Програмування інтелектуальних інформаційних систем

3 курс, осінь 2021

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

Лекція 11

Програмування фактів і правил в системі CLIPS

ФАКТИ

Факти є однією з основних форм подання інформації у системі CLIPS. Кожен факт є фрагментом інформації, який був поміщений в поточний список фактів (базу фактів). Факт є основну одиницю даних, використовувану правилами.

Кількість фактів у списку та обсяг інформації, який може бути збережений у факті, обмежується лише розміром пам'яті комп'ютера. Якщо при додаванні нового факту до списку виявляється, що він повністю збігається з одним із вже включених до бази фактів, то ця операція ігнорується (хоча таку поведінку можна змінити).

У системі CLIPS фактом є список неподільних (або атомарних) значення примітивних типів даних. CLIPS підтримує два типи фактів — упорядковані факти (ordered facts) та неупорядковані факти чи шаблони (non-ordered facts чи template facts). Посилатися на дані, що містяться у факті, можна або використовуючи строго задану позицію значення списку даних для впорядкованих фактів, або вказуючи ім'я значення для шаблонів. Упорядковані факти складаються з поля, що обов'язково є даним типу **`symbol`**, і наступної за ним, можливо порожньої, послідовності полів, розділених пробілами. Обмеженням факту є круглі дужки. Факт може описуватися індексом чи адресою. Щоразу, коли

факт додається до БФ, йому присвоюється унікальний цілочисельний індекс. Зміна існуючого факту також призводить до зміни адреси. Індекси фактів починаються з нуля і кожного нового чи зміненого факту збільшуються на одиницю. Щоразу після виконання команд **reset** та **clear** виділення індексів починається з нуля. Факт може також задаватися за допомогою адреси. Адреса факту може бути отримана шляхом збереження значення команд, які повертаються як результат адресу факту (таких як **assert**, **modify** і **duplicate**) або шляхом зв'язування змінної з адресою факту в лівій частині правила (див. далі).

Ідентифікатор факту – це короткий запис для відображення факту на екрані. Вона складається із символу **`f`** та записаного через тире індексу факту. Наприклад, запис **`f-10`** служить позначення факту з індексом **`10`**.

Існує два формати подання фактів: **позиційні (упорядковані)** та **шаблонні (невпорядковані) факти**.

Формати подання фактів

Упорядковані факти складаються з поля, що обов'язково є даними типу **symbol** і наступної за ним, можливо порожньої, послідовності полів, розділених пробілами. Обмеженням факту є круглі дужки. Наведемо приклади запису позиційних фактів:

```
(діагноз ангіна)
(діагноз_пацієнта "бронхіальна астма")
(список_товарів хліб молоко ковбаса пиво чіпси)
(поет "Олександр Сергійович Подерв'янський").
```

Шаблонні факти дозволяють абстрагуватися від структури факту, задаючи імена кожному полю факту. Спочатку потрібно визначити шаблон. Шаблон факту задається конструктором **deftemplate** (схожий на **struct** в Сі або на **record** в Паскалі).

Загальний вигляд шаблону.

```
(deftemplate <ім'я шаблону>
[необов'язковий коментар]
     (slot <ім'я слота>)
     (slot <ім'я слота>)
.................
)
```

Наведемо приклад шаблону.

```
(deftemplate співробітник
     (slot ім'я)
     (slot вік)
     (slot стаж)
     (slot посада)
)
```

Приклад шаблонного факту:

```
(співробітник
     (ім'я "Іван Попович")
     (вік 29)
     (стаж 5)
     (посада "начальник відділу")
)
```

Значення слота має містити лише одне поле. Тому в наведеному прикладі слоти ім'я та посада мають значення типу **string**.

Можливий запис у слоті та кількох полів. Для цього слот має бути визначений як **multislot (мультислот)**. Наступний приклад показує шаблон із мультислотами

```
(deftemplate співробітник
(multislot ім'я)
(slot вік)
(slot стаж)
(multislot
  посада)
)
```

У цьому випадку наведений вище факт може бути записаний наступним чином:

```
(співробітник
(ім'я Іван Попович)
(вік 29)
(стаж 5)
(посада начальник відділу)
)
```

Порядок проходження слотів у шаблонному факті довільний (іншими словами - слоти невпорядковані).

