МУЛЬТИПАРАДИГМЕННЕ ПРОГРАМУВАННЯ

Лекція 7

ФАКТИ І ПРАВИЛА CLIPS

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 - робочої 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-покажчик, 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 порожній, послідовності полів, розділених пробілами. Обмеженням факту служать круглі дужки.

Визначення 7.1. Упорядкований факт

(дане_типу_symbol [поле]*)

Перше поле факту визначає так називане відношення, або зв'язок факту (relation). Термін "зв'язок" означає, що даний факт належить деякому певному конструктором або неявно оголошеному шаблону. Докладніше мова про це піде нижче.

Приведемо кілька прикладів фактів:

Приклад 7.1 Упорядковані факти

```
(duck is psisa)
(schoolboys is Koti Goroshko)
(Nuke did report)
(altitude is 1000 hvutiv)
```

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 використає 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, використається конструктор **`deftemplate`**. Конструктор **`deftemplate`** аналогічний визначенням записів або структур у таких мовах програмування, як Pascal або C.

Конструктор **`deftemplate`** задає ім'я шаблона й визначає послідовність із нуля або більше полів неупорядкованого факту, називаних 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 посилатися 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 шаблона, 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  "Ivasyk Telesyk")   (id X9345A))
(point-mass   (x-velocity 100)    (y-velocity -200))
(class (teacher "Igor Baklan") (#-students  30) (Room "417-18")
(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 "Igor Baklan")(#-students 30)(Room "417-18"))
(class (#-students 30)(teacher "Igor Baklan")(Room "417-18"))
(class (Room "37A")(#-students 30)(teacher "417-18"))
```

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

