fever (level high))
  =>
  (assert (Diagnosis (diagnosis
measles)))
  (printout t "Measles diagnosed"
  crlf))

; Modified to only fire if no measles

(defrule Allergy1
  (and (Patient (spots yes))
  (not (Diagnosis (diagnosis
measles)))))
  =>
  (assert (Diagnosis (diagnosis
allergy)))
  (printout t "Allergy diagnosed from
spots and lack of measles" crlf))

(defrule Allergy2
  (Patient (rash yes))
  =>
  (assert (Diagnosis (diagnosis
allergy)))
  (printout t "Allergy diagnosed from
rash" crlf))

(defrule Flu
  (Patient (sore_throat yes))
  (Fever (level mild|high))
  =>
  (assert (Diagnosis (diagnosis
flu)))
  (printout t "Flu diagnosed" crlf))

; Rules for recommending treatments on
the basis of
; Diagnosis facts created.

(defrule Penicillin
  (Diagnosis (diagnosis measles))

```

```

  =>
  (assert (Treatment (treatment
pennicillin)))
  (printout t "Penicillin prescribed"
  crlf))

(defrule Allergy_pills
  (Diagnosis (diagnosis allergy))
  =>
  (assert (Treatment (treatment
allergy_shot)))
  (printout t "Allergy shot
prescribed" crlf))

(defrule Bed_rest
  (Diagnosis (diagnosis flu))
  =>
  (assert (Treatment (treatment
bed_rest)))
  (printout t "Bed rest prescribed"
  crlf))

; Facts are created with deffacts (the
can also
; be directly asserted while in
CLIPS). The list
; consists of a name, and a list of
facts.

(deffacts Symptoms
  (Patient (temperature 99)
  (spots no)
  (rash no)
  (sore_throat yes)
  (innoculated no)))

```

Д1.5. Приклад бази знань із введенням повідомлень

```

; This is a very simple example of a
CLIPS knowledge base,
; just using the pattern matching to
create new knowledge.

; We have modified this to use binding
to prompt users for input.

; To simplify this, we will no longer
use a single "Patient"
; object, but instead separate objects
for each symptom (note
; that we will only perform inference
on one patient at a time).

(deftemplate Temperature (slot
temperature))
(deftemplate Spots (slot spots))
(deftemplate Rash (slot rash))
(deftemplate SoreThroat (slot

```

```

sore_throat))
(deftemplate Innoculated (slot
innoculated))

; These deftemplates represent
conclusions which we may assign
; values to as a result of the
inference.

(deftemplate Diagnosis
  (slot diagnosis))

(deftemplate Treatment
  (slot treatment))

; We add another template for fever,
which is now a conclusion
; instead of a patient property

(deftemplate Fever
  (slot level))

; Our first rules will be used to
gather symptoms from the user.
; Note that there are no conditions,
which means that they will
; always fire. The action is to print
a prompt, bind the (read)
; to a variable, and then assert a new
fact using that value.

(defrule GetTemperature
  =>
  (printout t "Enter patient
temperature: ")
  (bind ?response (read))
  (assert (Temperature (temperature ?
response))))

(defrule GetSpots
  =>
  (printout t "Does the patient have
spots (yes or no): ")
  (bind ?response (read))
  (assert (Spots (spots ?response))))

(defrule GetRash
  =>
  (printout t "Does the patient have
a rash (yes or no): ")
  (bind ?response (read))
  (assert (Rash (rash ?response))))

(defrule GetSoreThroat
  =>
  (printout t "Does the patient have
a sore throat (yes or no): ")
  (bind ?response (read))
  (assert (SoreThroat (sore_throat ?
response))))

```

; We can also ask for certain
information only if necessary. For
example,
; it doesn't make sense to ask whether
the patient has been inoculated
; unless there is a possibility of
measles.

```

(defrule GetInnoculated
  (Fever (level high))
  (Spots (spots yes))
  =>
  (printout t "Has the patient been
innoculated for measles (yes or no):
")
  (bind ?response (read))
  (assert (Innoculated (innoculated ?
response))))

```

; Rules for concluding fever from
temperature.

; Note that these rules find the
patient temperature, and then bind
; it to ?t. The next part uses the
test keyword to evaluate the
; conditional expression as true or
false.

```

(defrule Fever1
  (Temperature (temperature ?t))
  (test (>= ?t 101))
  =>
  (assert (Fever (level high)))
  (printout t "High fever diagnosed"
crlf))

(defrule Fever2
  (Temperature (temperature ?t))
  (test (and (< ?t 101) (> ?t 98.6)))
  =>
  (assert (Fever (level mild)))
  (printout t "Mild fever diagnosed"
crlf))

```

; Rules for determining diagnosis on
the basis of patient symptoms
; Salience added to give this rule
priority

```

(defrule Measles
  (declare (salience 100))
  (Spots (spots yes))
  (Innoculated (innoculated no))
  (Fever (level high))
  =>
  (assert (Diagnosis (diagnosis
measles)))
  (printout t "Measles diagnosed"
crlf))

```

```

; Modified to only fire if no measles

(defrule Allergy1
  (declare (salience -100))
  (and (Spots (spots yes))
       (not (Diagnosis (diagnosis measles))))
  =>
  (assert (Diagnosis (diagnosis allergy)))
  (printout t "Allergy diagnosed from spots and lack of measles" crlf))

(defrule Allergy2
  (Rash (rash yes))
  =>
  (assert (Diagnosis (diagnosis allergy)))
  (printout t "Allergy diagnosed from rash" crlf))

(defrule Flu
  (SoreThroat (sore_throat yes))
  (Fever (level mild|high))
  =>
  (assert (Diagnosis (diagnosis flu)))
  (printout t "Flu diagnosed" crlf))

; Rules for recommending treatments on the basis of
; Diagnosis facts created.

(defrule Penicillin
  (Diagnosis (diagnosis measles))
  =>
  (assert (Treatment (treatment pennicillin)))
  (printout t "Penicillin prescribed" crlf))

(defrule Allergy_pills
  (Diagnosis (diagnosis allergy))
  =>
  (assert (Treatment (treatment allergy_shot)))
  (printout t "Allergy shot prescribed" crlf))

(defrule Bed_rest
  (Diagnosis (diagnosis flu))
  =>
  (assert (Treatment (treatment bed_rest)))
  (printout t "Bed rest prescribed" crlf))

; Facts are created with deffacts (the can also
; be directly asserted while in CLIPS). The list

```

```

; consists of a name, and a list of facts.

```

Д1.5. Приклад бази знань з простими фактами

```

; This is a very simple example of a CLIPS knowledge base,
; just using the pattern matching to create new knowledge.

```

```

; We have modified this to use binding to prompt users for input.

```

```

; To simplify this, we will no longer use a single "Patient"
; object, but instead separate objects for each symptom (note
; that we will only perform inference on one patient at a time).

```

```

; TO simplify this even further, since there are no longer any
; entities, we can just assert facts of the form (attribute value),
; which means that we will no longer need the deftemplates.

```

```

; Our first rules will be used to gather symptoms from the user.
; Note that there are no conditions, which means that they will
; always fire. The action is to print a prompt, bind the (read)
; to a variable, and then assert a new fact using that value.

```

```

(defrule GetTemperature
  =>
  (printout t "Enter patient temperature: ")
  (bind ?response (read))
  (assert (temperature ?response)))

```

```

(defrule GetSpots
  =>
  (printout t "Does the patient have spots (yes or no): ")
  (bind ?response (read))
  (assert (spots ?response)))

```

```

(defrule GetRash
  =>
  (printout t "Does the patient have a rash (yes or no): ")
  (bind ?response (read))
  (assert (rash ?response)))

```

```

(defrule GetSoreThroat
=>
  (printout t "Does the patient have
a sore throat (yes or no): ")
  (bind ?response (read))
  (assert (sore_throat ?response)))

; We can also ask for certain
information only if necessary. For
example,
; it doesn't make sense to ask whether
the patient has been inoculated
; unless there is a possibility of
measles.

(defrule GetInnoculated
  (fever high)
  (spots yes)
=>
  (printout t "Has the patient been
inoculated for measles (yes or no):
")
  (bind ?response (read))
  (assert (innoculated ?response)))

; Rules for concluding fever from
temperature.

; Note that these rules find the
patient temperature, and then bind
; it to ?t. The next part uses the
test keyword to evaluate the
; conditional expression as true or
false.

(defrule Fever1
  (temperature ?t)
  (test (>= ?t 101))
=>
  (assert (fever high))
  (printout t "High fever diagnosed"
crlf))

(defrule Fever2
  (temperature ?t)
  (test (and (< ?t 101) (> ?t 98.6)))
=>
  (assert (fever mild))
  (printout t "Mild fever diagnosed"
crlf))

; Rules for determining diagnosis on
the basis of patient symptoms
; Salience added to give this rule
priority

(defrule Measles
  (declare (salience 100))
  (spots yes)
  (innoculated no)

  (fever high)
=>
  (assert (diagnosis measles))
  (printout t "Measles diagnosed"
crlf))

; Modified to only fire if no measles

(defrule Allergy1
  (declare (salience -100))
  (and (spots yes)
      (not (diagnosis measles)))
=>
  (assert (diagnosis allergy))
  (printout t "Allergy diagnosed from
spots and lack of measles" crlf))

(defrule Allergy2
  (rash yes)
=>
  (assert (diagnosis allergy))
  (printout t "Allergy diagnosed from
rash" crlf))

(defrule Flu
  (sore_throat yes)
  (fever mild|high)
=>
  (assert (diagnosis flu))
  (printout t "Flu diagnosed" crlf))

; Rules for recommending treatments on
the basis of
; Diagnosis facts created.

(defrule Penicillin
  (diagnosis measles)
=>
  (assert (treatment penicillin))
  (printout t "Penicillin prescribed"
crlf))

(defrule Allergy_pills
  (diagnosis allergy)
=>
  (assert (treatment allergy_shot))
  (printout t "Allergy shot
prescribed" crlf))

(defrule Bed_rest
  (diagnosis flu)
=>
  (assert (treatment bed_rest))
  (printout t "Bed rest prescribed"
crlf))

; Finally, if there are no diagnosis
facts, we print the
; metaknowledge that the expert system
does not apply to
; this problem, and another expert

```

```

must be consulted. Note
; that the salience must be set to
make this the last rule
; checked.

(defrule None
  (declare (salience -100))
  (not (diagnosis ?))
  =>
  (printout t "No diagnosis possible
-- consult human expert" crlf))

```

Д1.6. Приклад бази знань з використанням відкату

```

; This example demonstrates how
knowledge can be retracted.
Specifically,
; if allergy is diagnosed, and measles
is diagnosed later, the initial
; diagnosis of allergy is retracted.

```

```

(deftemplate Fever
  (slot level))

; Our first rules will be used to
gather symptoms from the user.
; Note that there are no conditions,
which means that they will
; always fire. The action is to print
a prompt, bind the (read)
; to a variable, and then assert a new
fact using that value.

