МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

**ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
МІСЬКОГО ГОСПОДАРСТВА імені О. М. БЕКЕТОВА**

**М. В. Новожилова,
О. О. Петрова**

РОЗРОБКА ЕКСПЕРТНИХ СИСТЕМ В СЕРЕДОВИЩІ CLIPS

НАВЧАЛЬНИЙ ПОСІБНИК

**Харків
ХНУМГ ім. О. М. Бекетова
2019**

УДК 004.891
Н74

Автори:

Новожилова Марина Володимирівна, доктор фізико-математичних наук, професор, завідувач кафедри прикладної математики та інформаційних технологій Харківського національного університету міського господарства імені О. М. Бекетова;

Петрова Олена Олександрівна, кандидат технічних наук, доцент кафедри прикладної математики та інформаційних технологій Харківського національного університету міського господарства імені О. М. Бекетова

Рецензенти:

Яковлев Сергій Всеволодович, доктор фізико-математичних наук, професор кафедри інформатики Харківського національного аерокосмічного університету ім. М. Є. Жуковського «Харківський авіаційний інститут»;

Сізова Наталя Дмитрівна, доктор фізико-математичних наук, професор кафедри комп'ютерних наук та інформаційних технологій Харківського національного університету будівництва та архітектури

Рекомендовано до друку Вченою радою ХНУМГ ім. О. М. Бекетова, протокол № 6 від 28.12.2018.

Новожилова М. В.

Н74 Розробка експертних систем в середовищі CLIPS : навч. посібник / М. В. Новожилова, О. О. Петрова ; Харків. нац. ун-т міськ. госп-ва ім. О. М. Бекетова. – Харків : ХНУМГ ім. О. М. Бекетова, 2019. – 130 с.

У навчальному посібнику подано можливості та особливості CLIPS як інструменту створення експертних систем, наведено інформацію щодо використання вбудованих функцій та основних конструкцій мови. Для закріплення знань пропонуються детальні пояснення щодо використання основних елементів мови, описи функцій, конструкторів та команд, які пов'язані з правилами; внутрішні алгоритми подання та оброблення правил, приклади розроблення експертних систем з використанням евристичного, процедурного та об'єктно-орієнтованого програмування.

Призначено для студентів спеціальностей 122 – Комп'ютерні науки, 126 – Інформаційні системи та технології та 151 – Автоматизація та комп'ютерно-інтегровані технології, а також усіх тих, хто цікавиться цими питаннями.

УДК 004.891

ЗМІСТ

ВСТУП

Комерційне впровадження експертних систем (ЕС) почалося в 80-х роках XX ст. Наразі ЕС широко застосовуються в науці, техніці, бізнесі, медицині, сільському господарстві, на виробництві та в інших галузях людської діяльності [1]. Професор Едвард Фейгенбаум зі Стенфордського університету, один із перших дослідників технології ЕС, визначив поняття ЕС як «… інтелектуальної комп'ютерної програми, у якій використовуються знання та процедури логічного виводу для вирішення завдань, достатньо складних для того, щоб вимагати для свого вирішення значного об'єму експертних знань людини» [2]. Отже, ЕС – це комп'ютерна система, яка емулює здатності до прийняття рішень, а термін «емулює» означає, що ЕС повинна діяти в усіх відношеннях як людина-експерт.

У наш час терміни «система, яка заснована на знаннях», та «експертна система» використовуються як синоніми. Фактично ЕС стали розглядатися як модель програмування або підхід до програмування, альтернативний відносно звичайного алгоритмічного програмування. Процес створення ЕС відрізняється від процесу розроблення звичайних програм, тому що в ЕС розглядаються завдання, які не мають задовільненого алгоритмічного вирішення, тому для досягнення прийнятного рішення використовуються логічні виводи.

За своєю структурою ЕС підрозділяється на два основних компоненти: базу знань та машину логічного виводу [1]. База знань вміщує знання, на підставі яких машина логічного виводу формує висновки, що становлять відповіді ЕС на запити користувача, який бажає отримати експертні знання. Машина логічного виводу – це механізм міркувань, який оперує знаннями та даними з метою отримання нових даних щодо знань, які є наявними в робочій пам'яті. Для цього зазвичай використовується програмно реалізований механізм дедуктивного логічного виводу (будь-який його різновид) або механізм пошуку рішення в мережі фреймів або семантичній мережі.

Як знання в ЕС можна використовувати або експертні знання у визначеній предметній області, або загальнодоступні знання. Технологія ЕС може містити спеціальні мови ЕС, а також програмні та апаратні засоби, які призначені для забезпечення розроблення та експлуатації ЕС.