Опрацювання фактів

Усі факти, відомі системі CLIPS, групуються і зберігаються на основі фактів. Нові факти, як упорядковані, так і невпорядковані (шаблонні), можуть бути додані до бази фактів командою **assert**. Синтаксис команди такий:

```
(assert <факт 1> <факт 2>... <факт n>)
```

Приклади додавання факту до основи фактів.

```
(assert (діагноз ангіна) (діагноз грип))
(assert (співробітник (ім'я "Іван Попович"))
(вік 29)
(стаж 5)
(посада начальник відділу)
)
)
```

За допомогою команди **assert** можна ввести до бази фактів відразу кілька фактів:

```
(assert (діагноз ангіна) (діагноз "бронхіальна астма") (діагноз грип)
)
```

У тексті програми факти можна включати не поодинці, а цілим масивом. Для цього CLIPS є конструктор **deffacts** з наступним синтаксисом

```
(deffacts <ім'я_списку фактів> (<факт 1)
(<факт 2)... (<факт n))
```

Приклад масиву фактів:

```
(deffacts список_діагнозів (діагноз
ангіна) (діагноз "бронхіальна астма")
(діагноз грип) )
```

У чому відмінність конструктора **deffacts** від команди **assert**? Команда **assert** безпосередньо вставляє факти в основу фактів. Конструктор **deffacts** лише визначає список фактів, які автоматично додаватимуться щоразу після виконання команди **reset**, що очищає поточну базу фактів. Вираз **deffacts** можна вводити і в командний рядок інтерпретатора, але краще записати його в текстовий файл за допомогою редактора CLIPS або іншого текстового редактора (наприклад, використовувати стандартну програму Windows «Блокнот»).

Завантажити цей файл надалі можна за допомогою вибору команди **load** у пункті **File** меню або з командного рядка за допомогою команди **load**. Так, наприклад, якщо було створено файл з ім'ям **spisok_facts.clp**, то завантаження в CLIPS виглядає так:

```
CLIPS> (load "spisok_facts.clp").
```

Однак після завантаження файлу факти не передаються відразу до бази фактів CLIPS. Команда **deffacts** просто вказує інтерпретатору, що є масив з ім'ям **spisok_facts.clp**, який містить безліч фактів. Власне завантаження в БФ виконується, як говорилося вище, командою **reset**.

```
CLIPS> (reset)
```

Команда **`reset`** спочатку очищає базу фактів, а потім включає до неї факти з усіх раніше завантажених масивів. Вона також додає основу єдиний системно певний факт: **`f-0(initial-fact)`**. Це робиться за умовчанням, оскільки часто має сенс включити в програму стартове правило **`(start rule)`**, яке може бути зіставлене з цим фактом і дозволить виконати будь-які нестандартні операції, що ініціалізують. Однак включати таке правило в програму чи ні справа програміста.

Можна простежити, як виконується команда **reset**, якщо перед виконанням наведених команд встановити режим стеження за середовищем розробки. Для цього потрібно викликати команду **Watch** з меню **Execution** і встановити прапорець **Facts**.

Після додавання факту до БФ рано чи пізно постане питання, як його звідти видалити. Для видалення фактів з БФ у системі CLIPS передбачено функцію **retract**. Кожним викликом цієї функції можна видалити довільну кількість фактів. Видалення деякого факту може спричинити видалення інших фактів, які логічно пов'язані з видаленим.

Синтаксис команди **retract** має такий вигляд

(retract <визначення-факту1> <визначення-факту2>. . .)

Аргумент **<визначення-факту>** може бути або змінною, пов'язаною з адресою факту за допомогою правила (ця можливість буде описана нижче), 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 аргументом функції retract використовувався символ *****, то з поточної бази знань системи будуть видалені всі факти. Функція **retract** не має значення, що повертається.

Зміну факту в БФ можна виконати, вилучивши старий факт і додавши його модифіковане значення, тобто використовуючи функції **retract** та **assert**. Найпростіший приклад наведено нижче.

Приклад:

```
(clear) ⊣
(assert (температура низька)) ⊣
(retract 0) ⊣
(assert (температура висока)) ⊣
```

Для зміни упорядкованих фактів доступний лише цей спосіб. Для спрощення операції зміни невпорядкованих (шаблонних) фактів CLIPS надає функцію **modify**, що дозволяє змінювати значення слотів таких фактів.

