МЕТОДИ І СИСТЕМИ ШТУЧНОГО ІНТЕЛЕКТУ

3 курс, весна 2021

- Доц. Баклан І.В.
- Email: iaa@ukr.net
- Web: baklaniv.at.ua

Лекція 13

CLIPS. Правила (продовження)

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 13.15. 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.

Приведемо кілька прикладів використання зразків об'єктів.

Приклад 13.22. 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.

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 13.16. 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 13.23. Правило del-data-facts

```
(defrule  del-data-facts
          ?data-facts  <-   (data  $?)
          =>
          (retract  ?data-facts))
```

6.5. 2. Умовний елемент *test*

Умовний елемент test надає можливість накладення додаткових обмежень на слоти фактів або об'єктів. Елемент test задовольняється, якщо викликана в ньому функція повертає значення НЕ-FALSE. Як й у випадку предикатних обмежень зразка в умовному елементі test, можна використати змінні, уже зв'язані зі своїми значеннями. Усередині елемента test можуть бути виконані різні логічні операції, наприклад порівняння змінних.

Визначення 13.17. Синтаксис умовного елемента test

<умовний-елемент-test > ::= (test <виклик-функції>)

Вираження test обчислюється щораз при задоволенні інших умовних елементів. Це означає, що умовний елемент test буде обчислений більше одного разу, якщо оброблюване вираження може бути задоволене більш ніж однією групою даних. Використання умовного елемента test може стати причиною автоматичного додавання правилу деяких умовних виражень. Крім того, CLIPS може автоматично з умовні елементи test .

Наведене нижче правило знаходить пару фактів data, причому різниця між значеннями перших полів цих фактів повинна бути більше або рівної 3.

Приклад 13.24. Застосування умовного елемента test

```
(defrule  example
          (data ?x)
          (data ?y)
          (test (>= (abs (- ?y ?x) ) 3))
          =>)
```

Умовний елемент test може привести до автоматичного додавання зразків initial-fact або initial-object у ліву частину правила. Тому не забувайте використати команду reset (яка створює initial-fact й

initial-object), щоб бути впевненим у коректній роботі умовного елемента test.

6.5. 3. Умовний елемент *or*

Умовний елемент or дозволяє активувати правило кожним з декількох заданих умовних елементів. Якщо який-небудь із умовних елементів, об'єднаних за допомогою or, удоволений, то й все вираження or уважається вдоволеним. У цьому випадку, якщо всі інші умовні елементи, що входять у ліву частину правила (але не вхідні в or), також задоволені, правило буде активовано. Умовний елемент or може поєднувати будь-яка кількість елементів.

Зауваження

Правило буде активовано для кожного вираження в умовному елементі or, що було задоволено. Таким чином, умовний елемент or робить ефект, ідентичний написанню декількох правил зі схожими посилками й

наслідками.

Визначення 13.18. Синтаксис умовного елемента or

```
<умовний-елемент-or > ::= (or <умовний-
елемент>+)
```

Приклад 13.25. Застосування умовного елемента or

```
(defrule system-fault
     (error-status unknown) (or (temp high)
     (valve broken)
          (pump off))
           =>
          (printout t "The system has a
          fault." crlf))
```

Дане правило повідомить про поломку системи, якщо в списку фактів буде присутній факт error-status unknown й один з фактів temp high, valve broken або pump off. У випадку якщо будуть присутні два із цих трьох фактів, наприклад temp high й pump off, те повідомлення буде виведено два рази.
Помітьте, що наведений приклад - точний еквівалент наступних трьох окремих правил:

Приклад 13.26. Еквівалент правилу system-fault

```
(defrule system-fault-1
     (error-status unknown)
     (pump off)
     =>
```

```
     (printout t "The system has a fault."
     crlf))
(defrule  system-fault-2
     (error-status  unknown) (valve broken)
     =>
     (printout t "The system has a fault."
     crlf))
(defrule system-fault-3
     (error-status unknown) (temp high)
     =>
     (printout t "The system has a fault."
     crlf))
```

6.5. 4. Умовний елемент *and*

Всі умовні елементи в лівій частині правил CLIPS об'єднані неявним умовним елементом and. Це означає, що всі умовні елементи, задані в лівій частині, повинні задовольнитися, для того щоб правило було активовано. За допомогою явного застосування умовного елемента and можна змішувати різні умови and й or і групувати елементи так, як цього вимагає логіка правил. Умова and задовольняється, тільки якщо всі умови усередині явного and задоволені. У випадку, якщо решта умов у лівій частині правила також щирі, правило буде активовано. Елемент and може поєднувати будь-яке число умовних елементів.

Визначення 13.19. Синтаксис умовного елемента and

```
<умовний-елемент-and> ::= (and <умовн-
елемент>+)
```

Приклад 13.27. Застосування умовного елемента and

```
(defrule system-flow
     (error-status confirmed)
     (or    (and  (temp high)
       (valve closed))
(and  (temp low)
```

```
      (valve open)))
     =>
      (printout t "The system is having a flow
     problem. " crlf))
```

Якщо умовний елемент and містить умовні елементи test або not як перший елемент, то перед ними автоматично додається зразок initial-fact або initial-object. Пам'ятаємо, що ліва частина будь-якого правила містить неявний елемент and, тому наведене в прикладі 13.28 правило буде автоматично перетворений (див. приклад 13.29).

Приклад 13.28. Правило nothing-to-schedule

```
(defrule  nothing-to-schedule
```

```
(not   (schedule   ?))
=>
(printout  t   "Nothing to
schedule."  crlf))
```

Приклад 13.29. Перетворене правило nothing-to-schedule

```
(defrule nothing-to-schedule
         (and (initial-fact)
              (not (schedule ?)))
         =>
         (printout t "Nothing to schedule."
         crlf))
```

6.5. 5. Умовний елемент *not*

Іноді важливіше відсутність інформації, а не її присутність, тобто виникають ситуації, коли необхідно запустити правило, якщо зразок або інший умовний елемент не задовольняється (наприклад, факт не існує). Умовний елемент not надає цю можливість. Елемент not задовольняється, тільки якщо умовний елемент, що він містить, не задовольняється.

Визначення 13.20. Синтаксис умовного елемента not

```
<умовний-елемент-not>   ::=   (not
<умовний-елемент>)
```

Умовний елемент not може заперечувати тільки одне

вираження. Кілька умовних елементів потрібно заперечувати за допомогою декількох елементів not. Ретельно стежите за комбінаціями not з or або and; результат не завжди очевидний!

Приклад 13.30. Застосування умовного елемента not

```
(defrule high-flow-rate
          (temp high)
          (valve open)
          (not (error-status confirmed))
          =>
          (printout t "Recommend closing of
          valve due to high temp" crlf))
```

У логічному елементі not можна використати зв'язані змінні, так само як й в інших умовних елементах:

Приклад 13.31. Правило check-value

```
(defrule check-valve
     (check-status  ?valve)
          (not   (valve-broken  ?valve))
     =>
     (printout  t  "Device   "   ?valve   "
     is  OK"  crlf))
```

За допомогою умовного елемента not можна, нарешті, довести до досконалості наше правило Find-2-coeval-Person.

Якщо ви пам'ятаєте, це правило виводить усілякі пари персон однакового віку. Щоб дане правило не виводило еквівалентні за змістом пари імен (наприклад, Bob-Sue й Sue-Bob), перетворимо нашу програму наступним образом:

Приклад 13.32. Поліпшене правило Find-2-coeval-Person

```
(deftemplate person
             (slot name)
             (slot age))
(deftemplate person-pair
             (slot name1)
             (slot name2)
             (slot age))
(deffacts   people
             (person (name Joe)  (age 20))
             (person (name Bob)  (age 20))
             (person (name Joe)  (age 34))
             (person (name Sue)  (age 34))
             (person (name Sue)  (age 20)))
```

```
(defrule    Find-2-Coeval-Person
     (person (name ?x) (age ?z))
     (person (name ?ys~?x) (age ?z))
          (not (person-pair (namel ?x)
          (name2 ?y) (age ?z)))
          (not (person-pair (namel ?y)
          (name2 ?x) (age ?z)))
     =>
     (printout t "name=" ?x " name=" ?y "
     age=" ?z crlf)
     (assert (person-pair (namel ?x) (name2 ?
     y) (age ?z))))
```

Зверніть увагу на зроблені зміни. По-перше, за допомогою

конструктора deftemplate був Доданий додатковий шаблон person-pair. У фактах, що відповідають даному шаблону, буде зберігатися інформація про вже знайдені пари ровесників.