```
(class "Igor Baklan" 30 "417-18")
(class 30 "Igor Baklan" "417-18")
(class "417-18" 30 "Igor Baklan")
```

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 уведенні даної команди в CLIPS ви повинні побачити результат, наведений на рис. 7.1.

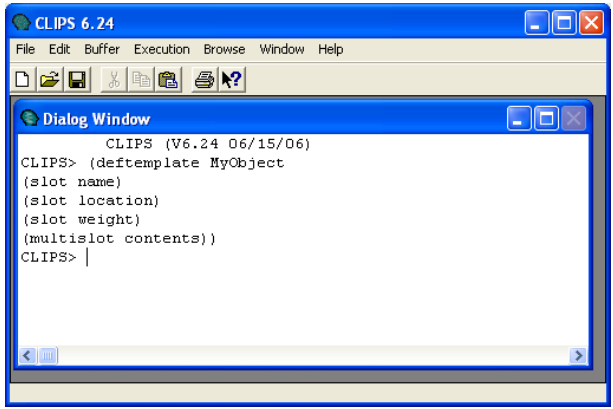

Рис. 7.1. *Використання конструктора* **`deftemplate`**

Подібна реакція середовища говорить про вдале додавання визначення шаблона 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 скористайтеся 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, видаляти обраний шаблон і відображати 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 шаблона й видаленні його з поточної бази знань.

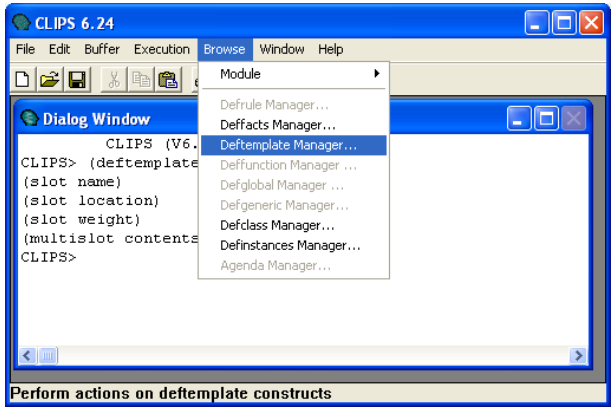

Рис. 7.2. Запуск менеджера шаблонів

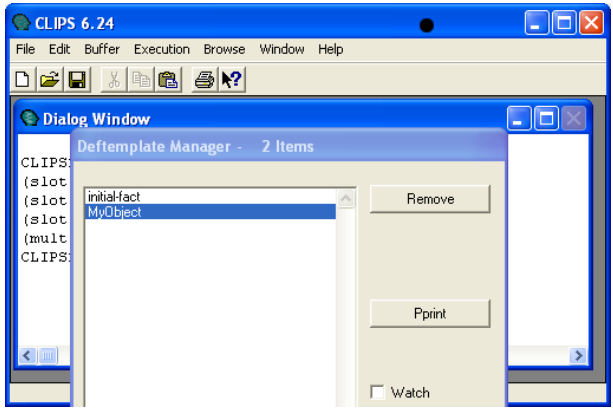

Рис. 7. 3. Вікно менеджера шаблонів

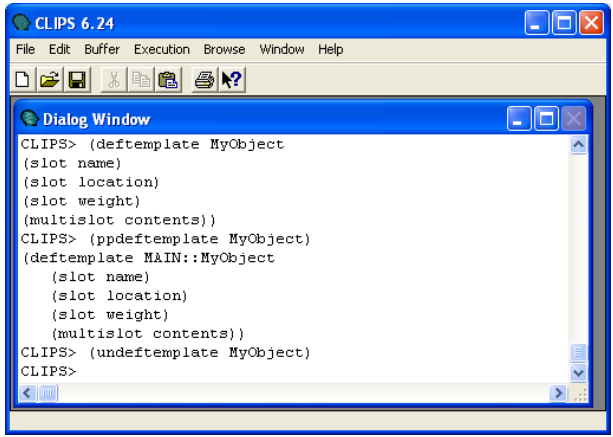

Рис. 7.4. Одержання інформації й видалення шаблона

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

У випадку, якщо при додаванні нового шаблона за допомогою конструктора **`deftemplate`** відбулася помилка, користувач одержить відповідне попередження. .

Перевизначення вже існуючого шаблона приводить до виключення попереднього визначення. Шаблон не може бути перевизначений доти, поки він використається (наприклад, фактом або правилом). Шаблон може мати будь-яка 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-шаблона>[<необов'язков-коментарі>]
[<определение-слота>*])

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

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

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

<атрибута-шаблона> ::= <атрибуту-значенню-по-
умовчанню>|<атрибута-обмеження>

<атрибуту-значенню-по-умовчанню> ::= (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 шаблона. Коментарі необхідно містити в лапки. Крім даного типу коментарів у конструкторі deftempiate 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 випливають після імені шаблона й ув'язнені в лапки, зберігаються в базі знань системи. Ці коментарі доступні при перегладі визначення шаблона.

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 шаблона в систему, за допомогою менеджера шаблонів, виведіть у головне вікно CLIPS інформацію про визначення шаблона 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 шаблона. На жаль, що ця версія CLIPS не сприймає символи кирилиці навіть як коментарі, тому всі коментарі прийде давати англійською мовою.

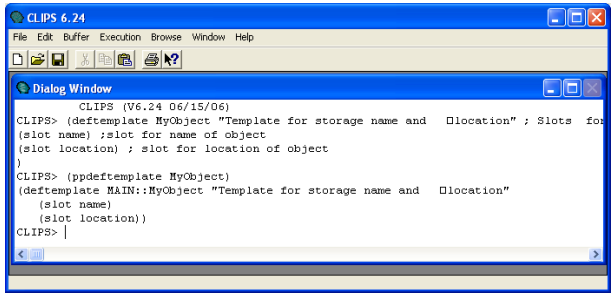

Рис. 8.5. Використання коментарів у конструкторі **`deftemplate`**

Крім ключового слова **slot**, що визначає простий слот, припустиме також застосування ключового слова **multisiot**, для визначення складеного слоту. Простий слот, або слот, призначений для зберігання одиниці інформації одного із примітивних типів даних CLIPS. Складений слот здатний зберігати список подібних одиниць інформації необмеженого об'єм. Для доступу до конкретних даних, що зберігається в складеному слоті, використаються спеціальні групові символи й функції, приклади й правила використання яких будуть наведені нижче.