(modify <адреса факту> <ім'я слота><нове значення слота>)

Функція **modify** полегшує процес зміни факту, але її внутрішня реалізація еквівалентна викликам пар функцій **retract** і **assert**. За один виклик **modify** дозволяє змінювати лише один факт. У разі успішного виконання функція повертає нову адресу модифікованого факту. Якщо в процесі виконання сталася якась помилка, то користувачеві виводиться відповідне попередження та функція повертає значення **FALSE.**

Наведемо приклад використання функції **modify**.

```
(deftemplate температура (slot значення))
(assert (температура (значення низька)));
адреса факту f-0
(modify 0 (значення висока))
```

Зазначимо, що змінений факт перебуватиме у БФ вже під новою адресою. Внаслідок цього при використанні функції **modify** можуть виникнути деякі проблеми, одну з яких ми розглянено пізніше.

Архітектура правил

Загалом визначення правила має наступний формат

```
(defrule <ім'я правила> ["коментар"]
(CE_1) (CE_2) ... (CE_n); LHS
=>
(< дію_1>) (< дію_2>) ...(< дію_m>) ; RHS
)
```

Ліва частина правила задається набором умовних елементів `CE_1,...CE_n` (`CE` — Conditional Element), який зазвичай складається з умов, застосованих до деяких зразків. У найпростішому випадку умовний елемент це сам зразок (шаблон).

Заданий набір зразків (шаблонів) використовується системою для порівняння з існуючими фактами. Якщо в лівій частині правила не вказано жодного умовного елемента, CLIPS автоматично підставляє умову-зразок як спеціальний факт (**initial-fact**). Все правило має бути укладено у круглі дужки; крім того, у круглі дужки необхідно укласти кожен із компонентів **<умовний елемент>** та **<дія>**.

Всі умови в лівій частині правила об'єднуються за допомогою неявний логічний оператор **AND**. Права частина правила (**RHS**) містить перелік дій, що виконуються під час активізації правила механізмом логічного висновку. Для поділу правої та лівої частини правил використовується стрілка **=>**. Правило немає обмежень на кількість умовних елементів чи дій.

Єдиним обмеженням є вільна пам'ять комп'ютера. Дії правила виконуються послідовно, але і тоді, коли всі умовні елементи у лівій частині цього правила задоволені.

Правила можуть бути введені в систему CLIPS в інтерактивному режимі (тобто безпосередньо з клавіатури у діалоговому вікні) або завантажені з файлу, який створюється текстовим редактором. Останній спосіб набагато зручніший.

Як приклад збудуємо дуже просту ЕС поточного контролю та реагування на кілька можливих надзвичайних ситуацій (НС), які можуть статися з деяким складним агрегатом. Однією з таких НС може стати пожежа, а іншою — відключення енергоживлення. Нижче наведено псевдокод одного з можливих правил у такій ЕС:

НС – пожежа ^ необхідна реакція – активізувати спринклери1, натиснувши синю кнопку.

Перш ніж перетворювати цей псевдокод в правило для CLIPS, необхідно визначити конструктор **deftemplate** для фактів такого типу, що згадуються в цьому правилі.

Надзвичайна ситуація може бути представлена за допомогою наступної конструкції

```
(deftemplate НС (slot тип))
```

У цій конструкції поле тип факту НС має містити такі дані типу **symbol**, як пожежа, повінь та припинення подачі електроенергії. Аналогічно рада ЕС, т. е. необхідна реакція, може бути представлений конструкцією

```
(deftemplate реакція (slot що_робити))
```

У цій конструкції поле **що_робити** факту реакція вказує на те, яка відповідь має бути вироблена системою.

Правило, що описує реакцію на пожежу, надаватиметься, наприклад, так:

```
(defrule П1
(НС (тип пожежа))
=>
(assert (реакція (що-робити включити_спринклери)))
)
```

У правилі мітка **П1** це ім'я правила, зразок **(НС (тип пожежа))** це умовний елемент для зіставлення з фактом з БФ (порівняння тут полягає в перевірці збігу з фактом з БФ).

Продовження побудови нашої експертної системи буде наведено нижче.

Алгоритм виконання правил (логічний висновок у CLIPS)

Як було сказано, CLIPS використовує як логічний висновок прямий ланцюжок міркувань. Після того, як до системи додані всі необхідні правила та підготовлені початкові списки фактів та об'єктів, CLIPS готовий виконувати правила. У традиційних мовах програмування точка входу, точка зупинки та послідовність обчислень явно визначаються програмістом.