Крім того, було сильно змінене й саме правило. У його лівій частині було додано дві умови:

```
(not (person-pair (name1 ?x) (name2 ?y) (age ?z)))
(not (person-pair (name1 ?y) (name2 ?x) (age ?z)))
```

Ці умовні елементи перевіряють наявність фактів типу person-pair

й, тим самим відслідковують, чи була вже оброблена дана пара або її перестановка. Якщо ці факти відсутні, то це означає, що обробка ще не була виконана. У цьому випадку правило активується, і виконуються дії, описані в правій частині правила. А саме виводиться на екран повідомлення про знайдену пару ровесників і додається відповідний факт person-pair, що затверджує, що дана пара вже була оброблена.

Для запуску програми виконайте команди reset й run. Програма виведе на екран наступну інформацію:

Приклад 13.33. Результат роботи правила Find-2-Coeval-Person

```
name=Sue name=Bob  age=20
name=Sue name=Joe  age=20
name=Sue name=Joe  age=34
name=Bob name=Joe  age=20
```

Якщо ви уважно подивитеся на отриманий результат і вихідні дані, то виявите, що це саме те, що нам було потрібно. Це список усіляких ровесників без повторень і з виключенням того факту, що всі люди є ровесниками самі собі. Тепер наше правило досягло повної досконалості! Зверніть увагу на той факт, що якщо ви повторно спробуєте виконати команду run, те нічого не побачите. Це відбувається тому, що в списку фактів утримується інформація про всі оброблені пари, що залишилася після першого запуску. Для того щоб повторно запускати даний приклад, виконуйте команду reset перед кожною командою run.

Умовний елемент not, так само як й test, може привести до автоматичного додавання зразків initial-fact або initial-object у лівій частині правил. Тому не забувайте використати команду reset (яка створює initial-fact й initial-object), щоб бути впевненим у

коректній роботі умовного елемента not.

В умовний елемент not, що містить елемент test, автоматично перетвориться в елемент not, що містить and з initial-fact і вихідним елементом test. Наприклад, що випливає умовний елемент із приклада 13.34 перетвориться в елемент із приклада 13.35.

Приклад 13.34. Умовний елемент not, що містить елемент test

```
(not   (test   (>  ?time-1   ?time-2)))
```

Приклад 13.35. Перетворений умовний елемент not, L утримуючий елемент test

```
(not (and   (initial-fact)
```

```
(test (> ?time-1 ?time-
2))))
```

Зауваження

Помітьте, що найбільш простим і правильним способом запису даного виразу буде:

```
(test   (not   (>  ?time-1  ?time-2)
) ).
```

9.5. 13. Умовний елемент *exists*

Умовний елемент exists дозволяє визначити, чи існує хоча б один набір даних (фактів або об'єктів), які задовольняють умовним елементам, заданим усередині елемента exists.

Визначення 13.21. Синтаксис умовного елемента exists

```
<умовний-елемент-exists>   ::=   (exists
<умовн-елемент>+)
```

CLIPS автоматично заміняє exists двома послідовними умовними елементами not. Наприклад, що випливає правило (приклад 13.36) буде перетворено в правило із приклада 13.37.

Приклад 13.36. Правило example

```
(defrule example
         (exists   (a  ?x)     (b  ?x))=>)
```

Приклад 13.37. Перетворене правило example

```
(defrule example
```

```
(not (not (and (a ?x) (b ?x))))=>)
```

Тому що внутрішній спосіб реалізації exists використовує умовний елемент not, то для exists справедливі всі зауваження й обмеження, наведені в попередніх лекціях.

Розглянемо наступний приклад:

Приклад 13.38. Використання умовного елемента exists

```
(deftemplate      hero
          (multislot name)
          (slot status  (default unoccupied)))
(deffacts                goal-and-heroes
          (goal save-the-world)
          (hero   (name Death Defying Man))
          (hero   (name  Stupendous Man))
          (hero   (name  Incredible Man)))
(defrule                   save-the-world
          (goal  save-the-world)
```

```
(exists (hero (status
unoccupied)))
=>
(printout t "The day is saved."
crlf))
```

Дана програма визначає шаблон - героя, що має складене поле з ім'ям героя й простої поле, що містить статус "не зайнятий" за замовчуванням. Конструктор deffacts визначає трьох нічим не зайнятих героїв і поточну мету - порятунок миру. Правило перевіряє, є в цей момент ця мета, і у випадку позитивної відповіді перевіряє, якщо чи який-небудь ще не зайнятий герой. Якщо всі умовні елементи правила задоволені, воно повідомляє, що мир урятований. Зверніть увагу: незважаючи на те, що в нас всі

три герої не зайняті, правило буде активовано тільки один раз.

Тому що спосіб реалізації exists використовує умовний елемент not, те умовний елемент exists може привести до автоматичного додавання зразків initial-fact або initial-object у ліву частину правила. Тому не забувайте використати команду reset (яка створює initial-fact й initial-object), щоб бути впевненим у коректній роботі умовного елемента exists.

6.5. 7. Умовний елемент *forall*

Умовний елемент forall дозволяє визначити, що деяка задана умова виконується для всіх заданих умовних елементів.

Визначення 13.22. Синтаксис умовного елемента forall

```
<умовний-елемент forall> ::= (forall
      <умовний-елемент>
                                  <умовний-
                                  елемент>+)
```

CLIPS автоматично заміняє f про rail комбінацію умовних елементів not й and. Наприклад, що випливає правило (приклад 13.39) буде перетворено так, як показано в прикладі 13.40.

Приклад 13.39. Правило example

```
(defrule  example
          (forall   (a  ?x)     (b ?x)     (c  ?
          x))=>)
```

Приклад 13.40. Перетворене правило example

```
(defrule example
          (not      (and   (a ?x)
```

```
(not   (and     (b ?x)
                (c  ?
               x)))))=>)
```

Розглянемо наступний приклад. Правило all-students-passed визначає, чи пройшли всі студенти читання, правопис й арифметику, використовуючи умову forall:

Приклад 13.41. Правило all-students-passed

```
(defrule all-students-passed
         (forall (student ?name)
                 (reading ?name)
                 (writing ?name)
                 (arithmetic ?name))
         =>
         (printout t "All students passed."
         crlf))
```

Помітьте, що дане правило задовольняється, поки немає жодного студента. При додаванні факту (student Bob) правило перестає задовольнятися, тому що немає фактів, що підтверджують, що Bob пройшов всі необхідні предмети. Правило не почне задовольнятися й після додавання фактів (reading Bob) і (writing Bob). А от після додавання факту (arithmetic Bob) правило буде активоване й зможе вивести на екран відповідний запис. Якщо додати факт (student John), правило знову перестане задовольнятися, тому що один зі студентів (John) не пройшов всі необхідні предмети. Використовуючи умовний елемент exists, ви без праці зможете змінити це правило так, щоб воно не виконувалося у випадку відсутності студентів. Тому що реалізація forall використовує умовний елемент not, те forall, так само як й not, test й exists, може привести до автоматичного додавання зразків initial-fact або initial-object у ліву частину правила. **Не** забувайте використати команду reset

для коректної роботи цього умовного елемента.

6.5. 8. Умовний елемент *logical*

Умовний елемент logical надає механізм підтримки вірогідності для створених правилом даних (фактів або об'єктів), що задовольняють зразкам. Дані, створені в правій частині правила, можуть мати логічну залежність від даних, що задовольнила зразки в лівій частині правила. Така залежність називається *логічною підтримкою*. Дані можуть залежати від групи даних або декількох груп даних, що задовольнили одне або кілька правил. Якщо віддаляються дані, які підтримують деякі інші дані, то залежні дані також автоматично віддаляються.