При створенні шаблона за допомогою конструктора **deftemplate** кожному полю можна призначати 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 шаблона. 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 витягатися 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 приведений 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 шаблона деякою командою або конструктором. Наприклад, для запису значень у слоти шаблона. 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 неоднозначної. Тому в командах, що використають 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 шаблона 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 настроювання 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.

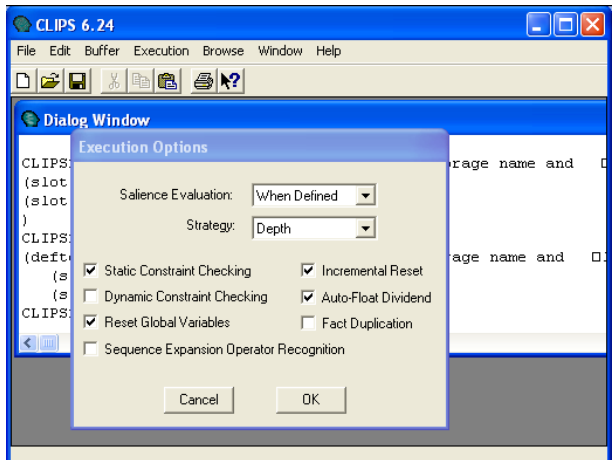

Рис. 7.6. Діалогове вікно Execution Options

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 конструкцій **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 вираз, 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 вираз, 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.

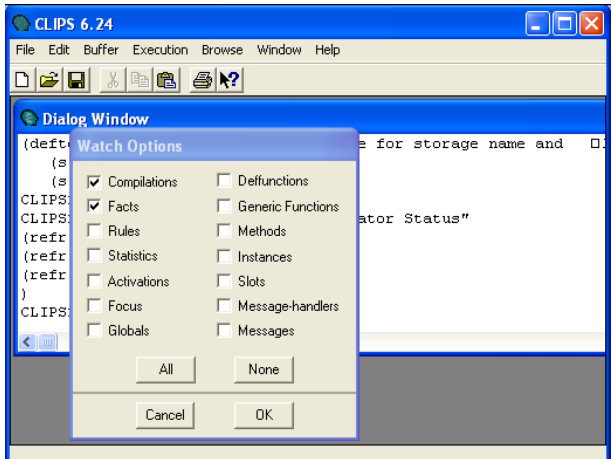

Рис. 7.7. Діалогове вікно Watch Options

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 наведеним на мал. 7.8.

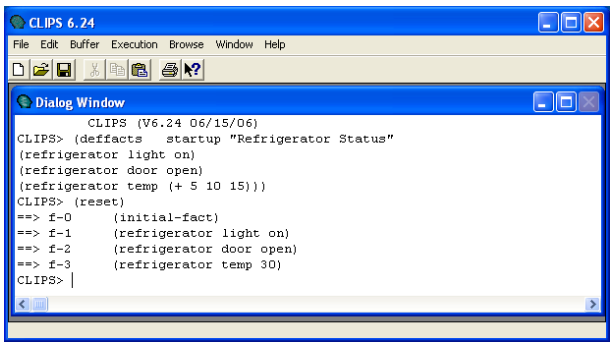

Рис. 7.8. Перегляд процесу додавання файлів

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.

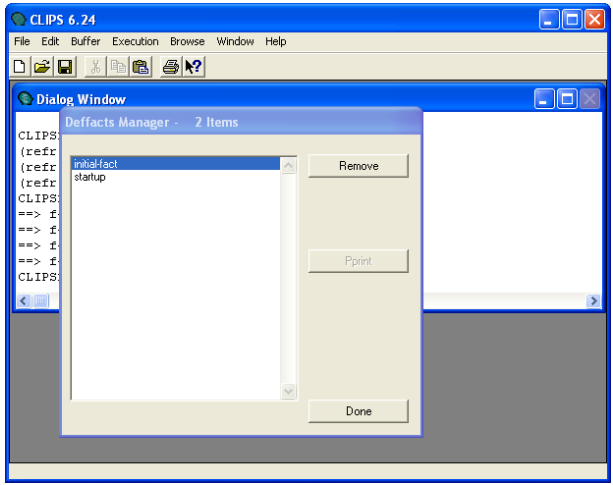

Рис. 7.9. Вікно менеджера визначених фактів

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, уведені після імені конструктора, зберігаються й виводяться на екран так само, як у конструкторі **deftemplate**.

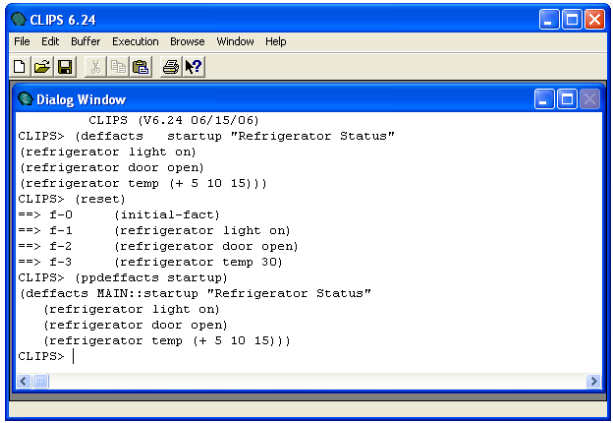

Рис. 7.10. Одержання інформації про певного конструктора

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** шаблона **initial-fact** надає зручний спосіб для запуску програм мовою CLIPS — правила, що не мають умовних елементів, автоматично перетворяться 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 далі, відразу після першого знайомства із правилами CLIPS.

7.2.3. Функція *assert*

Функція **assert** - одна з найбільше часто застосовних 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 робочі 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, те 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** очищує 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.

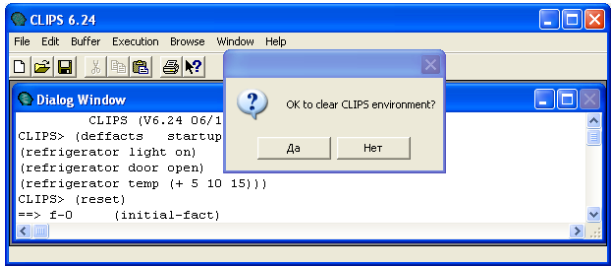

Рис. 7.11. Підтвердження очищення середовища CLIPS

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.

Включіть 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.

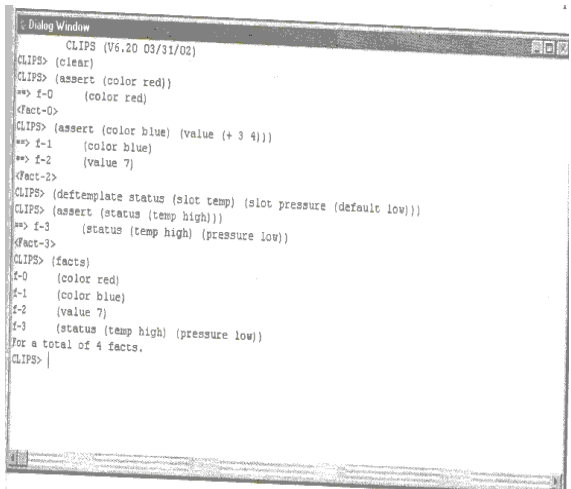

Рис. 7.12. Додавання фактів

Зверніть увагу, що при ініціалізації факту **value** використовувався 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 настроювання. 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 виразом, 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 повертається.

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.

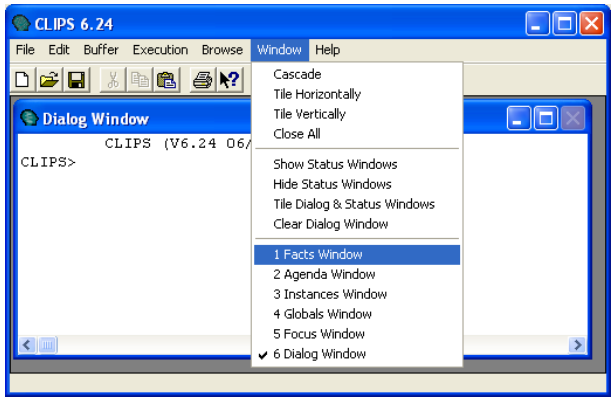

Рис. 7.13. Список фактів

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.

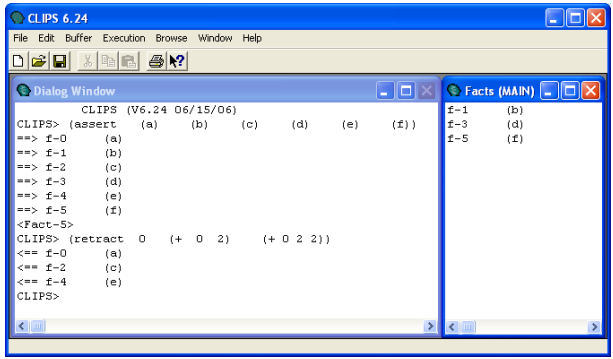

Рис. 7.14. *Результат додавання й видалення фактів*

У випадку, якщо факт із зазначеним індексом не буде знайдений, CLIPS видасть відповідне повідомлення про помилку.

Виконаємо команду:

Приклад 7.11. Видалення всіх фактів