(defrule GetTemperature
  =>
  (printout t "Enter patient
temperature: ")
  (bind ?response (read))
  (assert (temperature ?response)))

(defrule GetSpots
  =>
  (printout t "Does the patient have
spots (yes or no): ")
  (bind ?response (read))
  (assert (spots ?response)))

(defrule GetRash
  =>
  (printout t "Does the patient have
a rash (yes or no): ")
  (bind ?response (read))
  (assert (rash ?response)))

(defrule GetSoreThroat
  =>
  (printout t "Does the patient have
a sore throat (yes or no): ")

```

```

  (bind ?response (read))
  (assert (sore_throat ?response)))

; We can also ask for certain
information only if necessary. For
example,
; it doesn't make sense to ask whether
the patient has been inoculated
; unless there is a possibility of
measles.

(defrule GetInnoculated
  (fever high)
  (spots yes)
  =>
  (printout t "Has the patient been
inoculated for measles (yes or no):
")
  (bind ?response (read))
  (assert (innoculated ?response)))

; Rules for concluding fever from
temperature.

; Note that these rules find the
patient temperature, and then bind
; it to ?t. The next part uses the
test keyword to evaluate the
; conditional expression as true or
false.

(defrule Fever1
  (temperature ?t)
  (test (>= ?t 101))
  =>
  (assert (fever high))
  (printout t "High fever diagnosed"
crlf))

(defrule Fever2
  (temperature ?t)
  (test (and (< ?t 101) (> ?t 98.6)))
  =>
  (assert (fever mild))
  (printout t "Mild fever diagnosed"
crlf))

; Rules for determining diagnosis on
the basis of patient symptoms
; Salience added to give this rule
priority

(defrule Measles
  (declare (salience 100))
  (spots yes)
  (innoculated no)
  (fever high)
  =>
  (assert (diagnosis measles))
  (printout t "Measles diagnosed"
crlf))

```

```

; Modified to only fire if no measles

(defrule Allergy1
  (declare (salience 200))
  (spots yes)
  =>
  (assert (diagnosis allergy))
  (printout t "Allergy diagnosed from
spots and lack of measles" crlf))

; This is the key rule. The first
thing we check for is whether measles
has been
; diagnosed. The second thing is
whether allergy has been diagnosed,
and that
; entire fact is bound to the variable
?allergyfact. We need to bind the fact
; to a variable in order to retract
that fact in the RHS of the rule.

(defrule CheckAllergy
  (declare (salience 100))
  (diagnosis measles)
  ?allergyfact <- (diagnosis allergy)
  =>
  (retract ?allergyfact)
  (printout t "Allergy diagnosis
being retracted due to measles" crlf))

(defrule Allergy2
  (rash yes)
  =>
  (assert (diagnosis allergy))
  (printout t "Allergy diagnosed from
rash" crlf))

(defrule Flu
  (sore_throat yes)
  (fever mild|high)
  =>
  (assert (diagnosis flu))
  (printout t "Flu diagnosed" crlf))

; Rules for recommending treatments on
the basis of
; Diagnosis facts created.

(defrule Penicillin
  (diagnosis measles)
  =>
  (assert (treatment penicillin))
  (printout t "Penicillin prescribed"
crlf))

(defrule Allergy_pills
  (diagnosis allergy)
  =>
  (assert (treatment allergy_shot))

```

```

(printout t "Allergy shot
prescribed" crlf))

(defrule Bed_rest
  (diagnosis flu)
  =>
  (assert (treatment bed_rest))
  (printout t "Bed rest prescribed"
crlf))

(defrule None
  (declare (salience -100))
  (not (diagnosis ?))
  =>
  (printout t "No diagnosis possible
-- consult human expert" crlf))

```

Д1.7. Приклад бази знань з можливістю модифікації

```

; This is a very simple example of a
CLIPS knowledge base,
; just using the pattern matching to
create new knowledge.

; We have modified this to use binding
to prompt users for input.

; In this example, instead of having
separate facts, we will have
; a single Patient template which
stores all information about
; the patient. We will then modify
this fact as knowledge is added.

; Note that (unless we do some other
things) this only allows one value
; per slot, which is why we will keep
diganosis and treatment separate.

(deftemplate Patient (slot
temperature)
                                (slot spots)
                                (slot rash)
                                (slot
sore_throat)
                                (slot
innoculated)
                                (slot fever))

; One thing that we will need is an
initial patient fact, since
; we will modify this instead of
creating new facts.

(deffacts Initial
  (Patient ))

; Our first rules will be used to

```

```
gather symptoms from the user.
; Note that there are no conditions,
which means that they will
; always fire. The action is to print
a prompt, bind the (read)
; to a variable, and then assert a new
fact using that value. Note
; that one of these tests is for
temperature to be nil, which means
; that temperature is not yet known
(if this is left out, the shell
; will loop).
```

```
; For each of these rules, we bind the
Patient fact to a variable
; and then modify it.
```

```
(defrule GetTemperature
  (declare (salience 500))
  ?p <- (Patient (temperature nil))
  =>
  (printout t "Enter patient
temperature: ")
  (bind ?response (read))
  (modify ?p (temperature ?
response)))
```

```
(defrule GetSpots
  (declare (salience 500))
  ?p <- (Patient (spots nil))
  =>
  (printout t "Does the patient have
spots (yes or no): ")
  (bind ?response (read))
  (modify ?p (spots ?response)))
```

```
(defrule GetRash
  (declare (salience 500))
  ?p <- (Patient (rash nil))
  =>
  (printout t "Does the patient have
a rash (yes or no): ")
  (bind ?response (read))
  (modify ?p (rash ?response)))
```

```
(defrule GetSoreThroat
  (declare (salience 500))
  ?p <- (Patient (sore_throat nil))
  =>
  (printout t "Does the patient have
a sore throat (yes or no): ")
  (bind ?response (read))
  (modify ?p (sore_throat ?
response)))
```

```
; We can also ask for certain
information only if necessary. For
example,
; it doesn't make sense to ask whether
the patient has been inoculated
; unless there is a possibility of
```

measles.

```
(defrule GetInnoculated
  (declare (salience 500))
  ?p <- (Patient (fever high) (spots
yes) (innoculated nil))
  =>
  (printout t "Has the patient been
innoculated for measles (yes or no):
")
  (bind ?response (read))
  (modify ?p (innoculated ?
response)))
```

```
; Rules for concluding fever from
temperature.
```

```
; Note that these rules find the
patient temperature, and then bind
; it to ?t. The next part uses the
test keyword to evaluate the
; conditional expression as true or
false.
```

```
; The numberp is a test to check
whether the argument is a numeric
value.
```

```
; This is placed before the numeric
comparison to keep the compiler from
; complaining that the initial patient
fact does not have a numeric
; temperature value.
```

```
(defrule Fever1
  ?p <- (Patient (temperature ?t)
(fever nil))
  (test (numberp ?t))
  (test (>= ?t 101))
  =>
  (modify ?p (fever high))
  (printout t "High fever diagnosed"
crlf))
```

```
(defrule Fever2
  ?p <- (Patient (temperature ?t)
(fever nil))
  (test (numberp ?t))
  (test (and (< ?t 101) (> ?t 98.6)))
  =>
  (modify ?p (fever mild))
  (printout t "Mild fever diagnosed"
crlf))
```

```
; Rules for determining diagnosis on
the basis of patient symptoms
; Salience added to give this rule
priority
```

```
(defrule Measles
  (declare (salience 100))
  (Patient (spots yes) (innoculated
```

```

no) (fever high))
=>
  (assert (diagnosis measles))
  (printout t "Measles diagnosed"
  crlf))

; Modified to only fire if no measles

(defrule Allergy1
  (declare (salience -100))
  (and (Patient (spots yes))
       (not (diagnosis measles)))
  =>
  (assert (diagnosis allergy))
  (printout t "Allergy diagnosed from
spots and lack of measles" crlf))

(defrule Allergy2
  (Patient (rash yes))
  =>
  (assert (diagnosis allergy))
  (printout t "Allergy diagnosed from
rash" crlf))

(defrule Flu
  (Patient (sore_throat yes) (fever
mild|high))
  =>
  (assert (diagnosis flu))
  (printout t "Flu diagnosed" crlf))

; Rules for recommending treatments on
the basis of
; Diagnosis facts created.

(defrule Penicillin
  (diagnosis measles)
  =>
  (assert (treatment pennicillin))

  (printout t "Penicillin prescribed"
  crlf))

(defrule Allergy_pills
  (diagnosis allergy)
  =>
  (assert (treatment allergy_shot))
  (printout t "Allergy shot
prescribed" crlf))

(defrule Bed_rest
  (diagnosis flu)
  =>
  (assert (treatment bed_rest))
  (printout t "Bed rest prescribed"
  crlf))

; Finally, if there are no diagnosis
facts, we print the
; metaknowledge that the expert system
does not apply to
; this problem, and another expert
must be consulted. Note
; that the salience must be set to
make this the last rule
; checked.