Якщо деякі дані створені без логічної підтримки (наприклад, за допомогою конструкторів deffacts, definstance або команди assert, уведеної користувачем або викликаної в правій частині

правила), то вважається, що вони мають безумовну підтримку. Безумовна підтримка видаляє всі присутні в цей момент умовні підтримки цих даних (але не видаляє самі дані). Подальша логічна підтримка для даних з безумовною підтримкою ігнорується. Видалення правила, що викликало логічну підтримку для даних, видаляє логічну підтримку, згенеровану цим правилом (але не видаляє дані, якщо в них ще є логічна підтримка, згенерована іншим правилом).

Визначення 13.23. Синтаксис умовного елемента logical

```
<умовний-елемент-logical> ::= (logical
                                  <умовний-
                                  елемент>+)
```

Умовний елемент logical групує зразки, так само як це робить and. Дана властивість можна використати при об'єднанні елементів and, or й not. Однак тільки перші *n* зразків правила можуть використатися в умовному елементі logical. Наприклад, що випливає правило записане вірно:

Приклад 13.42. Правильний варіант використання умовного елемента logical

```
(defrule  ok
          (logical (a))
          (logical (b))
          (c)
```

```
=>
(assert (d))
```

А таке оголошення правил неприпустиме:

Приклад 13.43. Неправильні варіанти використання умовного елемента logical

```
(defrule not-ok-1
        (logical (a))
        (b)
        (logical (c))
        =>
        (assert (d)))
(defrule not-ok-2
        (a)
        (logical (b))
        (logical (c))
        =>
        (assert (d)))
(defrule not-ok-3
```

(or (a)
(logical (b)))
(logical (c))

=>

(assert (d)))

Розглянемо наступний приклад. Включите перегляд списку фактів за допомогою пункту **Facts Window** меню **Windows,** це допоможе стежити за тим, що відбувається в момент виконання програми.

Приклад 13.44. Використання умовного елемента logical

```
(clear)
(reset)
(defrule example
          (logical   (a))
          (b)
```

```
          =>
           (assert (c)))
(assert (a) (b) )
(run)
(retract 2)
(retract 1)
```

По команді run правило example, активоване фактами a й b, додає новий факт із, що має логічну підтримку (залежить) від факту a.

Після видалення факту b за допомогою команди (retract 2) нічого особливого не відбувається, але якщо ми видалимо факт a, те побачимо, що це відразу приведе до видалення пов'язаного з ним факту c.

Як згадувалося , умовний елемент logical може бути використаний для створення даних, які будуть логічно пов'язані зі змінами деяких окремих слотів об'єкта, а не від усього об'єкта цілком. Дану можливість можна використати тільки при роботі з об'єктом. При роботі із шаблонами фактів дану можливість використати не можна, тому що зміни слоту факту фактично приводить до видалення старого факту й додавання нового зі зміненими слотами й індексом.

На відміну від фактів зміни слотів об'єкта виконуються без видалення об'єкта. Це поводження ілюструється наведеним нижче приміром:

Приклад 13.45. Використання умовного елемента logical з об'єктами

```
(clear)
(defclass A (is-a USER)
            (role concrete)
            (pattern-match reactive)
            (slot foo (create-accessor write))
            (slot bar (create-accessor write)))
(defrule  match-A
            (logical (object (is-a A) (foo ?))))
```

```
        =>
         (assert (new-fact)))
(make-instance a of A)
(run)
(send [a] put-foo 100)
```

Після виконання команди run правило match-A додає факт new-fact, логічно пов'язаний з конкретним значенням слоту foo об'єкта a. При зміні значення даного слоту факт new-fact автоматично віддаляється зі списку фактів.

6.5.9. 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 13.24. Синтаксис визначеного факту й об'єкта

```
(initial-fact)
(object (is-a INITIAL-OBJECT) (name [initial-
object]))
```

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 13.46, буде перетворено так, як показано в прикладі 13.47:

Приклад 13.46. Правило без умов

```
(defrule  example
```

```
          => )
```

Приклад 13.47. Перетворене правило без умов

```
(defrule  example
          (initial-fact)
           =>)
```

Використання елементів *test* та *not* перед *and*

Умовні елементи test й not, що коштують перед and, додають зразок initial-fact або initial-object безпосередньо перед собою. Зразок initial-fact додається, якщо в першому умовному елементі використовується зразок факту. Зразок initial-object додається, якщо в першому умовному елементі використовується зразок об'єкта. Якщо в першому умовному елементі немає зразків, то тип зразка, що додає, визначається по наступному умовному елементі таким же методом. Якщо у всьому поточному умовному вираженні немає зразків, то система використає визначений факт initial-fact (хоча конфігурацію CLIPS можна настроїти таким чином, щоб замість зразка initial-fact додавався зразок initial-object).

Наприклад правила, що випливають із приклада 13.48, будуть змінені так, як у прикладі 13.49.

Приклад 13.48. 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 13.49. 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)))
           =>)
```

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 13.50 зміниться на еквівалентне (приклад 13.51):

Приклад 13.50. Правило з умовами not перед елементом test

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

Приклад 13.51. Перетворене правило з умовами not перед test

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

Використання елемента *not* перед or

Якщо відразу перед умовним елементом or використався умовний елемент not, то CLIPS автоматично заміняє комбінацію not/or на еквівалентну комбінацію and/not.

Наприклад, що випливає правило (приклад 13.52) будуть змінено так, як показано в прикладі 13.53.

Приклад 13.52. Правило з умовами not перед or

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

Приклад 13.53. Перетворене правило з умовами not перед or

```
(defrule        example
                (a  ?x)
                (and   (not   (b  ?x))
                       (not   (c  ?x)))
```

=>)

Зауваження про автоматичне додавання й перегрупування умовних елементів

У завершення опису синтаксису лівої частини правил CLIPS оборотна увага на наступні важливі особливості:

1. Повна версія лівої частини правила містить неявний умовний елемент and.

2. Перетворення умовних елементів forall й exists до еквівалентних виражень за допомогою not й and виконується перед додаванням відповідних зразків у ліву частину правила.
3. Умовний елемент test звичайно не використовується як

перший елемент в умові and.

4. 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.

9.6. 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. Повна специфікація цих функцій буде дана в *розд. 15* й *16*, у даній лекції ми розглянемо лише основні з них із прикладами використання.

9.6.1. 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 13.25. 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 13.26. Синтаксис команди list-defrules

```
(list-defrules <ім'я-модуля>)
```

Повний синтаксис цієї команди містить необов'язковий аргумент <ім'я-модуля> (про поняття модуля буде розказане в *розд. 12)*. 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 13.27. 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 13.54. Необхідні для подальшої роботи правила

```
(defrule Make                    (     b))
  (a)                           =>
  (b)                           (assert
  =>                            (d)))
  (assert  (defrule              Make
  (c)))                 (d)
(defrule Make                          (or (a)
  (c)                                  (b)
        (or               (c))
  (a)                     =>
```

```
(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. 13.6.