```
(retract *)
```

Після виконання даної команди список фактів буде очищене 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-6**, 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 7.12. 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>
<новое-значение-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.

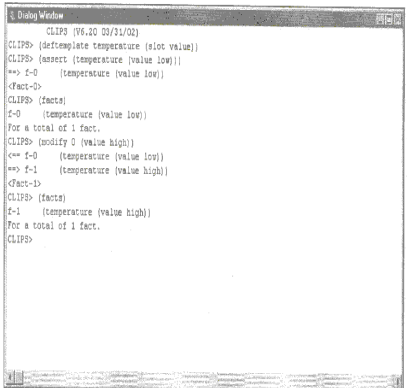

Рис. 7.15. *Результат зміни існуючого неупорядкованого факту*

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>
            <новое-значение-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 уведення 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 поводження 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 символьний 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 використається *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 використайте 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 далі.

Приведемо простий приклад використання даної функції:

Приклад 7.17. 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 шаблона, 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, так й в описаних 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 розглянутою 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>    <имя-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**.

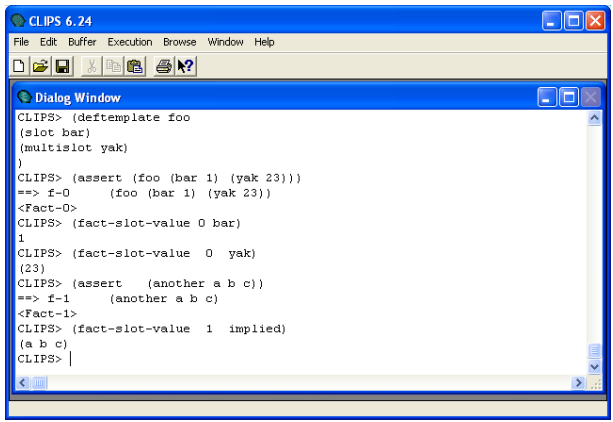

Рис. 7.16. Результат використання функції fact-slot-value

Виконаємо 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 кропітке 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 випливати 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-factl)    (simple-fact2))
(save-facts  fl  local  template simple-factl)
```