У CLIPS потік виконання програми не вимагає чіткого визначення. Знання (правила) та дані (факти) розділені, і механізм логічного висновку, що надається CLIPS, застосовує дані до знань, формуючи список застосовних правил, після чого послідовно виконує їх. Цей процес називається основним циклом виконання правил. Розглянемо послідовність дій (кроків), що виконуються системою CLIPS у цьому циклі на момент виконання нашої програми.

1. Відразу ж під час введення правил як інтерпретатора (з клавіатури) чи завантаження файлу з правилами (команда **load**) CLIPS робить спроби порівняти умовні елементи в лівій частині правила з фактами в БФ. Якщо всі (!) умовні елементи в **LHS** правила узгоджуються з фактами, правило виконується не відразу - спочатку воно активується і поміщається в так званий робочий список правил (РСП), який іноді називають планом вирішення завдання [1]. У РСП може бути від нуля і більше правил.

2. Активізація правил ще означає їх виконання. Логічний висновок у програмі CLIPS починається за командою **`run`**. Під час виконання програми CLIPS із робочого списку правил відбувається запуск першого правила. Запуск правила полягає у виконанні всіх дій, описаних у правій частині правила. Внаслідок цих дій можуть бути активовані нові правила. Активовані правила (тобто правила, умови яких задовольняються на даний момент) містяться в РСП. Розміщення в РСП визначається пріоритетом правила (**`salience`**) та поточною стратегією вирішення конфліктів (ці поняття будуть описані нижче). Деякі правила внаслідок виконання дій можуть бути деактивовані. І тут вони видаляються з РСП.

Список правил, що знаходяться в робочому списку правил, можале вивести на екран монітора командою **agenda**; синтаксис цієї команди такий: **(Agenda) .**

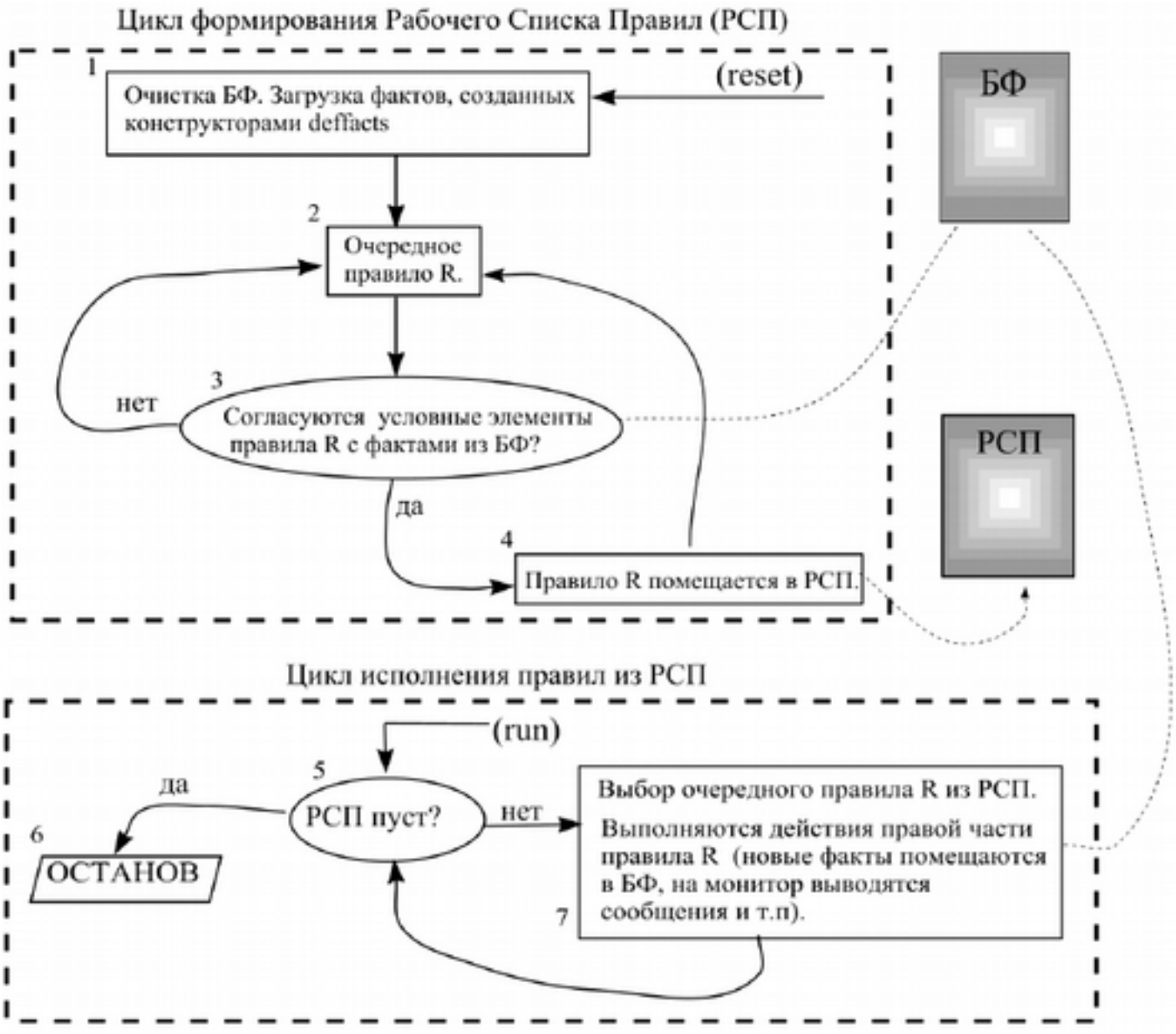
Таким чином, після команди **run** в логічному модулі висновку CLIPS фактично «прокручуються» два цикли (див. рис. 11.3). У першому циклі відбувається аналіз завантажених правил з метою визначення правил, ліві частини яких узгоджуються з фактами БФ; такі правила активуються та містяться в РСП. У другому циклі, який ініціюється командою **run**, відбувається послідовне виконання правил РСП.