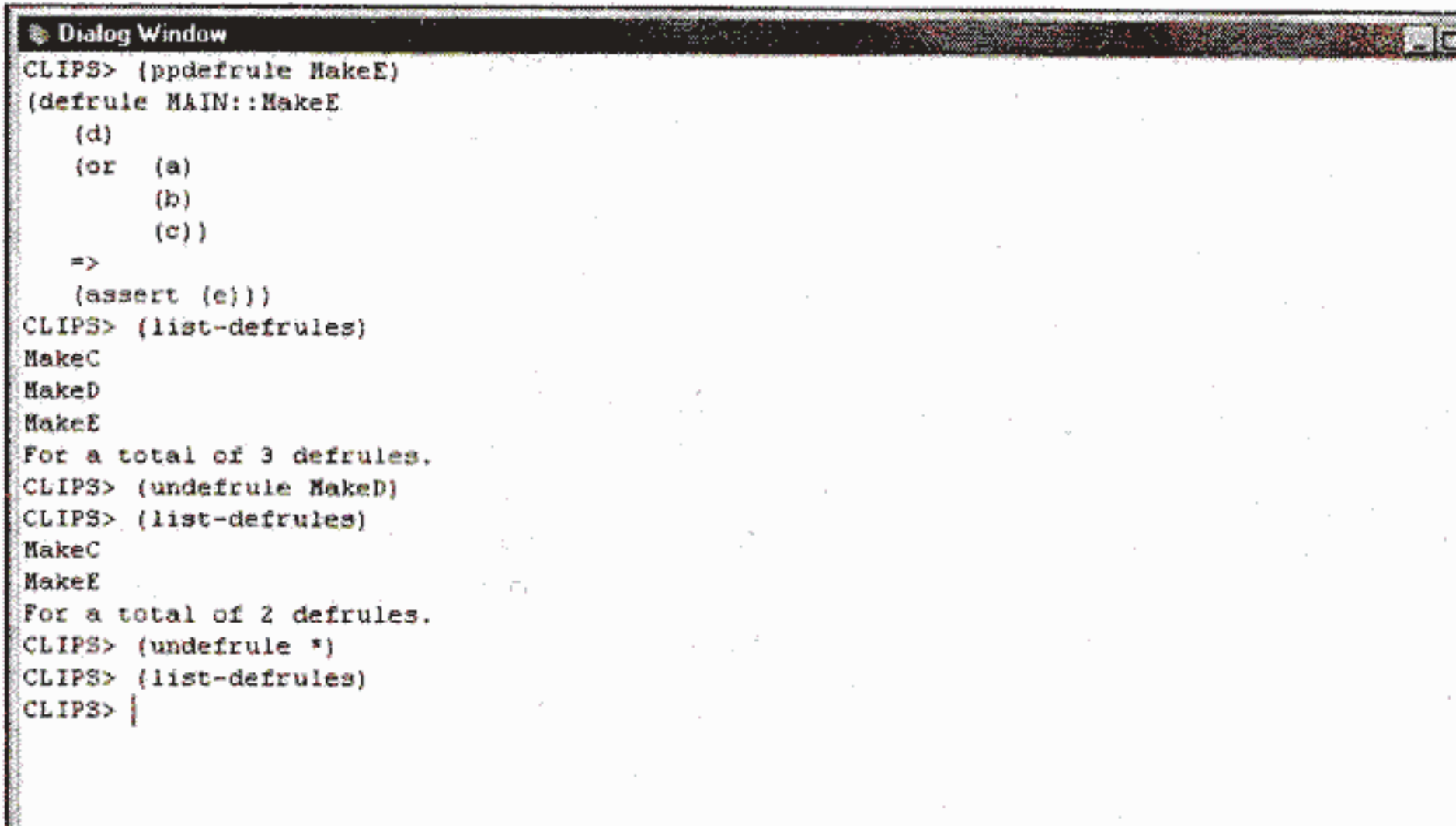
Dialog Window
CLIPS> (ppdefrule MakeE)
(defrule MAIN::MakeE
(d)
(or (a)
(b)
(c))
=>
(assert (e)))
CLIPS> (list-defrules)
MakeC
MakeD
MakeE
For a total of 3 defrules.
CLIPS> (undefrule MakeD)
CLIPS> (list-defrules)
MakeC
MakeE
For a total of 2 defrules.
CLIPS> (undefrule *)
CLIPS> (list-defrules)
CLIPS>

Рис. 13.6. 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. 13.7. 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.

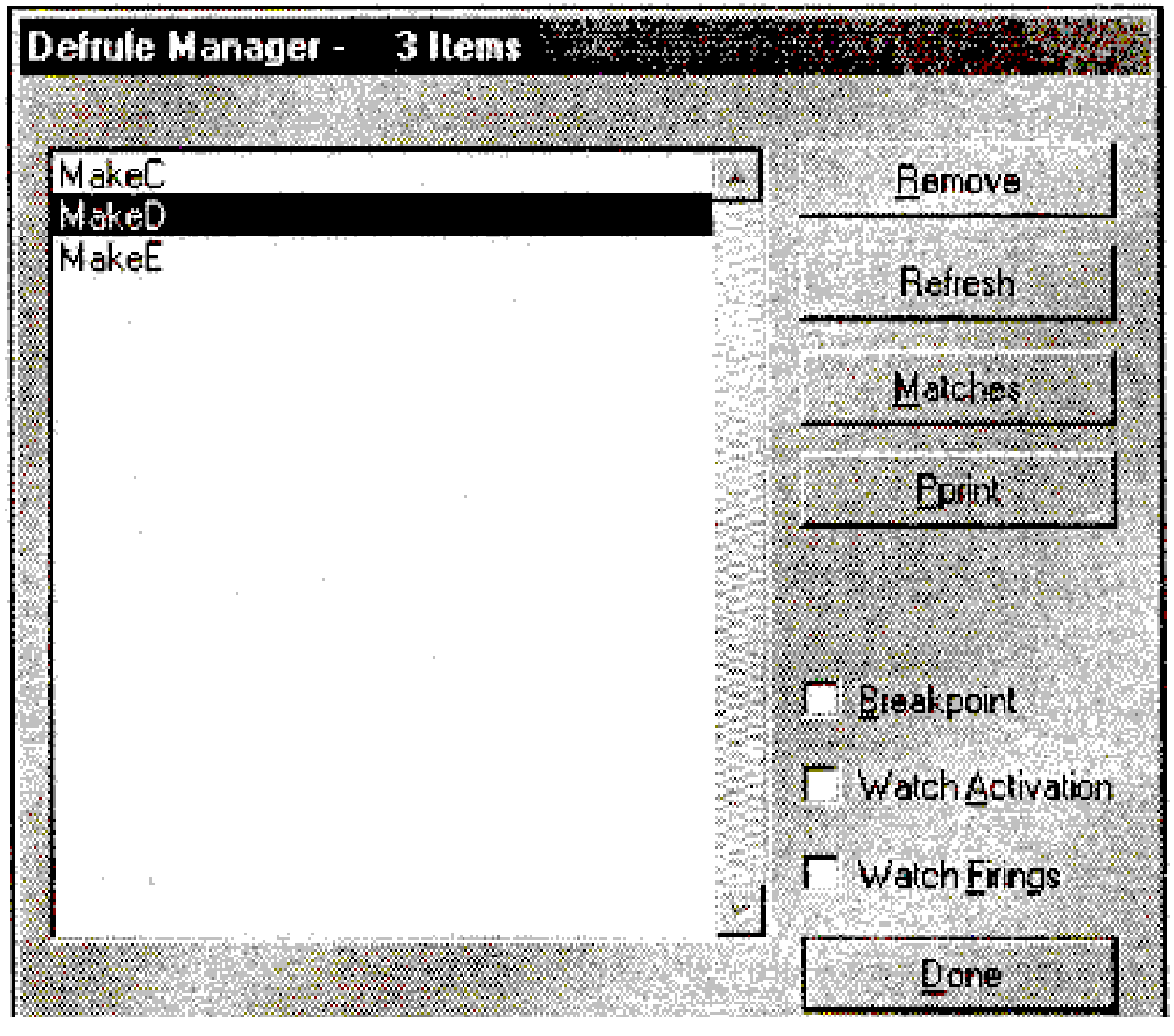
Defrule Manager - 3 Items
MakeC
MakeD
MakeE
Remove
Refresh
Matches
Print
Breakpoint
Watch Activation
Watch Firings
Done

Рис. 13.7. Перегляд списку правил за допомогою менеджера правил

CLIPS не містить спеціальних команд для зміни існуючих правил. Щоб змінити існуюче правило, користувачеві необхідно заново визначити таке правило за допомогою конструктора def rule. При цьому існуюче визначення правила буде автоматично вилучено із системи, навіть якщо новий конструктор містив помилки, і нове правило додане не було.

9.6.2. 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 13.28. 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 оцініте 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. 13.8.