(defrule None
  (declare (salience -100))
  (not (diagnosis ?))
  =>
  (printout t "No diagnosis possible
-- consult human expert" crlf))

```

Література

1. Abelson H., Sussman G. J. and Sussman J. (1996). *Structure and Interpretation of Computer Programs*. Cambridge MA: MIT Press.
2. Abu-Hakima S., Halasz M. and Phan S. (1993). An approach to hypermedia in diagnostic systems. In *Intelligent Multi-Media Interfaces* (Maybury M., eds.). AAAI Press/MIT Press.
3. Acorn T. L. and Walden S. (1992). SMART: Support Management Automated Reasoning Technology for Compaq customer service. In *Proc. Innovative Applications of Artificial Intelligence* (IAAI-92), p. 3-18.
4. Adams J. B. (1976). A probability model of medical reasoning and the MYCIN model. *Mathematical Biosciences*, 32, p. 177-186. See also Buchanan and Shortliffe (1984), Chapter 12.
5. Aiello N. (1983). A comparative study of control strategies for expert systems: AGE implementation of three variations of PUFF. In *Proc. National Conference on Artificial Intelligence*, p. 1-4.
6. Aiello N. (1986). User-Directed Control of Parallelism: The CAGE System. *Technical Report No. KSL-86-31*, Knowledge Systems Laboratory, Stanford University.
7. Aikins J. S. (1983). Prototypical knowledge for expert systems. *Artificial Intelligence*, 10, p. 163-210.
8. Aikins J. S., Kunz J. C., Shortliffe E. H. and Fallat R. J. (1984). PUFF: an expert system for interpretation of pulmonary function data. In *Readings in Medical Artificial Intelligence* (Clancey W. J. and Shortliffe E. H., eds.). Chapter 19. Reading, MA: Addison-Wesley.
9. Aleven V. and Ashley K. D. (1997). Evaluating A Learning Environment For Case-Based Argumentation Skills. In *Proc. Sixth International Conference on Artificial Intelligence and Law*, ACM Press, p. 170.
10. Alexander J. H., Freiling M. J., Shulman S. J., Staley J. L., Rehfuss S. and Messick S. L. (1986). Knowledge level engineering: ontological analysis. In *Proc. National Conference on Artificial Intelligence*, p. 963-968.
11. Allen B. (1994). Case-based reasoning: business applications. *Communications of the ACM*, 37(3), p. 40².
12. Allen J.F. (1983). Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11), p. 832-843.
13. Allen J. F. (1995). *Natural Language Understanding*, 2nd edn. Menlo Park, CA: Benjamin/Cummings.
14. Amarel S. (1968). On representations of problems of reasoning about actions. In *Machine Intelligence 3* (Michie D., eds.), p. 131-171. Edinburgh: Edinburgh University Press.
15. Anderson J. R. (1976). *Language, Memory and Thought*. Hillsdale NJ: Lawrence Erlbaum.
16. Andrews P. B. (1986). *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Orlando FL, Academic Press.
17. Ashley K. D. (1990). *Modeling Legal Argument: Reasoning With Cases and Hypotheticals*. Cambridge MA: MIT Press.
18. Ashley K. and Aleven V. (1997). Reasoning symbolically about partially matched cases. *International Joint Conference on Artificial Intelligence, IJCAI-1997*. San Francisco: Morgan Kaufmann, p. 335-341.
19. Ashley K. D. and Rissland E. L. (1988). A case-based approach to modelling legal expertise. *IEEE Expert*, 3(3), p. 70-77.

20. Bachant J. (1988). RIME: preliminary work towards a knowledge acquisition tool. In *Automating Knowledge Acquisition for Expert Systems* (Marcus S., eds.), Chapter 7. Boston: Kluiver Academic.
21. Baldwin J. E, eds. (1996). *Fuzzy Logic*. New York: Wiley.
22. Barr A. and Feigenbaum E. A., eds. (1981). *The Handbook of Artificial Intelligence* Vol. 1. Los Altos CA: Morgan Kaufmann.
23. Barr A. and Feigenbaum E. A., eds. (1982). *The Handbook of Artificial Intelligence* Vol. 2. Los Altos CA: Morgan Kaufmann.
24. Baum L. S., Dodhiawala R. T. and Jagannathan V. (1989). The Erasmus System. In *Blackboard Architectures and Applications* (Jagannathan V, Dodhiawala R. and Baum L., eds.), p. 347-370. New York: Academic Press.
25. Belew R. K. (1986). Adaptive information retrieval: machine learning in associative networks. *PhD thesis*, University of Michigan, Computer Science Department, Ann Arbor, MI.
26. Bennett J. S., Creary L., Englemore R. and Melosh R. (1978). SACON: A Knowledge Based Consultant for Structural Analysis. *Report No. HPP-78-23*, Computer Science Department, Stanford University.
27. Bergadano F. and Gunetti D. (1996). *Inductive Logic Programming: From Machine Learning to Software Engineering*. Cambridge MA: MIT Press.
28. Biswas G. and Anand T. S. (1987). MIDST: an expert system shell for mixed initiative reasoning. In *Proc. International Symposium on Methodologies for Intelligent Systems*, p. 1-8.
29. Bobrow D. G. and Collins A., eds. (1975). *Representation and Understanding*. New York: Academic Press.
30. Bobrow D. G. and Stefik M. (1983). *The LOOPS manual*. Xerox Corporation.
31. Bobrow D. G. and Winograd T. (1977). An overview of KRL, a knowledge representation language. *Cognitive Science*, 1(1).
32. Bobrow D. G. and Winograd T. (1979). KRL: another perspective. *Cognitive Science*, 3(1).
33. Booch G. (1994). *Object-Oriented Analysis and Design with Applications*, 2nd edn. Redwood City CA: Benjamin/Cummings. (Русский перевод первого издания: Буч Г. Объектно-ориентированное проектирование с примерами применения. — М.: Конкорд, 1992.)
34. Boose J. H. (1986). *Expertise Transfer for Expert System Design*. New York: Elsevier.
35. Boose J. H. (1989). A survey of knowledge acquisition techniques and tools. *Knowledge Acquisition*, 1(1), p. 3-37.
36. Boose J. H. and Bradshaw J. M. (1987). Expertise transfer and complex problems: using AQUINAS as a workbench for knowledge based systems. *International Journal of Man-Machine Studies*, 26, p. 1-28.
37. Boose J. H. and Gaines B. (1988). *Knowledge Acquisition Tools for Expert Systems*. New York: Academic Press.
38. Borron J., Morales D. and Klahr P. (1996). Developing and deploying knowledge on a global scale. In *Innovative Applications of Artificial Intelligence 8, Proceedings of AAAI-96*. Menlo Park CA: AAAI Press.
39. Brachman R. J. and Levesque H. J. (1985). *Readings in Knowledge Representation*. Los Altos CA: Morgan Kaufmann.

40. Brachman R. J. and Schmolze J. G. (1985). An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9, p. 171-216.
41. Bratko I. (1990). *PROLOG Programming for Artificial Intelligence*, 2nd eds. Wokingham, UK: Addison-Wesley. (Русский перевод первого издания: Братко И. Программирование на языке PROLOG для искусственного интеллекта. — М.: Мир, 1990. — 559 с.)
42. Breuker J. A. and van de Velde W., eds. (1994). *The CommonKADS Library for Expertise Modelling*. Amsterdam: IOS Press.
43. Brown D. C. and Chandrasekaran B. (1989). *Design Problem Solving: Knowledge Structures and Control Strategies*. Los Altos CA: Morgan Kaufmann.
44. Brown J. S., Burton R. R. and De Kleer J. (1982). Pedagogical, natural language and knowledge engineering techniques in SOPHIE I, II and III. In *Intelligent Tutoring Systems* (Sleeman D. and Brown J. S., eds.), Chapter 11. London: Academic Press.
45. Brownston L., Farrell R., Kant F. and Martin N. (1985). *Programming Expert Systems in OPS5*. Reading MA: Addison-Wesley.
46. Buchanan B. G. and Feigenbaum E. A. (1978). DENDRAL and META-DENDRAL: their applications dimension. *Artificial Intelligence*, 11, p. 5-24.
47. Buchanan B. G. and Shortliffe E. H., eds. (1984). *Rule-Based Expert Systems*. Reading MA: Addison-Wesley.
48. Buchanan B. G. and Wilkins D. C., eds. (1993). *Readings in Knowledge Acquisition and Learning*. Los Altos, CA: Morgan Kaufmann.
49. Buchanan B. G., Barstow D., Bechtel R., Bennet J., Clancey W., Kulikowski C., Mitchell T. M. and Waterman D. A. (1983). Constructing an expert system. In *Building Expert Systems* (Hayes-Roth F, Waterman D.A. and Levat D., eds.), Chapter 5. Reading, MA: Addison-Wesley.
50. Bundy A. (1978). Will it reach the top? Prediction in the mechanics world. *Artificial Intelligence*, 10, p. 129-146.
51. Bundy A., Byrd L., Luger G., Mellish C. and Palmer M. (1979). Solving mechanics problems using metalevel inference. In *Expert Systems in the Micro-Electronic Age* (Michie D., eds.), p. 50-64. Edinburgh: Edinburgh University Press.
52. Bundy A., Silver B. and Plummer D. (1985). An analytical comparison of some rule-learning programs. *Artificial Intelligence*, 27, p. 137-181.
53. Buntine W. L. (1990). Myths and legends in learning classification rules. In *Proc. National Conference on Artificial Intelligence*, p. 736-742.
54. Bylander T. (1994). The computational complexity of prepositional STRIPS planning. *Artificial Intelligence*, 69, p. 165-204.
55. Cannon H. I. (1982). FLAVORS: a non-hierarchical approach to object-oriented programming. Unpublished paper.
56. Carbonell J. G., Michalski R. and Mitchell T. (1983). An overview of machine learning. In *Machine Learning* (Michalski R. S., Carbonell J. S. and Mitchell T. M., eds.), Chapter 1. Palo Alto, CA: Tioga.
57. Carver N. and Lesser V. (1994). The evolution of blackboard control architectures. In *Expert Systems with Applications: Special Issue on the Blackboard Paradigm and its Applications*, 7(1), p. 1-30.
58. Cendrowska J. and Bramer M. (1984). Inside an expert system: A rational reconstruction of the MYCIN consultation system. In *Artificial Intelligence: Tools, Techniques and Applications* (O'Shea T. and Eisenstadt M., eds.). Chapter 15. New York: Harper and Row.

59. Chandrasekaran B. (1983). Towards a taxonomy of Problem solving types. *AI Magazine*, 4(1), Spring, p. 9-17.
60. Chandrasekaran B. (1984). Expert systems: matching techniques to tasks. In *Artificial Intelligence Applications for Business* (Reitman W, eds.). Norwood, NJ: Ablex.