Послідовність дій, наведена в даному прикладі, зберігає у файл **f1**, що перебуває в поточному каталозі, всі факти, видимі в поточному модулі й пов'язані із шаблонами **template** й **simple-factl** (як ви пам'ятаєте, після додавання факту **simple-factl** CLIPS визначає неявно створений шаблон **simple-factl**). У результаті буде отриманий текстовий файл із наступним змістом:

Приклад 7.22. Зміст файлу fl

```
(template   (a  1)    (b  2) } (simple-factl)
```

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 виконаємо:

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 надалі, 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 ***Т-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 використаються 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 застосовано більше одного правила, механізм логічного висновку використає так називану ***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**, використовуваного 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, C. 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)
```

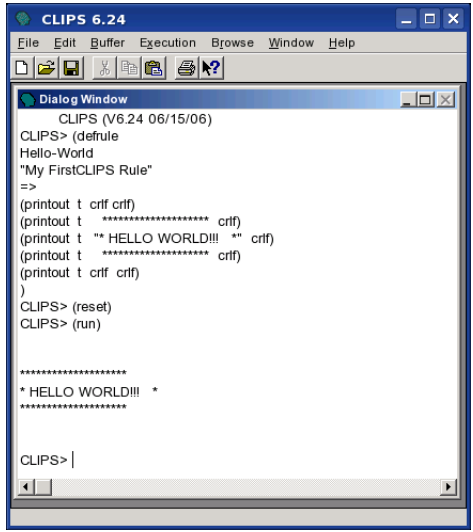

Рис. 8.1. *Результат роботи програми "Hello-World!"*

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++. Вираження **`clrf`** 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 цій 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.

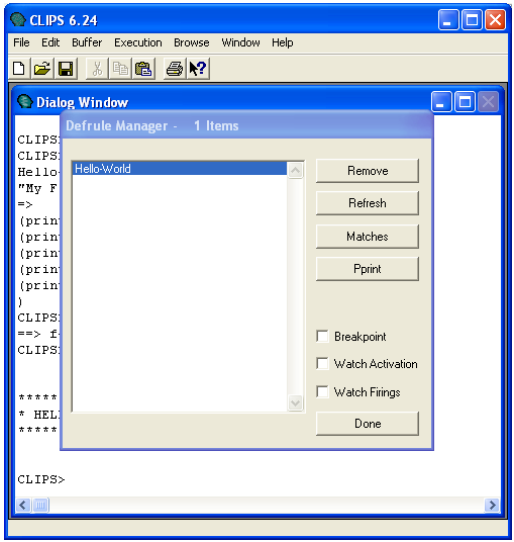

Рис. 8.2. *Вікно менеджера правил*

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 правила 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 раз рішення 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 прапорець **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 <цілочисельний вираз>):
(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 настроювання 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.

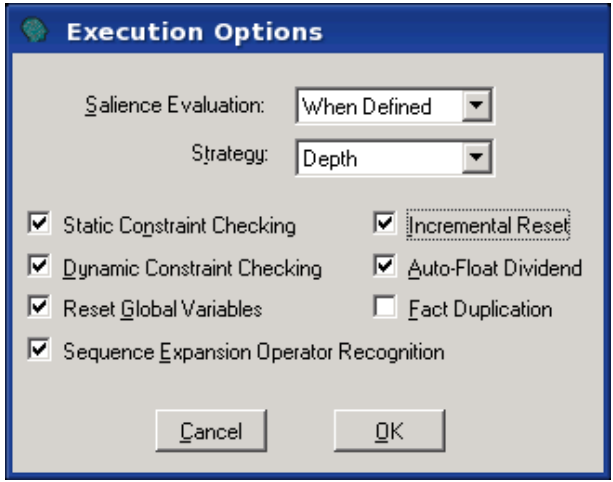

Рис. 8. 3. *Установка способу обчислення пріоритетів правил*

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** установлена 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 установлена в **FALSE**.

8.4. Стратегія вирішення конфліктів

План вирішення 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 вирішення задачі. Виконання плану подібно стеку (верхнє правило плану завжди буде виконано першим).

Коли активується нове правило, воно розміщається в плані вирішення 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 установлена 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'явилося, 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 розміщаються 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'язаної 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, випливаюче 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'язаної змінної **?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 розміщаються вище всіх активованих правил з рівною або меншою визначеністю.

8.4.5. Стратегія LEX

Між правилами з однаковим пріоритетом тільки що активовані правила розміщаються 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 убуванню тегів для визначення місця розташування правила. Активація правила, виконана більше новими зразками, розташовується перед активацією, здійсненої більше пізніми зразками. Для визначення порядку розміщення двох активацій правил, поодинці 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 розміщається 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 меншої. 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, те більше і його часовий тег. Для даного прикладу приймемо, що тимчасові теги дорівнюють індексам.

Приклад 8.3. 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 8.4. 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 розміщаються з використанням однойменної стратегії, уперше використаної в системі OPS5.

Основна відмінність стратегії MEA від LEX полчгає 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 Меа-порядке (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

Кожної 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 наступної установки 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 часта 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 РЄ)

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 символьного обмеження, однак, кілька обмежень можна з'єднувати разом. На додаток до символьних 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 шаблона, створеного за допомогою конструктора deftemplate або неявно створеного шаблона. 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

```
(deffacts   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))
(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))
```

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 символьні обмеження при зіставленні зразків повинні точно збігатися по всіх зазначених полях, інакше факт не буде вважатися придатному даному зразку.

Умовний елемент, що представляє собою зразок для неупорядкованого факту, у якому присутні тільки символьні обмеження, має наступний синтаксис:

Визначення 8.4. Синтаксис символьних обмежень для неупорядкованого факту

```
(<обмеження-1> ... <обмеження-n>)
```

Умовний елемент, що представляє собою зразок для шаблона, у якому присутні тільки символьні обмеження, виглядає так:

Визначення 8.5. Синтаксис символьних обмежень для шаблона