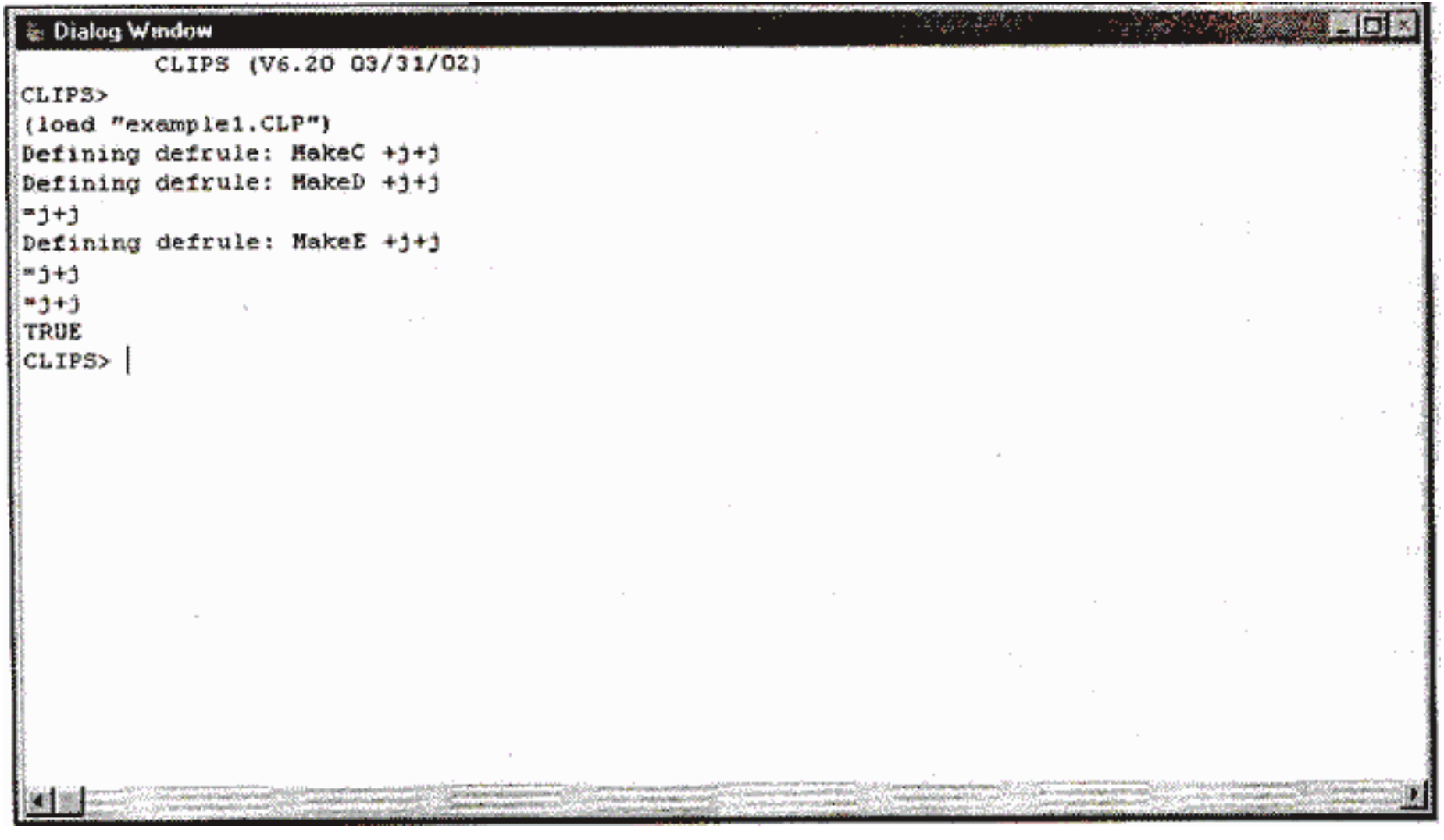
Dialog Window
CLIPS (V6.20 03/31/02)
CLIPS>
(load "example1.CLP")
Defining defrule: MakeC +j+j
Defining defrule: MakeD +j+j
=j+j
Defining defrule: MakeE +j+j
=j+j
=j+j
TRUE
CLIPS>

Рис. 13.8. 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 13.29. 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 13.30. 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.

9.6.3. Запуск і зупинка програми

Як було замічено, для запуску CLIPS-програм використовується команда run.

Визначення 13.31. Синтаксис команди run

```
(run <вцілочисельний-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. 13.9. Кількість правил, запущених за один крок трасування, відображається в поле **Step Rule Firing Increment.**

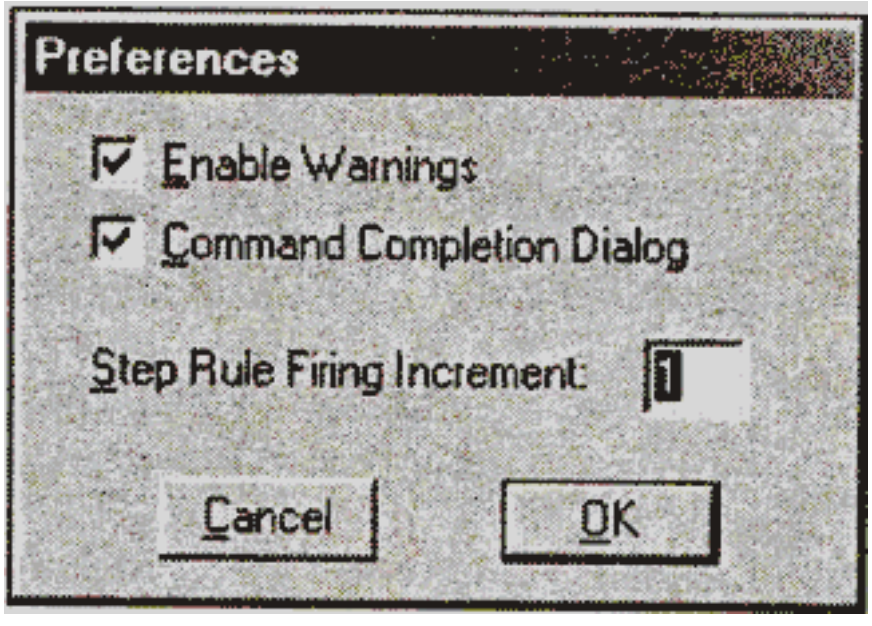

Рис. 13.9. Діалогове вікно **Preferences**

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

Визначення 13.32. 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 13.33. 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. 13.7. Для цього виберіть правило й установите прапорець **Breakpoint.**

У випадку якщо виконання програми необхідно зупинити, використайте команду halt без аргументів.

Визначення 13.34. Синтаксис команди halt