61. Chandrasekaran B. (1986). Generic tasks in knowledge-based reasoning: high-level building blocks for expert systems design. *IEEE Expert*, 1(3), p. 23-30.
62. Chandrasekaran B. (1988). Generic tasks as building blocks for knowledge-based systems: the diagnosis and routine design examples. *Knowledge Engineering Review*, 3(3), p. 183-210.
63. Chandrasekaran B. and Mittal S. (1984). Deep versus compiled knowledge approaches to diagnostic problem solving. In *Developments in Expert Systems* (Coombs M. J., eds.), Chapter 2. London; Academic Press.
64. Chappell D. (1996). *Understanding ActiveX and Ole*. Microsoft Press.
65. Charniak E. and McDermott D. (1985). *Introduction to Artificial Intelligence*. Reading, MA: Addison-Wesley.
66. Charniak E., Reisbeck C. and McDermott D. (1987). *Artificial Intelligence Programming*, 2nd ed. Hillsdale, NJ: Lawrence Erlbaum.
67. Cheeseman P. (1985). In defense of Probability. In *Proc. 8th International Joint Conference on Artificial Intelligence*, p. 1002-1009.
68. Church A. (1941). *The Calculi of Lambda Conversion*. Annals of Mathematics Studies, Princeton University Press.
69. Clancey W. J. (1983). The epistemology of a rule-based expert system: a framework for explanation. *Artificial Intelligence*, 10, p. 215-251.
70. Clancey W. J. (1985). Heuristic classification. *Artificial Intelligence*, 27, p. 289-350.
71. Clancey W. J. (1987, a). *Knowledge-Based Tutoring, The GUIDON Program*. Cambridge, MA: MIT Press.
72. Clancey W. J. (1987, b). Intelligent tutoring systems: a tutorial survey. In van Lamsweerde and Dufour (1987), Chapter 3.
73. Clancey W. J. (1987, c). From GUIDON to NEOMYCIN and HERACLES in twenty short lessons, In *Current Issues in Expert Systems* (van Lamsweerde A. and Dufour P., eds., Chapter 4. London: Academic Press.
74. Clancey W. J. (1993, a). Notes on "Heuristic Classification". *Artificial Intelligence*, 59, p. 191-196.
75. Clancey W. J. (1993, b). Notes on "Epistemology of a rule-based expert system". *Artificial Intelligence*, 59, p. 197-204.
76. Clancey W. J. and Letsinger R. (1984). NEOMYCIN: reconfiguring a rule-based expert system for application to teaching. In *Readings in Medical Artificial Intelligence* (Clancey W. J. and Shortliffe E. H., eds.). Chapter 15. Reading, MA: Addison-Wesley.
77. Clancey W. J. and Shortliffe E. H., eds. (1984). *Readings in Medical Artificial Intelligence*. Reading, MA: Addison-Wesley.
78. Clark K. L. and McCabe F. (1982). PROLOG: a language for implementing expert systems. In *Machine Intelligence 10* (Hayes J. E., Michie D. and Pao Y. H., eds.). Chichester, UK: Ellis Horwood.
79. Clark K. L. and Tarnlund S.-A., eds. (1982). *Logic Programming*, London: Academic Press
80. Cohen P. and Feigenbaum E. A., eds. (1982). *The Handbook of Artificial Intelligence, Vol. 3*.

Los Altos, CA: Morgan Kaufmann.

81. Collins A. M. and Quillian M. R. (1969). Retrieval time from semantic memory. *Journal of Verbal Learning and Verbal Behavior*, 8, p. 240-247.
82. Coombs M. J., eds. (1984). *Developments in Expert Systems*. London: Academic Press.
83. Cooper G. F. (1990). The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence*, 42, p. 393-405.
84. Corkill D. D. (1991). Blackboard systems. *AI Expert*, 6(9), p. 40-47. Coyne R, (1988). *Logic Models of Design*. London: Pitman.
85. Cox R. (1946). Probability frequency and reasonable expectation. *American Journal of Physics*, 14(1), p. 1-13.
86. Craig I. (1995). *Blackboard Systems*. Norwood, NJ: Ablex.
87. Davies J. and May R. (1995). The development of a prototype "correctly dressed" case-based reasoner: Efficacy of CBR-Express. In *Progress in Case-Based Reasoning* (Watson I., eds.). Lecture Notes in Artificial Intelligence, 1020. Berlin: Springer-Verlag.
88. Davis R. (1980, a). Meta-rules: reasoning about control. *Artificial Intelligence*, 15, p. 179-222.
89. Davis R. (1980, b). Applications of meta-level knowledge to the construction, maintenance and use of large knowledge bases. In *Knowledge-Based Systems in Artificial Intelligence* (Davis R. and Lenat D., eds.) p. 229-490. New York: McGraw-Hill.
90. Davis R. (1982). Expert systems: Where are we? And where do we go from here? *AI Magazine*, 3(2).
91. Davis R. (1984). Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24, p. 347-410.
92. Davis R. (1989). Expert systems: how far can they go? Part Two. *AI Magazine*, 10(2), Summer 1989, p. 65-77.
93. Davis R. and King J. (1977). An overview of production systems. In *Machine Intelligence 8* (Elcock E. W. and Michie D., eds.), p. 300-332. New York: Wiley.
94. Davis R. and Lenat D. (1980). *Knowledge-Based Systems in Artificial Intelligence*. New York: McGraw-Hill.
95. De Kleer J. (1986). An assumption based TMS. *Artificial Intelligence*, 28, p. 127-162.
96. De Kleer J. and Williams B. C. (1987). Diagnosing multiple faults. *Artificial Intelligence*, 32, p. 97-130.
97. Dean T. L. and Wellman M. (1991). *Planning and Control*. Los Altos, CA: Morgan Kaufmann.
98. Dietterich T. G. and Michalski R. (1983). A comparative review of selected methods for learning 'from examples. In *Machine Learning* (Michalski R. S., Carbonell J. G. and Mitchell T. M., eds.), Chapter 3. Palo Alto, CA: Tioga.
99. Doyle J. (1979). A truth maintenance system. *Artificial Intelligence*, 12, p. 231-272.
100. Dubois D., Prade H. and Yager R. R., eds. (1996). *Fuzzy Information Engineering: A Guided Tour of Applications*. New York: Wiley.
101. Elcock E. W. and Michie D., eds. (1977). *Machine Intelligence 8*. New York: Wiley.
102. Englemore R. and Morgan T., eds. (1988). *Blackboard Systems*. Reading, MA: Addison-Wesley.
103. Eriksson H., Shahar T., Tu S. W., Puerta A. R. and Musen M. A. (1995). Task Modeling

- with reusable problem-solving methods. *Artificial Intelligence*, 79 (2), p. 293-326.
104. Erman L., Hayes-Roth R., Lesser V. and Reddy D. (1980). The HEARSAY-II speech understanding system: integrating knowledge to resolve uncertainty. *Computing Surveys*, 12(2), p. 213-253.
 105. Erman L. D., London P. E. and Fickas S. F. (1983). The design and an example use of HEARSAY-III, *In Proc. National Conference on Artificial Intelligence*, p. 409^415.
 106. Eshelman L. (1988). A knowledge acquisition tool for cover-and-differentiate systems. In *Automating Knowledge Acquisition for Expert Systems* (Marcus S., eds.), Chapter 3. Boston: Kluiver Academic.
 107. Eshelman L. and McDermott J. (1986). MOLE: a knowledge acquisition tool that uses its head. In *Proc. National Conference on Artificial Intelligence*, p. 950-955.
 108. Eshelman L., Ehret D., McDermott J. and Tan M. (1987). MOLE: a tenacious knowledge acquisition tool. *International Journal of Man-Machine Studies*, 26, p. 41-54.
 109. Feigenbaum E. A. (1977). The art of artificial intelligence: themes and case studies of knowledge engineering. In *Proc. 5th International Joint Conference on Artificial Intelligence*, p. 1014-1029.
 110. Feigenbaum E.A. and Buchanan B.G. (1993). DENDRAL and Meta-DENDRAL: roots of knowledge systems and expert systems applications. *Artificial Intelligence*, 59, p. 233-240.
 111. Feigenbaum E. A. and Feldman J., eds. (1963). *Computers and Thought*. New York: McGraw-Hill.
 112. Feigenbaum E. A., McCorduck P. and Nii H. P. (1988). *The Rise of the Expert Company*. Times Books.
 113. Feiner S. K. and McKeown K. R. (1991). Automating the generation of coordinated multimedia explanations. In *Intelligent Multi-Media Interfaces* (Maybury M., eds.), Chapter 5. AAAI Press/MIT Press.
 114. Fikes R. E. and Nilsson N. J. (1971). STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2, p. 189-208. (Русский перевод: Файкс Р., Нильсон Н. Система STRIPS — новый подход к применению методов доказательства теорем при решении задач.— В сб. Интегральные роботы, вып. 1. Под. ред. Г. Е. Поздняка. — с. 382^103.)
 115. Findler N. V., eds. (1979). *Associative Networks*. New York: Academic Press.
 116. Forbus K. and de Kleer J. (1993). *Building Problem Solvers*. Cambridge, MA: MIT Press.
 117. Forge C. L. (1982). Rete: a fast algorithm for the many pattern/many object pattern match
 118. *problem. Artificial Intelligence*, 19, p. 17-37.
 119. Gale W. A. (1986). *Artificial Intelligence and Statistics*. Reading, MA: Addison-Wesley.
 - Gale W. A. (1987). Knowledge-based knowledge acquisition for a statistical consulting system.
 120. *International Journal of Man-Machine Studies*, 26, p. 54-64.
 121. Genesereth M. R. (1984). The use of design descriptions in automated diagnosis. *Artificial Intelligence*, 24, p. 411-436. Genesereth M. R. and Nilsson N. J. (1987). *Logical Foundations of Artificial Intelligence*. Los
 122. Altos, CA: Morgan Kaufmann. Giarratano J. and Riley G. (1994). *Expert Systems: Principles and Programming*, 2nd edn.
 123. Boston, MA: PWS Publishing.
 124. Gil Y. and Paris C. L. (1994). Towards method-independent knowledge acquisition.

- Knowledge Acquisition*, 6(2), p. 163-178. Ginsberg M., eds. (1987). *Readings in Nonmonotonic Reasoning*. Los Altos, CA: Morgan
125. Kaufmann.
 126. Ginsberg M. (1993). *Essentials of Artificial Intelligence*. Los Altos, CA: Morgan Kaufmann. Givan R. and Dean T. (1997). Model minimization, regression, and propositional STRIPS planning. In *Proc. 15th International Joint Conference on Artificial Intelligence*, p. 1163-1168. Glaser H., Hankin C. and Till D. (1984). *Functional Programming*. Englewood Cliffs, NJ:
 127. Prentice-Hall. Goldberg A. and Robson D. (1983). *Smalltalk-80; The Language and its Implementation*.
 128. Reading, MA: Addison-Wesley. Golding A. R. and Rosenbloom P. S. (1991). Improving rule-based systems through case-based