Тепер виникає цікаве питання: що станеться, якщо знову буде викликано команду **`run`**? Напрошується відповідь: оскільки є правило і є факт, який задовольняє це правило, то має знову відбутися запуск правила. Але якщо зараз буде спроба виконати команду **`run`**, це не призведе до отримання будь-яких результатів. Перевірка робочого списку правил покаже, що запуск правил не відбувся, оскільки у списку правил були відсутні правила.

Запуск правила не відбувається знову тому, що саме так спроектовано систему CLIPS. Правила у системі CLIPS демонструють властивість, яка називається релаксацією; під цим мається на увазі, що запуск правил стосовно якоїсь конкретної множини фактів не відбувається більше одного разу.

А якби властивість релаксації не було передбачено, експертні системи завжди потрапляли б у тривіальні цикли. Під цим мається на увазі, що після запуску якогось правила цей запуск відбувався б стосовно одного і того ж факту знову і знову, тоді як у реальному світі стимул, що викликав запуск рефлексу, зникає. Наприклад, вогонь був би нарешті загашений за допомогою протипожежної системи або пожежа, знищивши все, закінчився б сам.

Рис. 3. Цикл обробки правил



Локальні та глобальні змінні

Один із загальноприйнятих способів використання змінних полягає в тому, що зі змінною в лівій частині правила зв'язується значення, що отримується з деякого факту, а потім це значення застосовується у правій частині правила. Локальна змінна має вигляд **?<ім'я змінної>**, наприклад: **?ім'я_клієнта**, **?price**, **?діагноз**, **?x1_N**. Між знаком питання та ім'ям символічного поля не повинно бути пробілів. Як буде описано пізніше, сам знак питання має власне призначення.

Змінні використовуються у лівій частині правила для зберігання значень слотів, які надалі можуть порівнюватися з іншими значеннями у лівій частині правила або застосовуватись у правій частині правила. Для опису операції надання значення змінної та отриманого при цьому результату використовуються терміни «зв'язувати» і «пов'язаний».

Один із загальноприйнятих способів використання змінних полягає в тому, що зі змінною в лівій частині правила зв'язується значення, а потім це значення застосовується у правій частині правила.