```
(halt)
```

Діалогове вікно **Watch Options** (пункт **Watch** меню **Execution)** дозволяє встановити прапорець **Statistics,** як показано на мал. 13.10.

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.

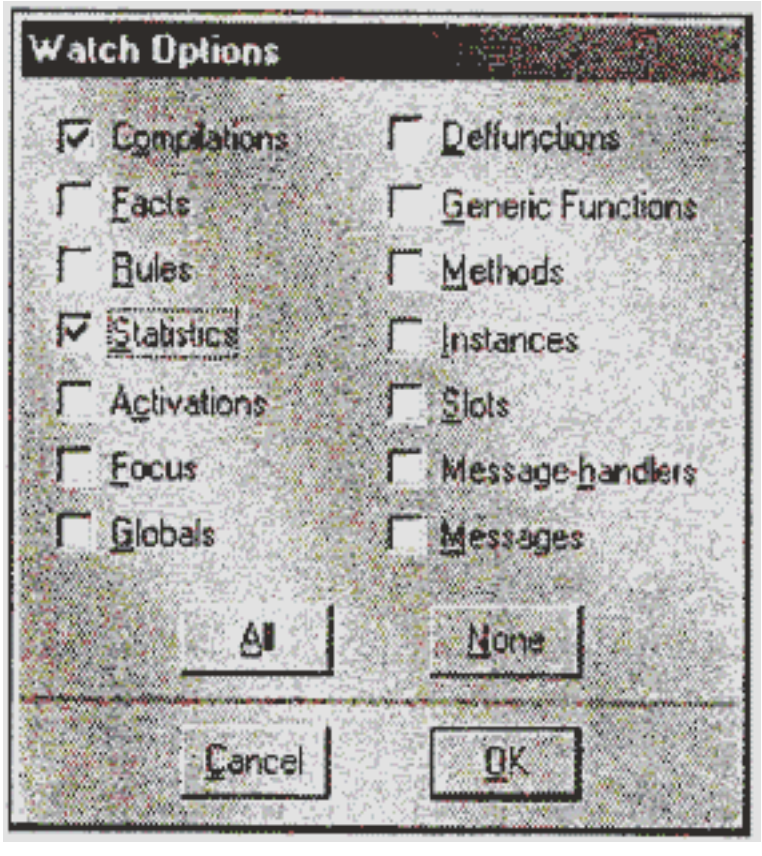

Рис. 13.10. Установка режиму з висновком статистичної

інформації

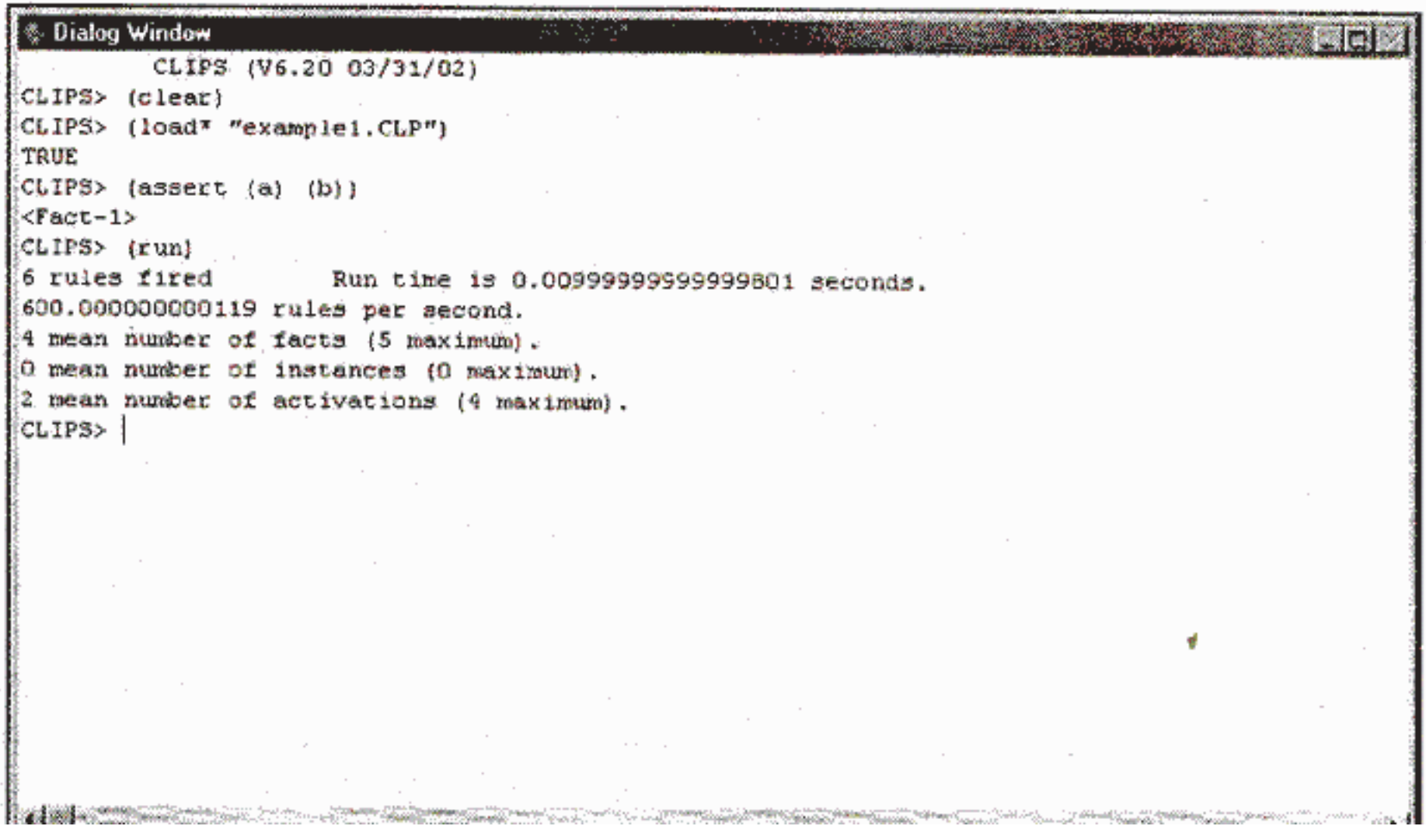
Dialog Window
CLIPS (V6.20 03/31/02)
CLIPS> (clear)
CLIPS> (load* "example1.CLP")
TRUE
CLIPS> (assert (a) (b))
<Fact-1>
CLIPS> (run)
6 rules fired Run time is 0.00999999999999801 seconds.
600.000000000119 rules per second.
4 mean number of facts (5 maximum).
0 mean number of instances (0 maximum).
2 mean number of activations (4 maximum).
CLIPS>

Рис. 13.11. 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. 13.11.

9.6.4. 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. 13.12.

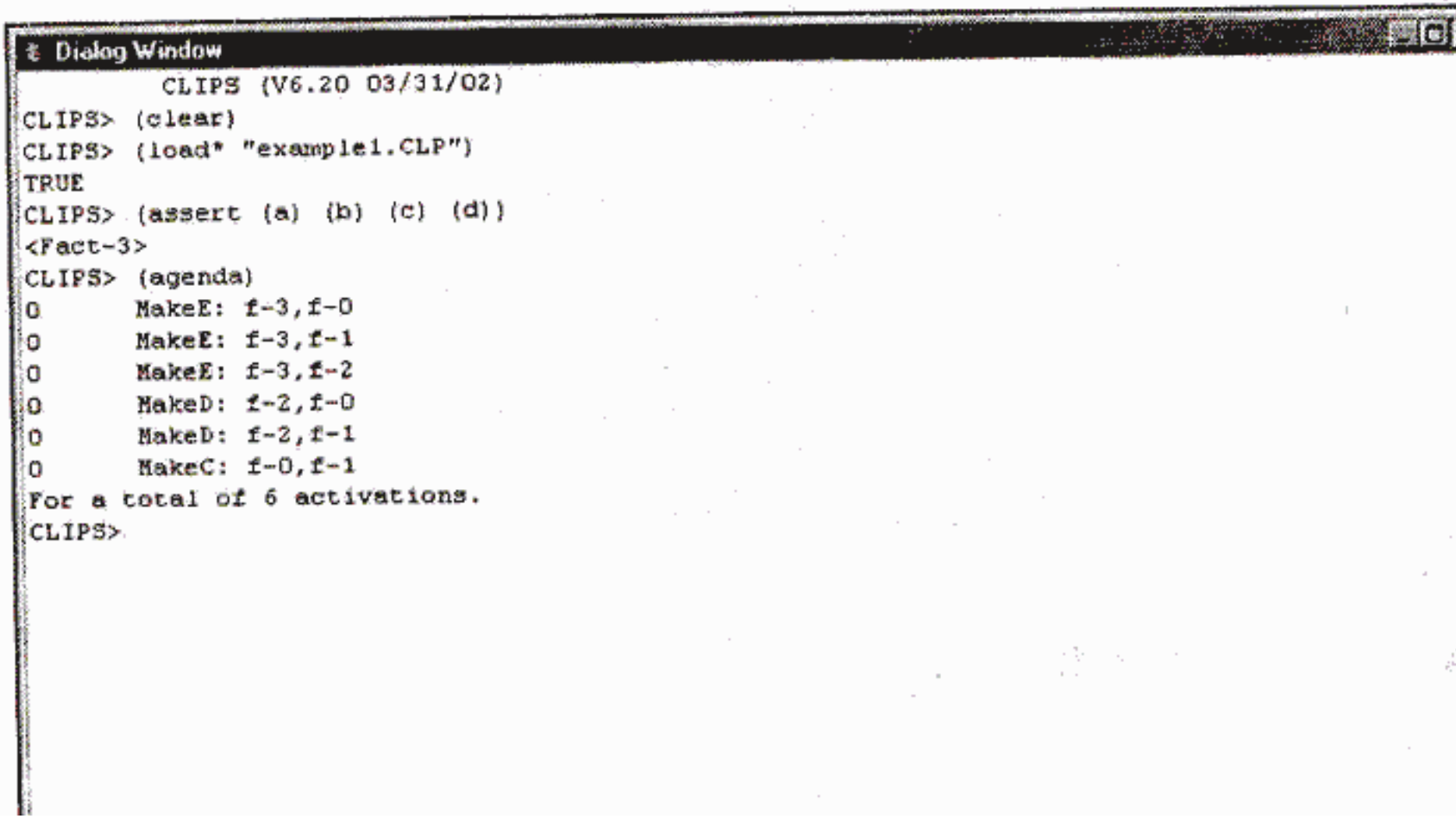
Dialog Window
CLIPS (V6.20 03/31/02)
CLIPS> (clear)
CLIPS> (load* "example1.CLP")
TRUE
CLIPS> (assert (a) (b) (c) (d))
<Fact-3>
CLIPS> (agenda)
0 MakeE: f-3,f-0
0 MakeE: f-3,f-1