 129. reasoning. In *Proc. National Conference on Artificial Intelligence*, p. 22-21. Golding A. R. and Rosenbloom P. S. (1996). Improving accuracy by combining rule-based and
 130. case-based reasoning. *Artificial Intelligence*, 87, p. 215-254. Goldman N. (1978). *AP3 User's Guide*. Information Sciences Institute, University of Southern
 131. California.
 132. Gordon J. and Shortliffe E. H. (1985). A method of managing evidential reasoning in a hierarchical hypothesis space. *Artificial Intelligence*, 26, p. 323-357. Graham P. (1994). *On Lisp: Advanced Techniques for Common Lisp*. Englewood Cliffs, NJ:
 133. Prentice-Hall.
 134. Haberi J. S., Norford L. K. and Spadaro G. V. (1989). Diagnosing building operational problems. *ASHRAE Journal*, June, p. 20-30.
 135. Hammond K. J. (1986). CHEF: a model of case-based planning. In *Proc. AAAI-86* Harmon P. (1992). Case-based reasoning III. *Intelligent Software Strategies*, 8(1). Harmon P. and Hall C. (1993). *Intelligent Software Systems Development: An IS Manager's Guide*. New York: Wiley.
 136. Harmon P. and Sawyer B. (1990). *Creating Expert Systems for Business and Industry*. New York: Wiley.
 137. Hasling D. W., Clancey W. J. and Rennels G. (1984). Strategic explanations for a diagnostic consulting system. *International Journal of Man-Machine Studies*, 20(1), p. 3-19.
 138. Hayes J. E. and Michie D., eds. (1984). *Intelligent Systems - The Unprecedented Opportunity*. Chichester, UK: Ellis Horwood.
 139. Hayes J. E., Michie D. and Pao Y. H., eds. (1982). *Machine Intelligence 10*. Chichester, UK: Ellis Horwood.
 140. Hayes-Roth B. (1985). Blackboard architecture for control. *Artificial Intelligence*, 26, p. 251-321.
 141. Hayes-Roth F., Waterman D. A. and Lenat D., eds. (1983). *Building Expert Systems*. Reading, MA: Addison-Wesley. (Русский перевод: Хейес-Рот Ф., Уотерман Д., Ленат Д. Построение экспертных систем. — М.: Мир, 1987. — 430 с.)
 142. Hayes-Roth B., Buchanan B., Lichtarge O., Hewett M., Altaian R., Brinkiey J., Cornelius C., Duncan B. and Jardetzky O. (1986). PROTEAN: deriving protein structure from constraints. In *Proc. National Conference on Artificial Intelligence*, p. 904-909.
 143. Hayes-Roth B., Garvey A., Johnson M. V. and Hewett H. (1987). A Modular and Layered Environment for Reasoning About Action. *Technical Report No. KSL 86-38*, Knowledge Systems Laboratory, Stanford University.

144. Hayes-Roth B., Johnson M. V., Garvey A. and Hewett H. (1988). Building systems in the BB* environment. In *Blackboard Systems* (Englemore R. and Morgan T., eds.), Chapter 29. Reading, MA: Addison-Wesley.
145. Heckerman D. (1986). Probabilistic interpretation for MYCIN's certainty factors. In *Uncertainty in Artificial Intelligence* (Kanal L. N. and Lemmer J. E., eds.), p. 169-196. Amsterdam: North-Holland.
146. Henderson P. (1980). *Functional Programming: Application and Implementation*. Englewood Cliffs, NJ: Prentice-Hall.
147. Hendrix G. G. (1979). Encoding knowledge in partitioned networks. In *Associative Networks* (Findler N. V., eds.), p. 51-92. New York: Academic Press.
148. Henrion M., Shachter R. D., Kanal L. N. and Lemmer J. E., eds., (1990). *Uncertainty in Artificial Intelligence 5*. Amsterdam: North-Holland.
149. Hewitt C. (1972): Description and Theoretical Analysis (Using Schemata) of PLANNER, a Language for Proving Theorems and Manipulating Models in a Robot. *Report No. TR-258*, AI Laboratory, MIT.
150. Hinton G. E. (1989). Connectionist learning procedures. *Artificial Intelligence*, 40, p. 185-234.
151. Hislop C. and Pracht D. (1994). Integrated problem resolution for business communications. In *Innovative Applications of Artificial Intelligence 6, Proceedings of AAAI-94*, p. 63-74, Memo Park, CA: AAAI Press.
152. Hopcroft J. E. and Ullman J. D. (1979). *Introduction to Automata Theory, Languages and Computation*. Reading, MA: Addison-Wesley.
153. Horty J. R., Thomason R. H. and Touretzky D. S. (1987). A skeptical theory of inheritance in nonmonotonic semantic nets. In *Proc. National Conference on Artificial Intelligence*, p. 358-363.
154. Horvitz E. and Heckerman D. (1986). The inconsistent use of measures of certainty in artificial intelligence research. In *Uncertainty in Artificial Intelligence* (Kanal L. N. and Lemmer J. R., eds.), p. 137-151. Amsterdam: North Holland.
155. Horvitz E., Heckerman D. and Langlotz C. P. (1986). A framework for comparing formalisms for plausible reasoning. In *Proc. National Conference on Artificial Intelligence*, p. 210-214.
156. Hunt E. B., Mann I. and Stone P. T. (1966). *Experiments in Induction*. New York: Academic Press.
157. Jagannathan V., Dodhiawala R. and Baum L. S., eds. (1989). *Blackboard Architectures and Applications*. New York: Academic Press.
158. Jamshidi M., Tilti A., Zadeh L. and Boverie S., eds. (1997). *Applications of Fuzzy Logic: Towards High Machine Intelligence Quotient Systems*. Englewood Cliffs, NJ: Prentice Hall.
159. Jensen R. V. (1996). *Introduction to Bayesian Networks*. Berlin: Springer Verlag.
160. Johnson M. V. and Hayes-Roth B. (1986). Integrating Diverse Reasoning Methods in the BB1 Blackboard Control Architecture. *Technical Report No. KSL 86-76*, "Knowledge Systems Laboratory, Stanford University.
161. Kahn G. (1988). MORE: from observing knowledge engineers to automating knowledge acquisition. In *Automating Knowledge Acquisition for Expert Systems* (Marcus S., eds.), Chapter 2. Boston: Kluiver Academic.
162. Kahn G. and McDermott J. (1984). The MUD system. In *Proc. 1st IEEE Conference on Artificial Intelligence Applications*.

163. Kahn G., Nowlan S. and McDermott J. (1985). MORE; an intelligent knowledge acquisition tool. In *Proc. 9th International Joint Conference on Artificial Intelligence*, p. 581-584.
164. Kahn G., Kepner A. and Pepper J. (1987). TEST: a model-driven application shell. In *Proc. National Conference on Artificial Intelligence*, p. 814-818.
165. Kahneman D. and Tversky A. (1972). Subjective probability: a judgement of representativeness. *Cognitive Psychology*, 3, p. 430-454.
166. Kahneman D., Slovic P. and Tversky A., eds. (1982). *Judgement under Uncertainty: Heuristics and Biases*. Cambridge: Cambridge University Press.
167. Kam M., Chow J.-C. and Fischl R. (1991). Hybrid expert system/neural network architecture for classifying power system contingencies. In *Proc. First International Forum on Applications of Neural Networks to Power Systems*, p. 76-82.
168. Kanal L. N. and Lemmer J. R., eds. (1986). *Uncertainty in Artificial Intelligence*. Amsterdam: North-Holland.
169. Keene S. E. (1989). *Object-Oriented Programming in COMMON LISP*. Reading, MA: Addison-Wesley.
170. Kingston J. K. C. (1997). Designing knowledge based systems: The CommonKADS design model. In *Research and Development in Expert Systems XIV, Proceedings of BCS SGES, Expert Systems '97*, Cambridge University Press.
171. Kingston J. K. C., Doherty J. G. and Filby I. M. (1995). Evaluation of workbenches which support the CommonKADS methodology. *Knowledge Engineering Review*, 10, 3.
172. Klahr P. and Waterman D., eds. (1986). *Expert Systems: Techniques, Tools, and Applications*. Reading, MA: Addison-Wesley.
173. Kodratoff Y. and Michalski R. S., eds. (1990). *Machine Learning: An Artificial Intelligence Approach*, Vol. III. Los Altos, CA: Morgan Kaufmann.
174. Kolodner J. L. (1991). Improving human decision making through case-based decision aiding. *AI Magazine*, 12 (2), p. 52-68.
175. Kolodner J. L. (1993). *Case-Based Reasoning*. Los Altos, CA: Morgan Kaufmann.
176. Kononenko I., Bratko I. and Riskar E. (1984). *Experiments in Automatic Learning of Medical Diagnostic Rules*. Jozef Stefan Institute, Ljubljana, Yugoslavia.
177. Korf R. E. and Reid M. (1998). Complexity analysis of admissible heuristic search. In *Proc. 15th National Conference on Artificial Intelligence (AAAI-98)*. p. 305-310, AAAI Press.
178. Kowalski R. A. (1979). *Logic for Problem Solving*. Amsterdam: North-Holland.
179. Kowalski R. A. (1982). Logic as a computer language. In *Logic Programming* (Clark K. L. and Tarnlund S. -A., eds.) Chapter 1. London: Academic Press.
180. Kunz J. C., Fallal R. J., McClung D. H., Osborn J. J., Votteri R. A., Nii H. P., Aikins J. S., Fa-gan L. M. and Feigenbaum E. A. (1978). A Physiological Rule-Based System for Interpreting Pulmonary Function Test Results. *Report No. HPP 78-19*, Heuristic Programming Project, Computer Science Department, Stanford University.
181. Kunz J. C., Kehler T. P. and Williams M. D. (1984). Applications development using a hybrid AI development system. *AI Magazine*, 5(3), Fall, p. 41-54.
182. Lan M. S., Panos R. M. and Balban M. S. (1987). Experience using S.I: an expert system for newspaper printing press configuration. *Knowledge Engineering Review*, 2(4), p. 277-285.

183. Langlotz C. P., Shortliffe E. H. and Fagan L. M. (1986). Using decision theory to justify heuristics. In *Proc. National Conference on Artificial Intelligence*, p. 215-219.
184. Laurent J. P., Ayel J., Thome F. and Ziebelin D. (1986). Comparative evaluation of three expert system development tools: KEE, Knowledge Craft, ART. *Knowledge Engineering Review*, 1(4), p. 19-29.
185. Lehnert W. and Wilks Y. (1979). A critical perspective on KRL *Cognitive Science*, 3, p. 1-28. Lenat D. B. (1982). The nature of heuristics. *Artificial Intelligence*, 19, p. 189-249.
186. Levesque H. and Mylopoulos J. (1979). A procedural semantics for semantic networks. In *Associative Networks* (Findler N. V., eds.), p. 93-120. New York: Academic Press.
187. Linster M. and Musen M. A. (1992). Use of KADS to create a conceptual model of the