Систему CLIPS (C Language Integrated Production System (мова С, інтегрована з продукційними системами) розпочав розробляти в космічному центрі NASA Гері Райлі (Gary Riley) у 1984 році, яка наразі є найбільш використовуваним інструментальним засобом створення експертних систем (A tool for building expert systems) завдяки своїй швидкості, ефективності та безкоштовності [3].

CLIPS складається з інтерактивного середовища – експертної оболонки зі своїм способом представлення знань, гнучкої і потужної об'єктно-орієнтованої мови COOL (CLIPS Object-Oriented Language) і допоміжних додаткових інструментів [4–5].

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

1) уведенням відповідних команд і конструкторів мови безпосередньо в середовище CLIPS;

2) використанням інтерактивного віконного інтерфейсу CLIPS;

3) за допомогою програм-оболонок, що реалізують свій інтерфейс спілкування з користувачем і використовують механізми подання знань і логічного виводу CLIPS [6].

У розробленому навчальному посібнику як основний метод спілкування з CLIPS використовується застосування командного рядка. Після появи в головному вікні CLIPS запрошення `CLIPS>` користувачу необхідно вводити команди в середовище безпосередньо з клавіатури. Команди можуть бути викликами системних або користувацьких функцій, конструкторами різних даних CLIPS тощо.

У разі виклику користувачем деякої функції, вона відразу виконується, а результат її роботи відображається користувачу. Для виклику функцій або операцій CLIPS використовує префіксну нотацію – аргументи завжди йдуть після імені функції або операції.

Після виклику конструкторів CLIPS створює новий об’єкт відповідного типу, який представляє деякі знання в системі.

Після введення в середовище імені створеної раніше глобальної змінної CLIPS відобразить її поточне значення, а введення в середовище константи спричинить її відображення в головному вікні CLIPS.

Для отримання практичних навичок роботи з CLIPS студентам пропонується виконати практичні заняття з використанням базових конструкцій CLIPS, розробити ЕС із використанням евристичної парадигми програмування та розробити експертні системи із застосуванням об’єктно-орієнтованого розширення CLIPS під назвою COOL.

Інструментарій мови експертних систем CLIPS можна використовувати в галузі комп’ютерних наук, інформаційних управлінських систем, у програмотехніці, для розроблення інформаційних систем з елементами штучного інтелекту.

1 ЗАГАЛЬНІ ПОЛОЖЕННЯ

Система CLIPS реалізована у вигляді файлу, що виконується, *Clipswin.exe*, призначеного для роботи в Windows. Після запуску файлу на екрані з'являється діалогове вікно з такими меню: *File*, *Edit*, *Execution*, *Browse*, *Window*, *Help*.

Для створення нового файлу доцільно використовувати вбудований редактор із подальшим завантаженням у базу знань змісту цього файлу командою *Load* меню *File*. Після завантаження файлу подається команда *Reset*, яка очищує список активованих правил. Список фактів після виконання команди *Reset* повинен вміщувати один факт `initial-fact`. Для запуску програми використовується команда *Run*, після виконання якої CLIPS починає виконувати всі правила, які знаходяться в списку активованих правил. Виконання програми зупиняється або коли в списку правил більше не залишається жодного активного правила, або після переривання, яке виконується користувачем комбінацією клавіш Ctrl+C. Якщо після завершення програми необхідно очистити базу знань CLIPS, а також прибрати всі факти зі списку фактів, тобто відновити початковий стан CLIPS, то необхідно виконати команду *Clear*, яка очищає CLIPS від усіх правил та фактів на відміну від команди *Reset*, яка знищує тільки всі факти зі списку фактів.

CLIPS є продукційною системою, реалізацією виведення якої є алгоритм Rete, від латинського слова rete, що означає мережа. Алгоритм Rete функціонує як мережа, яка призначена для зберігання великого обсягу інформації та забезпечує значне скорочення часу відклику, збільшення швидкодії під час запуску правил порівняно з великими групами правил IF–THEN, що повинні перевірятися один за одним у звичайній системі, заснованій на правилах. Цей алгоритм засновано на використанні динамічної структури даних, яка автоматично реорганізується з метою оптимізації пошуку [7].