```
(<ім'я-шаблону >
   (<ім'я-слота-1> <обмеження-1>)
   …
   (<ім'я-слота-n>  <обмеження-n>))
```

Розглянемо приклад правил, що використають як зразок — зразок фактів (як упорядкованих, так і шаблонів) із символьними 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.

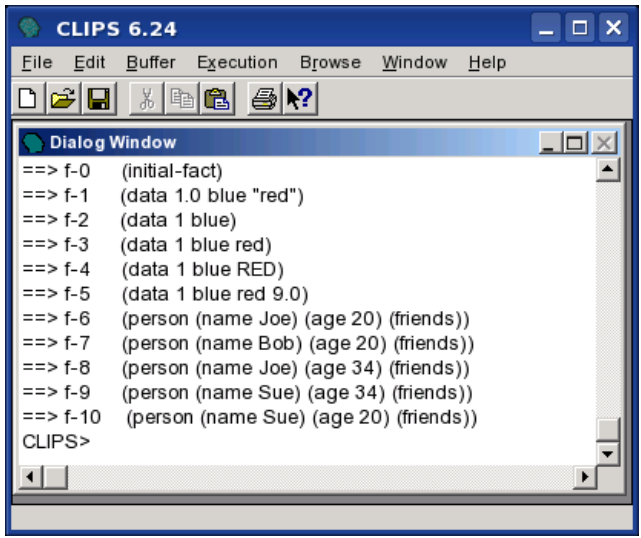

Рис. 8.4. *Список необхідних фактів*

Для нормальної роботи прикладів не забувайте виконувати команду reset перед кожним запуском правил. Уведемо в CLIPS визначення наступних правил:

Приклад 8.7. Правила із символьними 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))
```

Виконаємо команди **reset** й **run**. Ви повинні одержати результат, наведений на мал. 8.5.

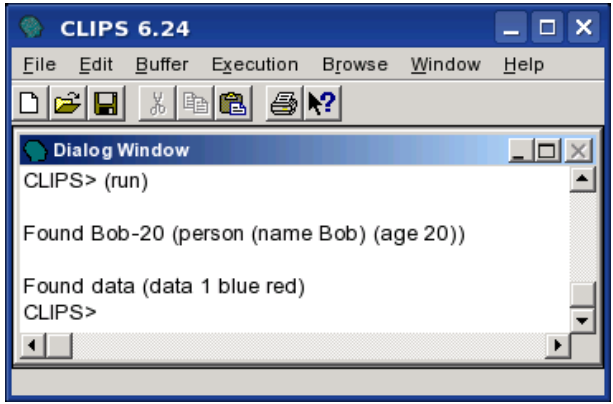

Рис. 8.5. Виконання правил із символьними обмеженнями

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 символьним 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 простій слот шаблона 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 символьні 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 шаблона зразок прийме вид:

Визначення 8.7. Синтаксис обмежень для шаблона

```
(<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 шаблона, 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 8.11. Перетворене правило match-all-persons

```
(defrule      match-all-persons
                  (person
                  (name  ?)
                  (age   ?)
                  (friends  $?))
                 =>)
```

Це правило буде активувати всі факти шаблона person.

Групові символи для складеного поля можна комбінувати із символьними 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 8.12. 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 сформувати за допомогою групових символів для простого поля. Однак подібна спільність приводить до того, що процес зіставлення зразків, що використають групові символи, іноді займає набагато більше часу, чим аналогічний процес зі зразками, що використають тільки групові символи для простих полів.

8.5.1.3. Змінні, пов'язані із простими й складеними полями

Групові символи заміняють 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 починатися 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 поводиться так само, як і відповідний груповий символ. У цей момент CLIPS зв'язує значення поля із заданої змінної. 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 лівої, так і правої частини 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 шаблона або об'єкта.

Як приклад уведемо 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))
```

Виконаємо 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          в=()                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 8.18. 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-Coevai-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 випливають & 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, наведених вище, можна розширити так:

Визначення 8.10. Синтаксис обмежень

```
<обмеження> ::=  ?   :
                  $? :
                  <зв'язане-обмеження>
<зв'язане-обмеження>          ::=  <просте-обмеження>   |
      <просте-обмеження>&<зв'язане-обмеження>|
      <просте-обмеження>|<зв'язане-обмеження>|
<просте-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-coevai-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 8.20. 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 використає 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 кожнім зіставленні зразків, а не один раз при визначенні правила.

Обмеження, що повертають значення, мають наступний синтаксис:

Визначення 8.13. Синтаксис обмеження, що повертає значення

```
=<виклик-функції>
```

Визначення поняття "елемент", наведені в попередньому розділі, приймуть такий вид:

Визначення 8.14. Синтаксис поняття "елемент"

```
<елемент>       ::= <константа>|
                    <проста-змінна>|
                    <складова-змінна>|
                    :<виклик-функції>
                    =<виклик-функції>
```

Правило із приклада 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 8.21. 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 8.15. Синтаксис зразків об'єктів

```
<зразок об'єкта>       ::=   (object  <атрибута-обмеження>)
<атрибута-обмеження>   ::=   (is-a  <обмеження>)|
                                    (name  <обмеження>)|
                                    (slot  <обмеження>)
```

Обмеження is-a (є) 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 8.24. 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 випливає із приклада 9.46, буде перетворено так, як показано в прикладі 9.47:

Приклад 8.46. Правило без умов