188. ONCOCIN task. *Knowledge Acquisition*, 4, p. 55-87. MacGregor R. (1991). Inside the LOOM classifier. *SIGART Bulletin*, 2(3), p. 70-76.
189. Mamdani E. H. and Gaines B. R. (1981). *Fuzzy Reasoning and its Applications*. London: Academic Press.
190. Marcus S., eds. (1988, a). *Automating Knowledge Acquisition for Expert Systems*. Boston: Kluiver Academic.
191. Marcus S. (1988, b). A knowledge acquisition tool for propose-and-revise systems. In *Automating Knowledge Acquisition for Expert Systems* (Marcus S., eds.). Chapter 4. Boston: Kluiver Academic.
192. Marcus S., Stout J. and McDermott J. (1988). VT: an expert elevator configurer that uses knowledge-based backtracking. *AI Magazine*, 9(1), p. 95-112.
193. Marir F. and Watson I. (1994). Case-based reasoning: a categorized bibliography. *Knowledge Engineering Review*, 9(4), p. 355-381.
194. Martins J. P. and Reinfrank M. (1991). *Truth Maintenance Systems*. Lecture Notes in Artificial Intelligence, No. 515. Berlin: Springer Verlag.
195. Maybury M., eds. (1993). *Intelligent Multi-Media Interfaces*. Cambridge, MA: AAAI Press/MIT Press.
196. Press/MIT Press.
197. McAllester D. (1980). An Outlook on Truth Maintenance. *Report No. AIM-551*, Artificial Intelligence Laboratory, Massachusetts Institute of Technology.
198. McArthur D., Klahr P. and Narain S. (1986). ROSS: an object-oriented language for constructing simulations. In *Expert Systems: Techniques, Tools and Applications* (Klahr P. and Waterman D., eds.). Chapter 3. Reading, MA: Addison-Wesley.
199. McCarthy J. (1960). Recursive functions of symbolic expressions and their computation by machine. *Communications of the Association for Computing Machinery*, April, p. 184-195. McCarthy J. and Hayes P. (1969). Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4* (Meltzer B. and Michie D., eds.), p. 463-502. Edinburgh: Edinburgh University Press.
200. McCarthy J., Abrahams P. W., Edwards D. I., Hart T. P. and Levin M. I. (1965). *LISP 1.5 Programmer's Manual*, 2nd edn. Cambridge, MA: MIT Press.
201. McDermott J. (1980). RI: an expert in the computer system domain. In *Proc. National Conference on Artificial Intelligence*, p. 269-271. McDermott J. (1981). RI's formative years. *AIMagazine*, 2(2).
202. McDermott J. (1982, a). RI: a rule-based configurer of computer systems. *Artificial Intelligence*, 19, p. 39-88. McDermott J. (1982, b). XSEL: a computer sales person's assistant. In *Machine Intelligence JO*

203. (Hayes J. E., Michie D. and Pao Y. H., eds.), p. 325-337. Chichester, UK: Ellis Horwood.
McDermott J. (1984). Building expert systems. In *Artificial Intelligence Applications for Business* (Reitman W., eds.). Norwood, NJ: Ablex.
204. McDermott J. (1988). Preliminary steps towards a taxonomy of problem solving methods. In *Automating Knowledge Acquisition for Expert Systems* (Marcus S., eds.), Chapter 8. Boston: Kluiver Academic. McDermott J. (1993). RI ("XCON") at age 12: lessons from an elementary school achiever.
205. *Artificial Intelligence*, 59, p. 241-247. McDermott J. and Bachant J. (1984). RI revisited: four years in the trenches. *AI Magazine*,
206. 5(3), Fall, p. 21-32.
207. McDermott J. and Forgy C. L. (1978). Production system conflict resolution strategies. In *Pattern Directed Inference Systems* (Waterman D. A. and Hayes-Roth F., eds.), p. 177-199. New York: Academic Press.
208. McNeill D. and Freiburger P. (1993). *Fuzzy Logic*. New York: Simon and Schuster. Meltzer B. and Michie D., eds. (1969). *Machine Intelligence 4*. Edinburgh: Edinburgh University Press.
209. Meyers S. (1995). *More Effective C++ : 35 New Ways to Improve Your Programs and Designs*. Reading, MA: Addison-Wesley.
210. Meyers S. (1997). *Effective C++ ; 50 Specific Ways to Improve Vom Programs and Designs*, 2nd edn. Reading, MA: Addison-Wesley.
211. Michalski R. S. (1983). A theory and methodology of inductive learning. In *Machine Learning* (Michalski R. S., Carbonell J. G. and Mitchell T. M., eds.), Chapter 4. Palo Alto, CA: Tioga.
212. Michalski R. S., Carbonell J. G. and Mitchell T. M., eds. (1983). *Machine Learning*. Palo Alto, CA: Tioga.
213. Michalski R. S., Carbonell J. G. and Mitchell T. M, eds. (1986). *Machine Learning Vol. II*. Palo Alto, CA: Tioga.
214. Michie D., eds. (1968). *Machine Intelligence 3*. Edinburgh: Edinburgh University Press.
215. Michie D., eds. (1979). *Expert Systems in the Micro-Electronic Age*. Edinburgh: Edinburgh University Press.
216. Mingers J. (1989, a). An empirical comparison of selection measures for decision tree induction. *Machine Learning*, 3, p. 319-342.
217. Mingers J. (1989, b). An empirical comparison of pruning methods for decision tree induction. *Machine Learning*, 4, p. 227-243.
218. Minsky M., eds. (1968). *Semantic Information Processing*. Cambridge, MA: MIT Press. Minsky M. (1972). *Computation: Finite and Infinite Machines*. London: Prentice-Hall.
219. Minsky M. (1975). A framework for representing knowledge. In *The Psychology of Computer Vision* (Winston P. H., eds.) p. 211-277. New York: McGraw-Hill. (Русский перевод: Минский М. Структура для представления знания. — В сб. Психология машинного зрения. Под. ред. П. Уинстона. М.: Мир, 1978. — с. 249-338.)
220. Mitchell T. M. (1978). Version Spaces: An Approach to Concept Learning. *Report No. STAN-CS-78-711*, Computer Science Department, Stanford University.
221. Mitchell T. M. (1982). Generalization as search. *Artificial Intelligence*, 18, 203-226.
222. Mitchell T. M. (1997). *Machine Learning*. New York, NY: McGraw-Hill.
223. Mitchell T. M., Keller R. M. and Kedar-Cabelli S. T. (1986). Explanation-based generalization:

224. A unifying view. *Machine Learning*, 1(1), p. 47-80. Moore J. D. (1995). *Participating in Explanatory Dialogues*. Cambridge MA: MIT Press.
225. Moser M. G. (1983). An Overview of NIKL, the New Implementation of KL-ONE. *Technical Report No. 5421*, Cambridge MA: Bolt, Beranek and Newman.
226. Moore J. D. and Paris C. L. (1993). Planning text for advisory dialogues: capturing intentional and rhetorical information. *Computational Linguistics*, 19(4), p. 651-695.
227. Moore J. D., Lemaire B. and Rosenblum J. A. (1996). Discourse generation for instructional applications: identifying and exploiting relevant prior explanations. *Journal of the Learning Sciences*, 5(1), p. 49-94.
228. Musen M. A. (1992). Overcoming the limitations of role-limiting methods. *Knowledge Acquisition*, 4(2), p. 165-170.
229. Musen M. A. (1989). Automated support for building and extending expert models. *Machine Learning*, 4(3-4), p. 347-376.
230. Musen M. A., Gennari J. H. and Wong W. W. (1995). A rational reconstruction of INTERNIST-I using PROTEGE-II. Knowledge Systems Laboratory, Medical Computer Science, *KSL-95-46*.
231. Neale I. M. (1988). First generation expert systems: a review of knowledge acquisition methodologies. *Knowledge Engineering Review*, 3(2), p. 105-145.
232. Neapolitan R. E. (1990). *Probabilistic Reasoning in Expert Systems: Theory and Algorithms*. New York: Wiley.
233. Neches R., Swartout W. R. and Moore J. (1985). Explainable (and maintainable) expert systems. In *Proc. 9th International Joint Conference on Artificial Intelligence*, p. 382-389.
234. Newell A. (1981). Physical symbol systems. In *Perspectives on Cognitive Science* (Norman D. A., eds.), Chapter 4. Norwood, NJ: Ablex.
235. Newell A. (1982). The knowledge level. *Artificial Intelligence*, 18, p. 87-127. Newell A. and Simon H. A. (1972). *Human Problem Solving*. Englewood-Cliffs, NJ: Prentice-Hall.
236. Newell A. and Simon H. A. (1976). Computer science as empirical enquiry. *Communications of the Association for Computing Machinery*, 19(3), p. 113-126.
237. Nguyen T., Czerwinski M. and Lee D. (1993). Compaq QuickSource: Providing the consumer with the power of artificial intelligence. In *Innovative Applications of Artificial Intelligence 5, Proceedings of AAAI-93*, p. 142-151. Menlo Park, CA: AAAI Press.
238. Nii H. P. (1986, a). Blackboard systems (Part 1). *AIMagazine*, 7(2), p. 38-53. Nii H. P. (1986, b). Blackboard systems (Part 2). *AI Magazine*, 7(3), p. 82-106.
239. Nii H. P. and Aiello N. (1979). AGE (Attempt to GEneralize): a knowledge-based program for building knowledge-based programs. In *Proc. 6th International Joint Conference on Artificial Intelligence*, p. 645-655.
240. Nii H. P., Aiello N. and Rice J. (1988). Frameworks for concurrent problem solving: a report on CAGE and POLIGON. In *Blackboard Systems* (Englemore R. and Morgan T, eds.), Chapter 25. Reading, MA: Addison-Wesley.
241. Nilsson N. J. (1971). *Problem Solving Methods in Artificial Intelligence*. New York: McGraw-Hill. (Русский перевод: НильсонН. Искусственный интеллект. Методы поиска решений. — М.: Мир, 1973. — 265 с.)
242. Nilsson N. J. (1980). *Principles of Artificial Intelligence*. Palo Alto, CA: Tioga. (Русский перевод: НильсонН. Принципы искусственного интеллекта.— М.: Радио и связь, 1985. —376с.)
243. Norman D. A., eds. (1981). *Perspectives on Cognitive Science*. Norwood NJ: Ablex.

244. Norvig P. (1992). *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Los Altos, CA: Morgan Kaufmann.
245. O'Rourke P. (1982). A Comparative Study of Two Inductive Learning Systems AQ11 and ID3 Using a Chess Endgame Test Problem. *Report No. 82-2*, Department of Computer Science, University of Illinois.
246. O'Shea T. and Eisenstadt M., eds. (1984). *Artificial Intelligence: Tools, Techniques, and Applications*. New York: Harper and Row.
247. Paterson A. and Niblett T. (1982). *ACLS Manual*, Version 1. Glasgow, UK: Intelligent Terminals.