0 MakeE: f-3,f-2
0 MakeD: f-2,f-0
0 MakeD: f-2,f-1
0 MakeC: f-0,f-1
For a total of 6 activations.
CLIPS>

Рис. 13.12. Перегляд плану рішення задачі

План рішення задачі містить 6 активацій правил. По команді agenda всі ці активації будуть виведені на екран разом із пріоритетом правил (ліворуч від імені правила) і списком даних, що активували правило (праворуч від імені правила). Порядок правил у плані рішення задачі сильно залежить від обраної стратегії дозволу конфліктів і пріоритету правил.

Крім цього, Windows-версія CLIPS дозволяє виводити план рішення задачі в окремому вікні — **Agenda.** Для того щоб зробити вікно видимим, скористайтеся пунктом **Agenda Window** меню **Window.** Зовнішній вигляд цього вікна показаний на мал. 13.13, 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.

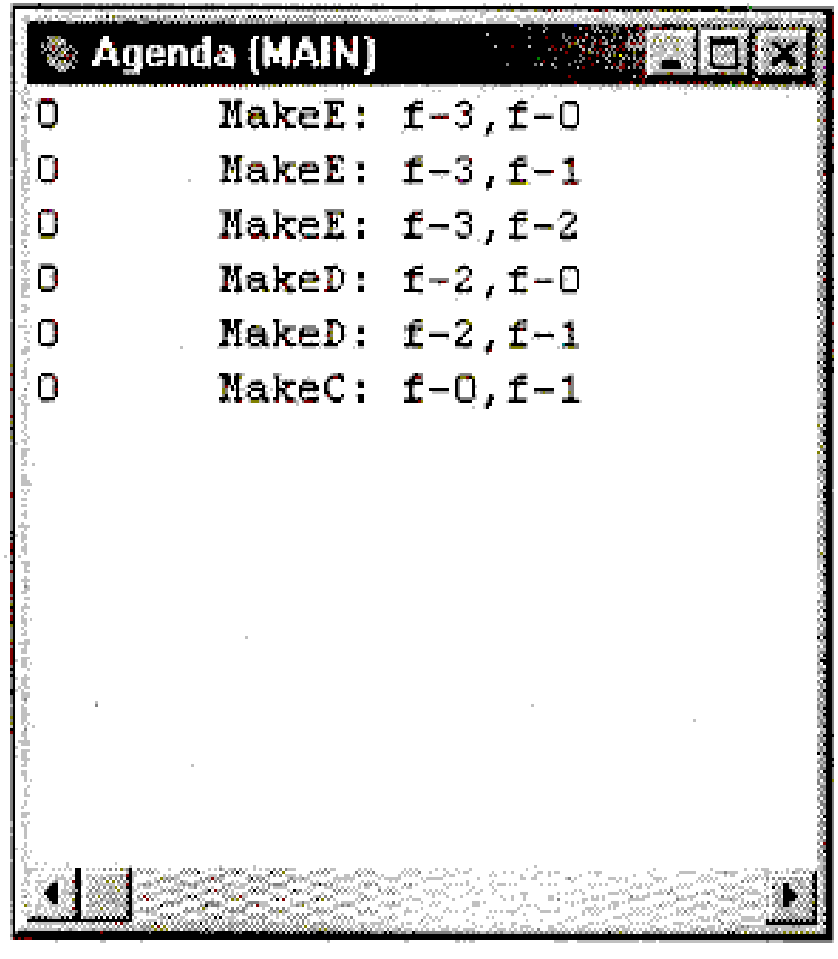

Рис. 13.13. Вікно **Agenda**

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. 13.14. З його допомогою можна видаляти із плану рішення задачі окремі активації правил або запускати правила в деякому довільному порядку.