```
(defrule  example
                          => )
```

Приклад 8.47. 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 розміщують 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 випливають із приклада 9.48, будуть змінені так, як у прикладі 8.49.

Приклад 8.48. 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 8.49. 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 приклада 8.50 зміниться на еквівалентне (приклад 8.51):

Приклад 8.50. Правило з умовами not перед елементом test

```
(defrule        example
                   (a   ?x)
                   (not   (b   ?x))
                   (test   (>  ?x  5)}
                   =>)
```

Приклад 8.51. 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, правило (приклад 8.52) буде змінено так, як показано в прикладі 8.53.

Приклад 8.52. Правило з умовами not перед or

```
(defrule  example
            (a ?x)
            (not (or (b ?x)
            (c ?x)))
            =>)
```

Приклад 8.53. 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 у якості першого елементу 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 8.25. 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 8.26. Синтаксис команди list-defrules

```
(list-defrules <ім'яі-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 8.27. Синтаксис команди undefrule

```
(undefrule  <ім'яі-правила>)
```

Як параметр команда undefruie приймає ім'я правила, яке потрібно видалити. Якщо як ім'я правила був заданий символ *****, то будуть вилучені всі правила.

Для демонстрації роботи команд, наведених у цьому й наступному розділах, будемо використати наступні правила:

Приклад 8.54. 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.

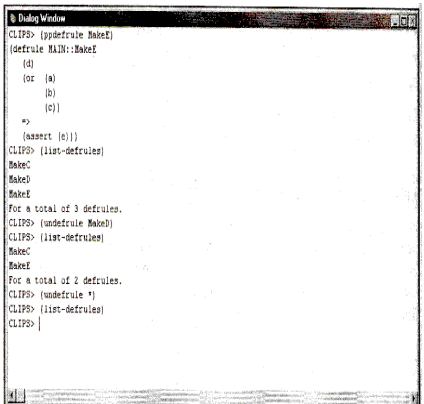

Рис. 8.6. *Результат застосування команд ppdefrule, list-defrules й undefrule*

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. 8.7. 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.

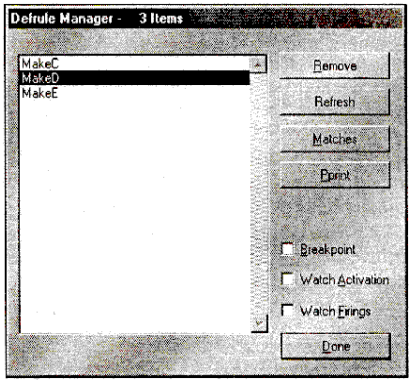

Рис. 8.7. *Перегляд списку правил за допомогою менеджера правил*

CLIPS не містить спеціальних команд для зміни існуючих правил. Щоб змінити існуюче правило, користувачеві необхідно заново визначити таке правило за допомогою конструктора defrule. При цьому існуюче 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 участи 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 лапки. 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.

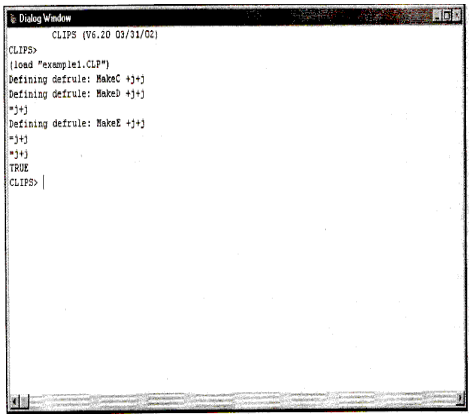

Рис. 9.8. *Результат завантаження файлу example1.CLP*

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 8.29. 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 8.30. 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 (а саме load, save, bsave й bload) доступні в меню **File** Windows-версії середовища CLIPS. Це команди **Load, Save, Save Binary й Load Binary** відповідно. Усі використають 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 необов'язковим аргументом команди run. У найпростішому випадку як цей аргумент можна використати будь-яку целую 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 поле **Step Rule Firing Increment.**

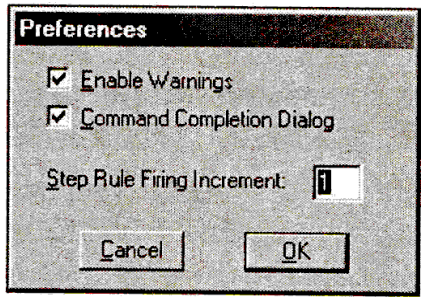

Рис. 8.9. *Діалогове вікно* ***Preferences***

Крім виконання програми по кроках, CLIPS дозволяє установку точок зупину (breakpoints) на окремих правилах.

Визначення 8.32. 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.