Наведемо приклад використання локальної змінної:

```
; Файл («D:\Clips\blue-eyes_students.clp»)
(deftemplate студент
(slot ім"я)
(slot колір-очей)
(slot колір-волосся)
)
```

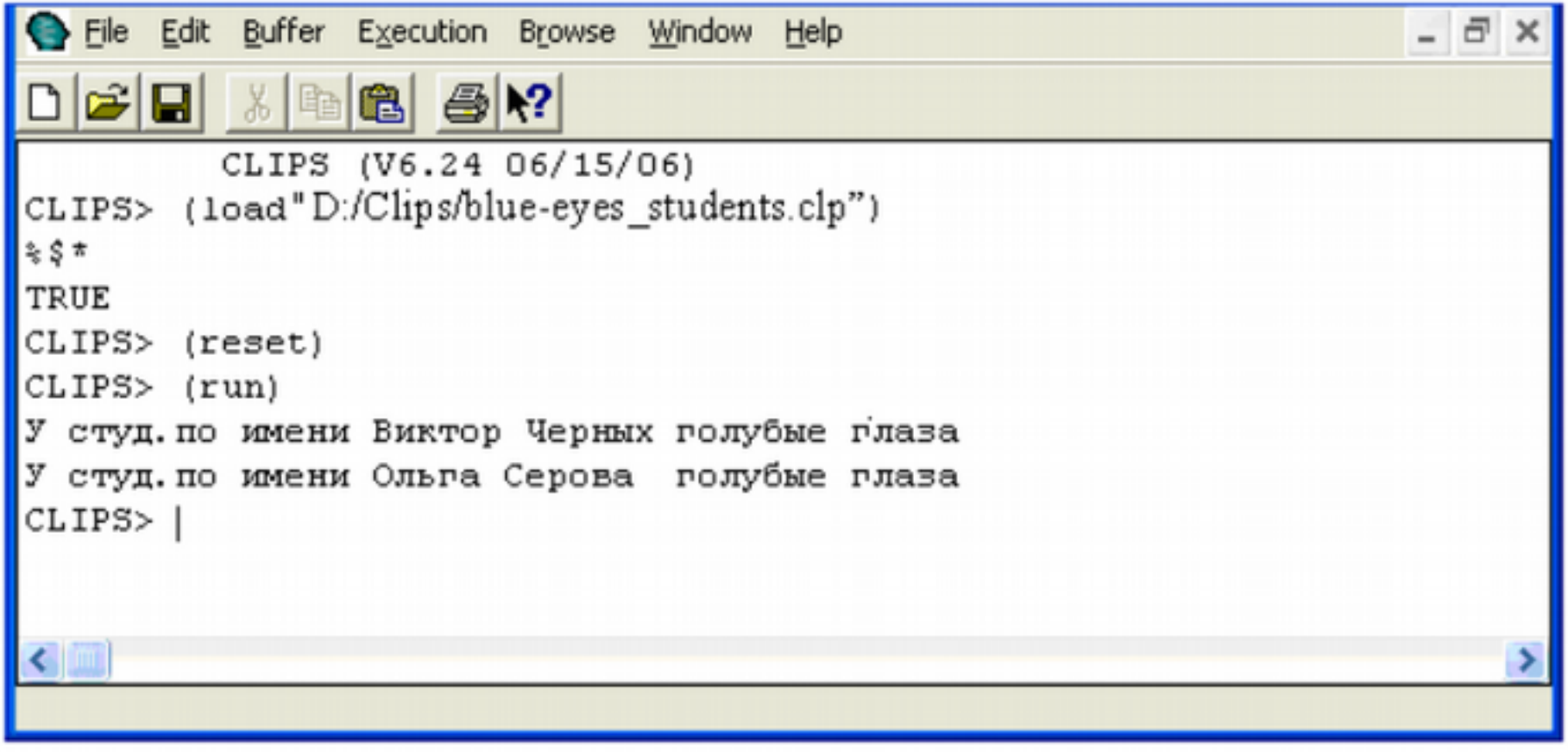
```
(deffacts Група
(студент (ім’я "Іван Попович") (колір-очей сірий)
(колір-волосся світлий))
(студент (ім'я "Ольга Сєрова") (колір-очей блакитний)
(колір-волосся чорний))
(студент (ім'я "Ганна Тюріна") (колір-очей зелений) (колір-волосся руде))
(студент (ім'я "Віктор Чорних") (колір-очей блакитний)
(колір-волосся чорний))
)
```

```
(defrule R
(студент (ім'я ? name) (колір-очей
блакитний))
=>
(printout t "У студ. на ім'я "? name "
блакитні очі" crlf)
)
```

Після введення команд

```
Clips> (load "D:/Clips/blue-
eyes_students.clp") ↵
Clips> (reset) ↵
Clips> (run) ↵
```

Одно-єдине правило **R** спрацьовуватиме кілька разів (за кількістю блакитнооких студентів у БФ), тому що щоразу його умовний елемент (зразок у лівій частині правила) узгоджуватиметься з деяким новим фактом у БФ — черговим блакитнооким студентом (студенткою). Нижче наведено результат роботи програми.