Алгоритм Rete широко використовується для реалізації зіставлення зі зразком у системах із циклом: зіставлення – рішення – дія (match – resolve – act), для генерації і логічного виводу. Під час використання алгоритму Rete будується спеціальний граф або префіксне дерево, вузлам якого відповідають частини умов правил. Шлях від кореня до листа утворює повну умову деякої продукції. У процесі роботи кожен вузол зберігає список фактів, відповідних умові. У разі додавання або модифікації факту останній проганяється по мережі, і відзначаються вузли, умовам яких цей факт відповідає. У разі виконання повної умови правила та за досягнення системою листа графа правило виконується.

Алгоритм Rete призначено для підвищення швидкодії системи з прямим логічним виводом, яка заснована на правилах, завдяки обмеженню обсягу роботи, що необхідна для повторного обчислення конфліктної множини після запуску одного з правил.

Мова CLIPS спроектована на використання прямого логічного виводу, а як базовий синтаксис для визначення конструкцій мови використовується стандартна БНФ-нотація (форма Бекуса – Наура, Backus – Naur Form, BNF).

БНФ-нотація – це спосіб запису правил контекстно-вільної граматики, тобто це форма опису формальної мови. БНФ визначає скінченну кількість символів (нетерміналів) та правила заміни символу на деяку послідовність букв (терміналів) і символів. Процес отримання ланцюжка літер можна визначити поетапно:

– спочатку є один символ (символи зазвичай знаходяться у кутових дужках);

– цей символ замінюється на деяку послідовність літер і символів, відповідно до одного з правил;

– процес повторюється: на кожному кроці один із символів замінюється на послідовність згідно з правилом. Зрештою, виходить ланцюжок, що складається з літер і не містить символів. Це означає, що отриманий ланцюжок може бути виведений з початкового символу.

Основні вимоги, які використовуються в CLIPS для побудови команд та визначень:

– усі команди мають бути укладені в круглі дужки і розділені символом переходу на новий рядок;

– команди користувач вводить із клавіатури після появи в головному вікні CLIPS запрошення: `CLIPS>`;

– аргументи завжди йдуть після імені функції або операції, наприклад, функція суми двох чисел має вигляд: `(+ 3 4)`;

– слово або вираз, укладене в кутові дужки, називається *нетермінальним символом* (наприклад, `<string>`) і вимагає подальшого визначення. Якщо за нетермінальним символом йде символ «*», то в цьому місці може знаходитися список із нуля або більше елементів цього типу. Якщо за нетермінальним символом йде символ «+», то в цьому місці може знаходитися список із одного або більше елементів цього типу. Символи «*» та «+», які зустрічаються у формулах, є термінальними символами;

– слово або вираз, що не укладені в кутові дужки, називаються *термінальними символами* і представляють синтаксис конструкції мови CLIPS, яка описується;

– елементи, укладені в квадратні дужки [наприклад, [<коментарі>]), є необов’язковими елементами;

– символ «|» (вертикальна риска), який поділяє два або більше елементів визначення, вказує на те, що в конструкції необхідно використовувати один з перерахованих елементів;

– лексема «`::=`» використовується для позначення необхідності заміни деякого нетермінального символу. Наприклад, визначення:

```
<lexeme> :: = <symbol> | <string>
```

означає, що нетермінальний символ `<lexeme>`, який зустрічається в деякому визначенні, має бути замінений або на символ `<symbol>`, або на символ `<string>`.

2 ПРОЦЕДУРНЕ ПРОГРАМУВАННЯ

CLIPS – це мова програмування, яка дозволяє використовувати низку підходів, що забезпечує підтримку програмування на підставі правил, об'єктно-орієнтованого та процедурного програмування [1, 6].

Процедурний механізм дозволяє користувачеві за допомогою вбудованих у мову функцій розробляти або конструювати нові функції, які виконують деякі дії, або повертають деякі значення. Для подання знань у процедурній парадигмі CLIPS надає такі механізми, як *функції, глобальні змінні та родові функції*.

2.1 Примітивні типи даних

CLIPS підтримує такі типи даних: `integer`, `float`, `symbol`, `string`, `external-address`, `fact-address`, `instance-name`, `instance-address`.

1. Для зберігання числової інформації призначені типи `integer` та `float`.

Представлення в CLIPS цілого числа в БНФ-нотації має вигляд:

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

Предствлення в CLIPS дійсного числа в БНФ-нотації має вигляд:

< дійсне > ::= < ціле > <експонента> |
< ціле > .[експонента] |
< беззнакове−ціле > [експонента] |
< ціле > . < беззнакове−ціле > [експонента]
< беззнакове−ціле >::=<цифра>+
<експонента>::= e | E < ціле >.

2. Для зберігання символьної інформації призначені типи `string` та `symbol`.