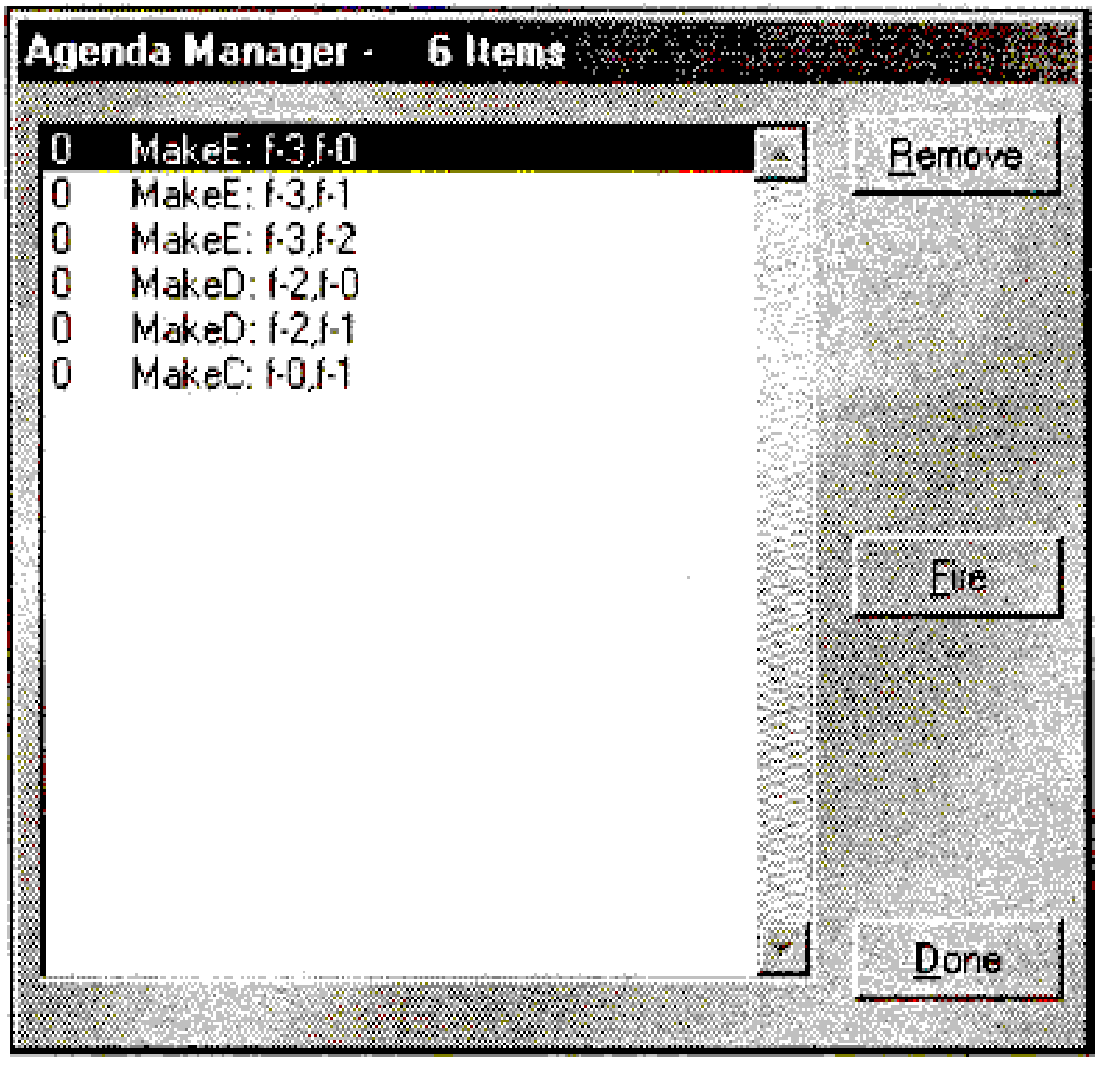
Agenda Manager - 6 Items
0 MakeE: f-3,f-0
0 MakeE: f-3,f-1
0 MakeE: f-3,f-2
0 MakeD: f-2,f-0
0 MakeD: f-2,f-1
0 MakeC: f-0,f-1
Remove
File
Done

Рис. 13.14. Вікно менеджера плану рішення задачі

За допомогою діалогового вікна **Watch Options** (див. мал. 13.10) 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.

Перегляд даних, здатних активувати правило

CLIPS надає можливість переглядати списки наборів даних (фактів або об'єктів), здатних активувати задане правило.

Визначення 13.35. 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. 13.15 й

13.16 відповідно.

На цьому ми закінчимо вивчення синтаксису правил CLIPS й основних команд і функцій для роботи з ними.

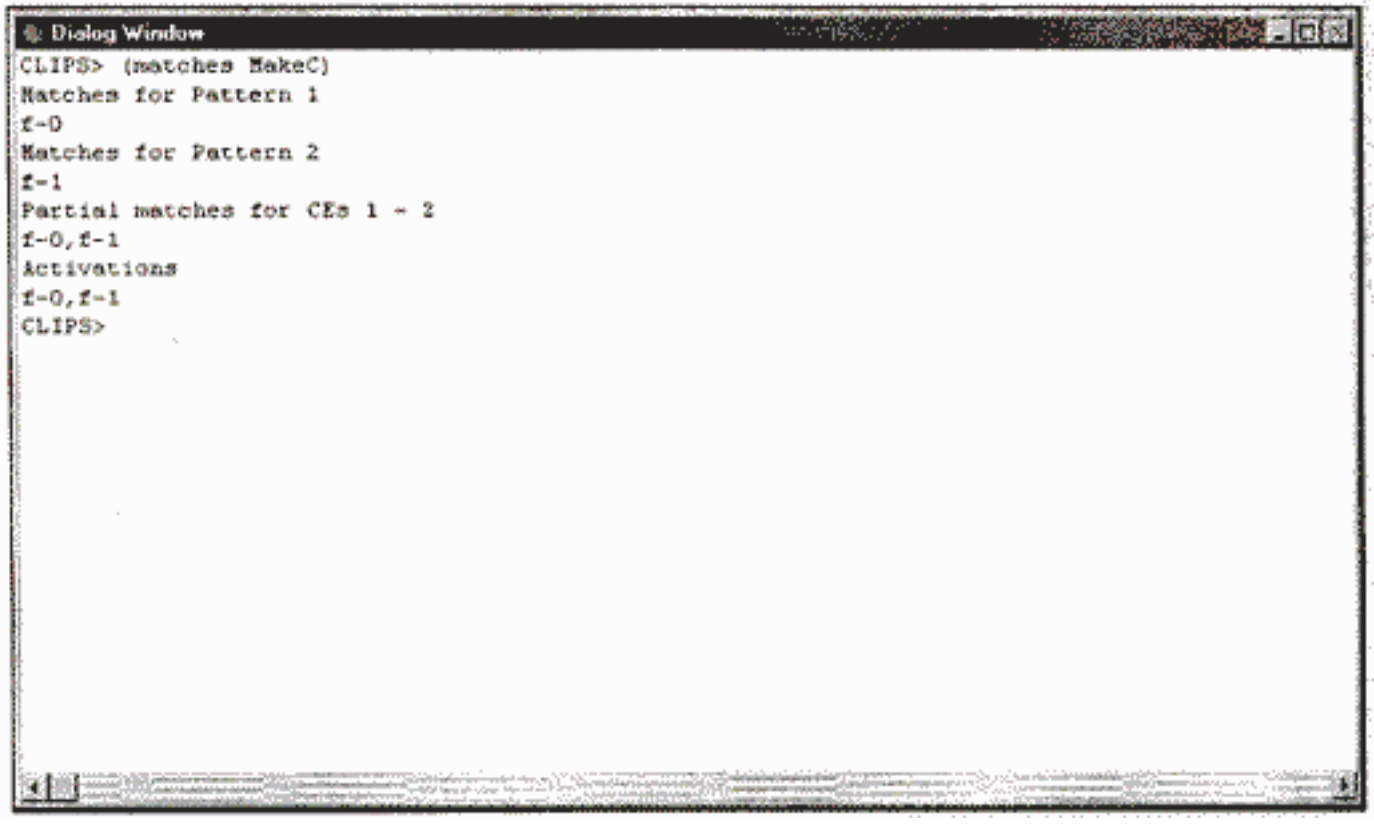
Dialog Window
CLIPS> (matches MakeC)
Matches for Pattern 1
f-0
Matches for Pattern 2
f-1
Partial matches for CEs 1 - 2
f-0,f-1
Activations
f-0,f-1
CLIPS>

Рис. 13.15. Дані, що активують правило Make

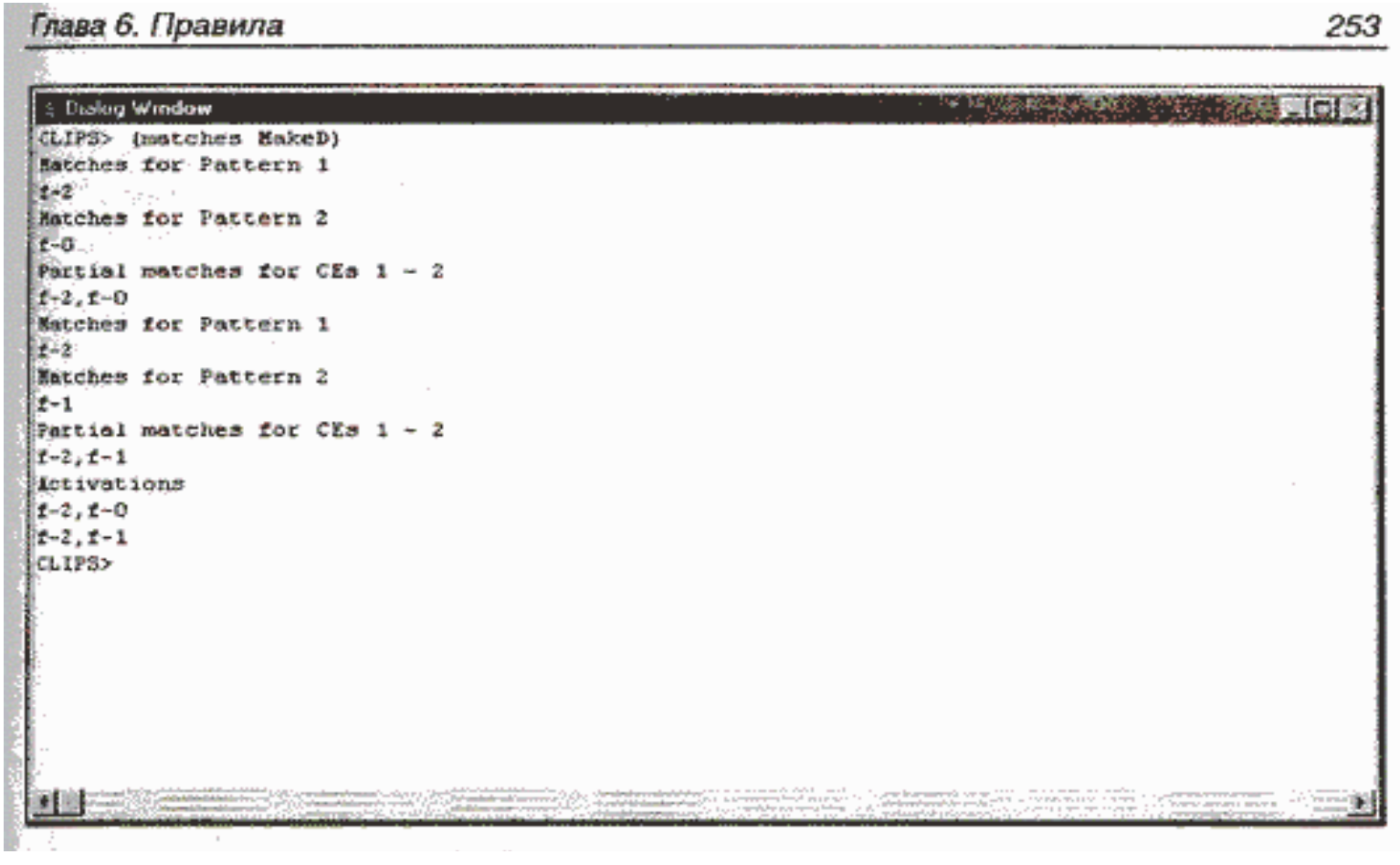

Рис. 13.16.

У наступній лекції ми ознайомимся з прикладами побудови експертних систем в CLIPS.