Мал. 11.4. Приклад багаторазового спрацювання того самого правила

Локальна змінна діє не більше одного правила.

Якщо ми хочемо використовувати ту саму змінну в різних правилах, то вона має бути оголошена як глобальна. Глобальна змінна створюється конструктором **defglobal** відповідно до наступного синтаксису:

```
defglobal ?*<ім'я змінної>* = <значення за замовчуванням>
```

Приклади:

```
defglobal ?*ім'я* = “ “,
defglobal? *рівень_небезпеки* = "низький",
```

Функція bind

Часто виникає необхідність зберегти деяке значення у тимчасовій змінній, щоб уникнути повторного обчислення. Це особливо важливо, якщо використовуються функції, які виробляють побічні ефекти. Для зв'язування значення Часто виникає необхідність зберегти деяке значення у тимчасовій змінній, щоб уникнути повторного обчислення. Це особливо важливо, якщо використовуються функції, які виробляють побічні ефекти. Для зв'язування значення змінної із значенням деякого виразу може застосовуватися функція **bind**. Функція **bind** має наступний синтаксис:

```
(bind <ім'я змінної> <значення>)
```

Приклади:

```
(bind ?рівень високий)
(bind? c (+ (*? a? a) (*? b? b)))
```

Значення зв'язування зі змінною може бути отримано безпосередньо від користувача в режимі діалогу. Для цього треба використовувати команду **read**:

```
(bind
  ?ім'я
  (read)).
```

Багатозначні змінні

Працюючи з фактами, шаблони яких містять мультислоти, може виникнути ситуація, яку пояснимо наступному прикладі. Нехай у БФ є безліч фактів, створених за шаблоном «співробітник» (див. приклад раніше).

Нам необхідно визначити посаду співробітника на ім'я Іван Попович. Припустимо, що у БФ є факт з такою інформацією про цього співробітника:

```
співробітник (ім'я Іван Попович) (вік 36)
(стаж 8) (посада начальник відділу)
```

(Зверніть увагу, що ім'я та посада — мультислоти).

Створимо таке правило для знаходження значення мультислота «посада»:

```
(defrule get_status
(співробітник (ім'я Іван Попович) (посада status))
=>
(printout t "посада Івана Поповича -" status crlf)
)
```

Вірне з точки зору синтаксису правило **get_status** не буде активоване вказаним вище фактом! Справа в тому, що звичайні (однозначні) змінні, що позначаються префіксом питання (**?**), узгоджуються точно з одним полем. У нашому випадку мультислот «посада» містить два поля типу **symbol**, і тому спроба узгодити звичайну змінну **status** призведе до невдачі.

Для вирішення проблеми, що виникла, необхідно скористатися багатозначною змінною, яка може узгоджуватися одразу з декількома полями. Багатозначна змінна має префікс знак долара, за яким слідує знак питання (**$?**), наприклад, **$?status**. Враховуючи це, перепишемо правило **get_status** таким чином:

```
(defrule get_status1
(співробітник (ім'я Іван Попович) (посада
$?status))
=>
(printout t "посада Іван Попович - "$?
status crlf)
)
```

Тепер це правило буде активовано при введенні та його виконання після команди run призведе до появи на екрані монітора тексту

`посада Івана Поповича - (начальник відділу)`

Зверніть увагу на те, що при виведенні на зовнішній пристрій багатозначне значення, пов'язане зі змінною **`$status`**, полягає в круглі дужки.

Література до лекції 11

1. Частиков, А. П. Разработка экспертных систем. Среда CLIPS / А. П. Частиков, Д. Л. Белов, Т. А. Гаврилова. — СПб. : БХВ-Петербург, 2003.
2. Джарратано, Дж. Экспертные системы : принципы разработки и программирование / Дж. Джарратано, Г. Райли. — М. : И. Д. Вильямс, 2007.
3. Джексон, П. Введение в экспертные системы / П. Джексон. — М. : Вильямс, 2001.

Наступна лекція буде присвячена прикладам створення баз знань в продукційній системі CLIPS.