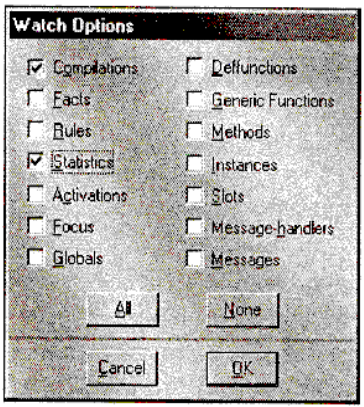

Рис. 8.10. Установка режиму з висновком статистичної інформації

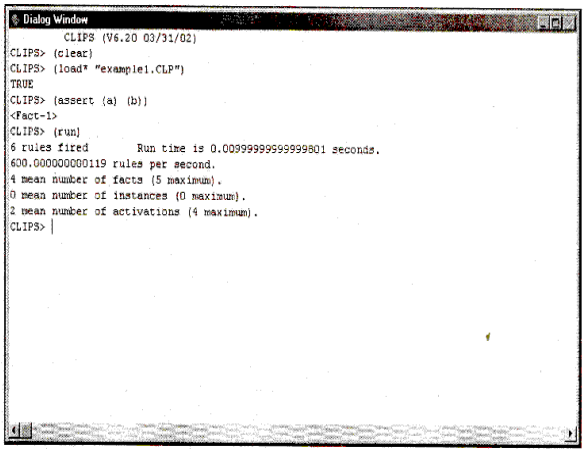

Рис. 8.11. *Одержання статистичної інформації*

Якщо ви встановите прапорець **Statistics,** завантажимо файл example 1.CLP, додамо факти (а) і (Ь) і запустимопрограму, 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, завантажите файл example1.CLP, додайте факти a, b, c и d і викличте команду agenda. Отриманий результат повинен відповідати наведеному на мал. 8.12.

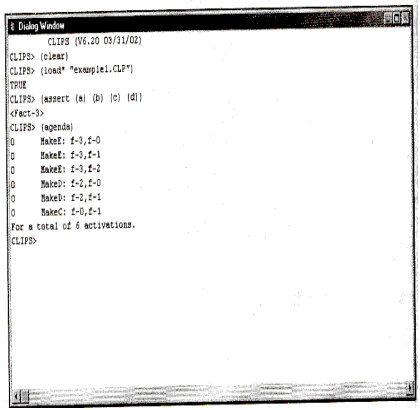

Рис. 8.12. *Перегляд плану вирішення задачі*

План вирішення 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, скористайтеся 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.

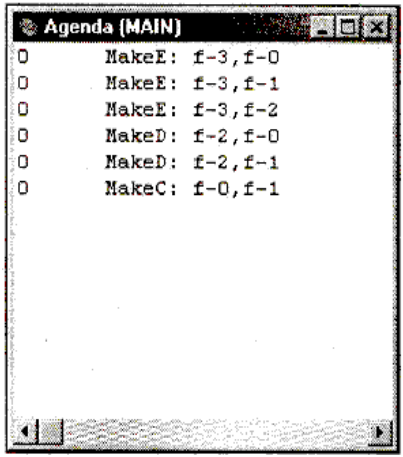

Рис. 8.13. *Вікно* ***Agenda***

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.

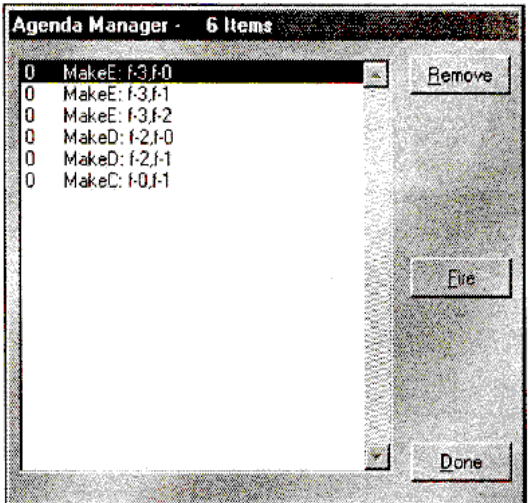

Рис. 8.14. *Вікно менеджера плану рішення задачі*

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. Подивитеся 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.

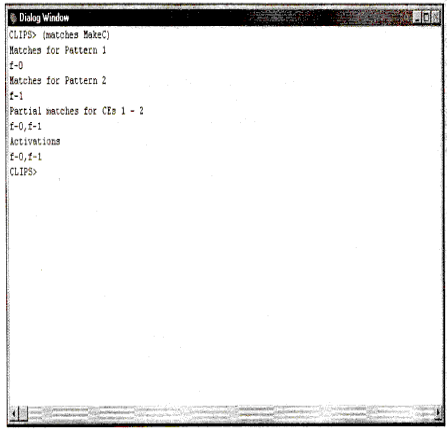

Рис. 8.15. *Дані, що активують правило Make*

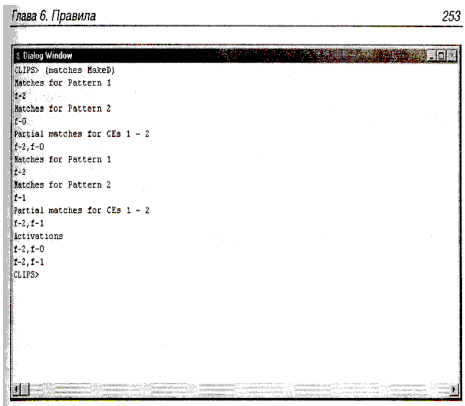

Рис. 8.16. *Приведемо кілька прикладів використання зразків об'єктів.*

Приклад 8.22. 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))
```

В наступній лекції ми розглянемо парадигму нечіткого програмування та FuzzyCLIPS.