248. Pearl J. (1982). Reverend Bayes on inference engines: a distributed hierarchical approach. In *Proc. National Conference on Artificial Intelligence*, p. 133-136.
249. Pearl J. (1984). *Heuristics. Intelligent Search Strategies for Computer Problem-Solving*. Reading, MA: Addison-Wesley.
250. Pearl J. (1986). On evidential reasoning in a hierarchy of hypotheses. *Artificial Intelligence*, 28, p. 9-15.
251. Pearl J. (1988). *Probabilistic Reasoning for Intelligent Systems*. Los Altos, CA: Morgan Kaufmann.
252. Pearl J. (1997). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Los Altos, CA: Morgan Kaufmann.
253. Pepper J. and Kahn G. (1987). Repair strategies in a diagnostic expert system. In *Proc. 10th International Joint Conference on Artificial Intelligence*, p. 531-534.
254. Peterson G. E., eds. (1987). *Object-oriented computing*, Vols 1 and 2. Washington DC: The Computer Society Press of the IEEE.
255. Poeson M. C. and Richardson J. (1988). *Foundations of Intelligent Tutoring Systems*. Hillsdale, NJ: Lawrence Erlbaum.
256. Pople H. E. Jr. (1977). The formation of composite hypotheses in diagnostic problem solving: an exercise in synthetic reasoning. In *Proc. 5th International Joint Conference on Artificial Intelligence*, p. 1030-1037.
257. Pople H. E. Jr. (1982). Heuristic methods for imposing structure on ill-structured problems: the structuring of medical diagnosis. In *Artificial Intelligence in Medicine* (Szolovits P., eds.), p. 119-190. Boulder, Co: Westview Press.
258. Post E. L. (1943). Formal reductions of the general combinatorial decision problem. *American Journal of Mathematics*, 65, p. 197-268.
259. Poundstone W. (1988). *Labyrinths of Reason*. New York: Doubleday.
260. Prerau D. S. (1990). *Developing and Managing Expert Systems*. Reading, MA: Addison-Wesley.
261. Provan G. M. (1990). The application of Dempster-Shafer theory to a logic-based visual recognition system. In *Uncertainty in Artificial Intelligence* (Henrion M., Shachter R. D., Kanal L. N. and Lemmer J. E., eds.), p. 389-405. Amsterdam: North Holland.
262. Puerta A. R., Neches R., Eriksson H., Szeleky P., Luo P. and Musen M. A. (1994). Toward ontology-based frameworks for knowledge acquisition tools. In *Proc. 8th Banff Knowledge-Based Systems Workshop*, 26, p. 1-14.
263. Quillian M. R. (1968). Semantic memory. In *Semantic Information Processing* (Minsky M., eds.), p. 227-270. Cambridge, MA: MIT Press.

264. Quine W. V. O. (1979). *Methods of Logic*. London: Routledge Kegan Paul.
265. Quinlan J. R. (1979). Discovering rules from large collections of examples: a case study. In *Expert Systems in the Micro-Electronic Age* (Michie D., eds.), p. 168-201. Edinburgh: Edinburgh University Press.
266. Quinlan J. R. (1983). Learning efficient classification procedures and their application to chess endgames. In *Machine Learning* (Michalski A. S., Carbonell J. G. and Mitchell T. M., eds.). Chapter 15. Palo Alto, CA: Tioga.
267. Quinlan J. R. (1986, a). Induction of decision trees. *Machine Learning*, 1, p. 81-106.
268. Quinlan J. R. (1986, b). The effect of noise on concept learning. In *Machine Learning Vol. II* (Michalski R. S., Carbonell J. G. and Mitchell T. M., eds.). Chapter 6. Palo Alto, CA: Tioga.
269. Quinlan J. R., eds. (1987). *Applications of Expert Systems*. Sydney: Addison-Wesley.
270. Quinlan J. R. (1993). *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann.
271. Raphael B. (1976). *The Thinking Computer: Mind inside Matter*. San Francisco: W. H. Freeman.
272. Rayward-Smith V. J. (1994). *Applications of Modern Heuristic Methods*. Alfred Waller.
273. Rayward-Smith V. J., Osman I. H. and Reeves C. R., eds. (1996). *Modern Heuristic Search Methods*. New York: Wiley.
274. Reichgelt H. and van Harmelen E. (1986). Criteria for choosing representation languages and control regimes for expert systems. *Knowledge Engineering Review*, 1(4), p. 2-17.
275. Reitman W., eds. (1984). *Artificial Intelligence Applications for Business*. Norwood, NJ: Ablex.
276. Reynolds D. (1988). MUSE: a toolkit for embedded, real-time, AI. In *Blackboard Systems* (Englemore R. and Morgan T., eds.), Chapter 27. Reading, MA: Addison-Wesley.
277. Rice J. (1986). POLIGON: A System for Parallel Problem Solving. *Technical Report No. KSL-86-19*, Knowledge Systems Laboratory, Stanford University.
278. Rice J. (1989). The advanced architectures project. *AI Magazine*, 10(4), p. 26-39. Rich E. and Knight K. (1991). *Artificial Intelligence*. New York: McGraw-Hill.
279. Richer M. H. and Clancey W. J. (1985). A graphic interface for viewing a knowledge based system. *IEEE Computer Graphics and Applications*, 5(11), p. 51-64.
280. Rine D. C., eds. (1975). *Computer Science and Multiple-Valued Logic Theory and Applications*. Amsterdam: North-Holland.
281. Rissland E. L., Daniels J. J., Rubinstein Z. B. and Skalak D. B. (1993). Case-based diagnostic analysis in a blackboard architecture. In *Proc. 11th National Conference on Artificial Intelligence*, p. 66-72.
282. Robinson J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12, p. 23-41. (Русский перевод: Робинсон Дж. Машинно-ориентированная логика, основанная на принципе резолюции. — В кн. Кибернетический сборник, вып. 7, 1970, — с. 194-218.)
283. Robinson J. A. (1979). *Logic: Form and Function*. Edinburgh: Edinburgh University Press.
284. Robinson V., Hardy N. W., Barnes D. P., Pace C. J. and Lee M. H. (1987). Experiences with a knowledge engineering toolkit: an assessment in industrial robotics. *Knowledge Engineering Review*, 2(1), p. 43-54.

285. Rose D. E. (1994). *A Symbolic and Connectionist Approach to Legal Information Retrieval*. Hillsdale, NJ: Lawrence Erlbaum.
286. Rosenman M. A., Coyne R. D. and Gero I. S. (1987). Expert systems for design applications. In *Applications of Expert Systems* (Quinlan J. R., eds.). Chapter 4. Sydney: Addison-Wesley.
287. Rothenfluh T. E., Gennari J. H., Eriksson H., Puerto A. R., Tu S. W. and Musen M. A. (1994). Reusable ontologies, knowledge acquisition tools, and performance systems: PROTOGE-II solutions to Sisyphus-2. In *Proc. 8th Banff Knowledge-Based Systems Workshop*, 43, p. 1-30.
288. Rumelhart D. E. (1988). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Cambridge, MA: MIT Press.
289. Rumelhart D. E. and McClelland J. L. (1986) *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*. Cambridge, MA: MIT Press.
290. Russell S. J. and Norvig P. (1995). *Artificial Intelligence: A Modern Approach*. Englewood Cliffs, NJ: Prentice-Hall.
291. Sacerdoti E. D. (1974). *A Structure for Plans and Behavior*. Amsterdam: Elsevier North-Holland.
292. Sandewall E. (1986). Nonmonotonic inference rules for multiple inheritance with exceptions. In
293. *Proc. IEEE*, 74, p. 81-132. Sanford A. J. (1987). *The Mind of Man: Models of Human Understanding*. Brighton: Harvester
294. Press.
295. Schach S. R. (1993). *Software Engineering*. Richard D. Irwin.
296. Schank R. C. (1975). *Conceptual Information Processing*. Amsterdam: North-Holland.
297. Schank R. C. and Abelson R. (1977). *Scripts, Plans, Goals and Understanding*. Hillsdale, NJ: Lawrence Erlbaum.
298. Schank R. C. and Colby K., eds. (1973). *Computer Models of Thought and Language*. New York: W. H. Freeman.
299. Schmolze J. G. (1991). Guaranteeing serializable results in synchronous parallel production systems. *Journal of Parallel and Distributed Computing*, 13(4), p. 348-365.
300. Schreiber A. Th., Wielinga B. J. and Breuker J. A., eds. (1993). *KADS: A Principled Approach to Knowledge-Based System Development*. London: Academic Press.
301. Schubert L. K. (1976). Extending the expressive power of semantic networks. *Artificial Intelligence*, 1, p. 163-198.
302. Selman B. and Levesque H. J. (1989). The tractability of path-based inheritance. In *Proc. 11th International Joint Conference on Artificial Intelligence*, p. 1140—1145.
303. Shafer G. (1976). *A Mathematical Theory of Evidence*. Princeton NJ: Princeton University Press.
304. Shafer G. and Pearl J. (1990). *Readings in Uncertain Reasoning*. Los Altos, CA: Morgan Kauffman.
305. Shafer G. and Tversky A. (1985). Languages and designs for probability judgment. *Cognitive Science*, 9, p. 309-339.
306. Shannon C. E. (1950). Automatic chess player. *Scientific American*, 182(48). (Русский перевод: Шеннон К. Работы по теории информации и кибернетике. — М.: ИЛ., 1963.)
307. Shepherdson J. C. (1984). Negation as Failure. *Journal of Logic Programming*, 1, p. 51-81.

308. Shepherdson J. C. (1985). Negation as Failure II. *Journal of Logic Programming*, 3, p. 185-202.
309. Shortliffe E. H. (1976). *Computer-Based Medical Consultations: MYCIN*. New York: Elsevier.
310. Shortliffe E. H., Scott A. C., Bischoff M. B., van Melle W. and Jacobs C. D. (1981). ONCOCIN: an expert system for oncology protocol management In *Proc. 7th International Joint Conference on Artificial Intelligence*, p. 876-881.
311. Sime M. E. and Coombs M. J., eds. (1983). *Designing for Human Computer Communication*. London: Academic Press.
312. Simon H. A. (1983). Why should machines learn? In *Machine Learning* (Michalski R. S., Carbonell J. G. and Mitchell T. M., eds.), Chapter 2. Palo Alto, CA: Tioga.
313. Slade S. (1991). Case-based reasoning: A research paradigm. *AI Magazine*, 12(1), Spring, p. 42-55.
314. Sleeman D. and Brown J. S., eds. (1982). *Intelligent Tutoring Systems*. London: Academic Press.
315. Smith B. C. (1982). Reflection and Semantics in a Procedural Language. *MIT-TR-272*, Massachusetts Institute of Technology. Also in *Readings in Knowledge Representation* (Brachman R. J. and Levesque H. J., eds.), 1985. Los Altos, CA: Morgan Kaufmann.
316. Smith R. G., Winston H. A., Mitchell T. and Buchanan B. G. (1985). Representation and use of explicit justifications for knowledge base refinement. In *Proc. 9th International Joint Conference on Artificial Intelligence*, p. 673-680.
317. Smullyan R. (1978). *What Is the Name of this Book?* New York: Simon and Schuster.
318. Stefik M. (1979). An examination of a frame-structured representation system. In *Proc. 6th International Joint Conference on Artificial Intelligence*, p. 845-852.
319. Stefik M. (1981, a). Planning with constraints. *Artificial Intelligence*, 16, p. 111-140.
Stefik M. (1981, b). Planning and meta-planning. *Artificial Intelligence*, 16, p. 141-169.
Stefik M. (1995). *Introduction to Knowledge Systems*. San Francisco, CA: Morgan Kaufmann.
320. Stefik M. and Bobrow D. G. (1986). Object-oriented programming: themes and variations. *AI Magazine*, 6(4), p. 40-62.
321. Stefik M., Aikins J., Balzer R., Benoit J., Bimbaum L., Hayes-Roth F. and Sacerdoti E. (1983). The architecture of expert Systems. In *Building Expert Systems* (Hayes-Roth F., Waterman D. A. and Lenat D., eds.). Chapter 4. Reading, MA: Addison-Wesley.
322. Sterling L. and Shapiro E. (1994). *The Art of Prolog*, 2nd edn. Cambridge, MA: MIT Press. (Русский перевод первого издания: Стерлинг Л., Шапиро Э. Искусство программирования на языке ПРОЛОГ. — М.: Мир, 1990.)
323. Stroustrup B. (1997). *The C++ Programming Language*, 3rd ed. Reading, MA: Addison-Wesley.
324. Studer R., Benjamins V. R. and Fensel D. (1998). Knowledge Engineering: Principles and Methods. *Data and Knowledge Engineering*, 25(1, 2), p. 161-198.
325. Sussman G. J. (1975). *A Computer Model of Skill Acquisition*. New York: American Elsevier.
326. Suwa W., Scott A. C. and Shortliffe E. H. (1982). An approach to verifying completeness and consistency in a rule-based expert system. *AI Magazine*, 3(4), p. 16-21.
327. Swartout W. R. (1983). XPLAIN: a system for creating and explaining expert consulting programs. *Artificial Intelligence*, 21, p. 285-325.

328. Szolovits P., eds. (1982). *Artificial Intelligence in Medicine*. Boulder, CO: Westview Press.
329. Tate A. (1977). Generating project networks. In *Proc. 5th International Joint Conference on Artificial Intelligence*, p. 888-893.
330. Teknowledge (1985). *S.I. Reference Manual*. Palo Alto, CA: Teknowledge.
331. Thomason R. (1992). NETL and subsequent path-based inheritance theories. *Computers Math. Applic.*, 23 (2-5), p. 179-204.
332. Tommelein I. D., Johnson M. V., Hayes-Roth B. and Levitt R. E. (1987). SIGHTPLAN—a blackboard expert system for the layout of temporary facilities on a construction site. In *Proc. IFIP WG5.2 Working Conference on Expert Systems in Computer-Aided Design*, Sydney, Australia.
333. Touretzky D. S. (1986). *The Mathematics of Inheritance Systems*. London: Pitman.
334. Touretzky D. S., Horty J. F. and Thomason R. H. (1987). A clash of intuitions: the current state of nonmonotonic multiple inheritance systems. In *Proc. 10th International Joint Conference on Artificial Intelligence*, p. 476-482.
335. Tu S. W., Eriksson H., German J., Shahar Y. and Musen M. A. (1995). Ontology-based configuration of problem-solving methods and generation of knowledge-acquisition tools: application of PROTEGE-II to protocol-based decision support. *Artificial Intelligence in Medicine*, 1, p. 257-289.
336. Tversky A. and Kahneman D. (1974). Judgment under uncertainty: Heuristics and biases. *Science*, September. Also in *Readings in Uncertain Reasoning* (Shafer G. and Pearl J., eds.), 1990. Los Altos, CA: Morgan Kaufmann.
337. van Lamsweerde A. and Dufour P., eds. (1987). *Current Issues in Expert Systems*. London: Academic Press.
338. van Melle W. J. (1981). *System Aids in Constructing Consultation Programs*. Ann Arbor MI: UMI Research Press.
339. von Luck K. and Marburger H., eds. (1994). *Management and Processing of Complex Data Structures, Third Workshop on Information Systems and Artificial Intelligence, Proceedings*. Lecture Notes in Computer Science Vol. 777. Berlin: Springer Verlag.
340. Walker E. A. and Nguyen H. T. (1996). *A First Course in Fuzzy Logic*. Boca Raton, FL: CRC Press.
341. Ward R. D. and Sleeman D. (1987). Learning to use the S.I knowledge engineering tool. *Knowledge Engineering Review*, 2(4), p. 265-276.
342. Waterman D. A. (1986). *A Guide to Expert Systems*. Reading, MA: Addison-Wesley. (Русский перевод: Уотермен Д. Руководство по экспертным системам. —М.: Мир, 1989.)
343. Waterman D. A. and Hayes-Roth F. (1978). *Pattern Directed Inference Systems*. New York: Academic Press.
344. Watson I. (1995). *Progress in Case-Based Reasoning*. Lecture Notes in Artificial Intelligence, 1020. Berlin: Springer-Verlag.
345. Watson I. and Marir F. (1994). Case-Based Reasoning: A Review. *The Knowledge Engineering Review*, 9 (4), p. 355-381.
346. Weiss S. M. and Kulikowski C. A. (1983). *A Practical Guide to Designing Expert Systems*. London: Chapman and Hall.
347. Wielinga B. J. and Breuker J. A. (1986). Models of expertise. In *Proc. 7th European Conference on Artificial Intelligence*, p. 306-318.

348. Wielinga B. J. and Schreiber A. Th. (1994). Conceptual modelling of large reusable knowledge bases. In *Management and Processing of Complex Data Structures, Third Workshop on Information Systems and Artificial Intelligence, Proceedings* (von Luck K. and Marburger H., eds.), p. 181-200. Lecture Notes in Computer Science, Vol. 777. Berlin: Springer Verlag.
349. Wielinga B. J., Schreiber A. Th. and Breuker J. A. (1992). KADS: A modelling approach to knowledge engineering. *Knowledge Acquisition*, 4(1), p. 5-53. Special issue: The KADS approach to knowledge engineering. Reprinted in *Readings in Knowledge Acquisition and Learning* (Buchanan B. G. and Wilkins D. G., eds.), 1992. Los Altos, CA: Morgan Kaufmann.
350. Wilkins D. C. (1988). Knowledge base refinement using apprenticeship learning techniques. In *Proc. National Conference on Artificial Intelligence*, p. 646-651.
351. Wilkins D. C. (1990). Knowledge base refinement as improving an incomplete and incorrect domain theory. In *Machine Learning: An Artificial Intelligence Approach*, Vol. III (Kodratoff Y. and Michalski R. S., eds.), p. 493-514. Los Altos, CA: Morgan Kaufmann.
352. Wilkins D. C. and Buchanan B. G. (1986). On debugging rule sets when reasoning under certainty. In *Proc. National Conference on Artificial Intelligence*, p. 448—454.
353. Winograd T. (1972). Understanding natural language. *Cognitive Psychology*, 1, p. 1-191. (Русский перевод первого издания: Виноград Т. Программа, понимающая естественный язык. — М.: Мир, 1976.)
354. Winograd T. (1975). Frame representations and the declarative/procedural controversy. In *Representation and Understanding* (Bobrow D. G. and Collins A., eds.). New York: Academic Press.
355. Winston P. H. (1975, a). Learning structural descriptions from examples. In *The Psychology of Computer Vision* (Winston P. H., eds.), Chapter 5. New York: McGraw-Hill.
356. Winston P. H., eds. (1975, b). *The Psychology of Computer Vision*. New York: McGraw-Hill. Winston P. H. (1982). Learning new principles from precedents and examples. *Artificial Intelligence*, 19, p. 321-350.
357. Winston P. H. (1984). *Artificial Intelligence*. Reading, MA: Addison-Wesley Winston P. H. (1992). *Artificial Intelligence*, 3rd edn. Reading, MA: Addison-Wesley. (Русский перевод первого издания: Уинстон П. Искусственный интеллект. — М.: Мир, 1980. — 519с.)
358. Winston P. H. and Brown R. H. (1979). *Artificial Intelligence: An MIT Perspective*, Vols 1 and 2. Cambridge, MA: MIT Press.
359. Winston P. H. and Horn K. P. (1981). *LISP*. Reading, MA: Addison-Wesley. Winston P. H. and Horn K. P. (1984). *LISP*, 2nd ed. Reading, MA: Addison-Wesley. Winston P. H. and Horn K. P. (1988). *LISP*, 3rd ed. Reading, MA: Addison-Wesley.
360. Winston P. H. and Shellard S. A. (1990). *Artificial Intelligence at MIT: Expanding Frontiers*, Vols 1 and 2. Cambridge, MA: MIT Press.
361. Wise B. P. and Henrion M. (1986). A framework for comparing uncertain inference systems to probability. In *Uncertainty in Artificial Intelligence* (Kanal L. N. and Lemmer J. F., eds.) p. 69-83. Amsterdam: North-Holland.
362. Woods W. (1975). What's in a link: Foundations for semantic networks. In *Representation and Understanding* (Bobrow D. G. and Collins A., eds.). New York: Academic Press.
363. Yager R. R. and Filev D. P. (1994). *Essentials of Fuzzy Modeling and Control*. New York: Wiley.

364. Yen J. (1986). A reasoning model based on an extended Dempster-Shafer theory. In *Proc National Conference on Artificial Intelligence*, p. 125-131. Yen J., Juang H. and MacGregor R. (1991, a). Using polymorphism to improve expert system maintainability. *IEEE Expert*, April. Yen J., Neches R. and MacGregor R. (1991, b). CLASP: Integrating term subsumption systems and production systems. In *IEEE Transactions on Knowledge and Data Engineering*, 3, 1, March.
365. Zadeh L. A. (1965). Fuzzy sets. *Information and Control*, 8, p. 338-353. Zadeh L. A. (1975). Fuzzy logic and approximate reasoning. *Synthese*, 30, p. 407-428. Zadeh L. A. (1978). Fuzzy sets as a basis for a theory of possibility. *Fuzzy Sets and Systems* 1, p. 3-28. Zadeh L. A. (1981). PUF — a meaning representation language for natural languages. In *Fuzzy Reasoning and its Applications* (Mamdani E. H. and Gaines B. D., eds.), p. 1-66. London: Academic Press.

НАВЧАЛЬНИЙ ПОСІБНИК

БАКЛАН І.В.
ЕКСПЕРТНІ СИСТЕМИ

Здано до набору 01.03.2011. Підписано до друку 03.03.2011.
Формат видання 60X90 1/16. Папір офсетний. Гарнітура Times New Roman.
Наклад 300 прим. Замовлення N__.

Національна академія управління

03151, Київ, вул. Вінницька, 10