Значенням типу `symbol` може бути будь-яка послідовність символів, яка починається з будь-якого некерованого ASCII–символу. Значення типу `symbol` закінчуються обмежувачем, яким виступають пробіли, символи табуляції або переходу на інший рядок, подвійні лапки, круглі дужки, символи: «&», «|», «<», «~», «;» (точка з комою є і символом початку коментаря, і обмежувачем значення типу `symbol`). Значення типу `symbol` не можуть починатися з символу «?» або «$?», але можуть вміщувати ці символи.

Значення типу `integer` та `float` є окремим випадком типу `symbol`.

Значення типу `string` становить рядок символів у подвійних лапках, крім того символ подвійних лапок можна включити в рядок перед символом «\», наприклад, `"a\"quote"`.

3. Значення типу `external-address` становить адресу структури даних, яка повернута зовнішньою функцією.

4. Значення типу `fact-address` становить адресу факту, який є списком атомарних значень примітивних типів, посилатися на які можна або використовуючи порядок визначення цих значень (упорядковані факти), або за ім'ям (неупорядковані факти або шаблони).

5. Значення типу `instance-name` призначено для зберігання значення імені об'єкта. Об'єкт у CLIPS є екземпляром визначеного користувачем класу, для роботи з яким використовується конструктор `defclass`. Для створення об'єкта використовується функція `make-instance`. Посилатися на об'єкт можна або за адресою, або в межах окремого модуля за ім'ям об'єкта. Для представлення імені використовується значення типу `symbol` у квадратних дужках.

6. Значення типу `instance-address` призначено для зберігання значення адреси об'єкта. Значення цього типу можна отримати викликом функції `instance-address`.

2.2 Функції в CLIPS

Функцією в CLIPS називається частина коду, яка має ім'я, повертає результат і не змінює стан бази знань середовища CLIPS.

CLIPS оперує з декількома типами функцій: визначені користувачем зовнішні функції, системні (внутрішні) функції, функції, які визначені в середовищі CLIPS за допомогою конструктора `deffunction`, родові функції.

2.2.1 Конструктор `deffunction`

Для створення нових функцій використовується конструктор `deffunction`, що має такий синтаксис:

```
(deffunction ім'я–функції
      [необов'язкові коментарі]
      (<обов'язкові параметри>)
      [<груповий параметр>]
      (дія_1)
      (дія_2)
      .....
      (дія_N)),
```

де <обов'язкові параметри>**::=**<вираз–просте–поле>,

< груповий параметр>::=<вираз–складове–поле>.

Синтаксис конструктора `deffunction` вміщує такі елементи:

– ім'я функції;

– коментарі;

– список із нуля або більше параметрів (обов'язкові параметри);

– необов'язковий символ групових параметрів для указання того, що функція може мати змінне число аргументів;

– послідовність дій або виразів, які будуть виконані (обчислені) по порядку в момент виклику функції.

Обов'язкові параметри визначають мінімальне число аргументів, яке має бути передано функції під час її виклику. У діях функції можна посилатися на кожен із цих параметрів як на звичайні змінні, що містять прості значення.

Залежно від того, чи задано груповий параметр, функція, яка створена конструктором, може приймати точне число параметрів або число параметрів не менше, ніж деяке задане число.

Якщо був заданий груповий параметр, то функція може приймати будь-яку кількість аргументів, яка дорівнює або перевищує мінімальне число. Якщо груповий параметр не заданий, то функція може приймати число аргументів, яке точно дорівнює кількості обов'язкових параметрів.

У прикладі 2.1 визначено функцію `om(x,y)`, яка повертає цілу частину частки від поділу змінної `y` на змінну `x`:

```
CLIPS> (deffunction om
(?x ?y)
(div ?y ?x))
CLIPS> (om 5 10)
2
CLIPS>
```

Потрібно звернути увагу на те, що в CLIPS ім'я змінної починається з символу «?» і що для виклику функції (у цьому випадку вбудованої функції поділу без залишку `div`) використовується префіксна нотація, а також на те, що вся конструкція є списком.

Усі аргументи функції, які не відповідають обов'язковим параметрам, групуються в одне значення складового поля, посилатися на яке можна, використовуючи символ групового параметра «`$`». Для роботи з груповими параметрами використовуються стандартні функції для роботи зі складеними полями `length` та `nth`. Визначення функції може містити тільки один груповий параметр, як показано в прикладі 2.2:

```
CLIPS>
     (deffunction print-args (?a ?b $?c)
```

```
    (printout t ?a  " "  ?b  "and"  (length ?c) "extras:
"  ?c   crlf))
    CLIPS> (print-args 1 2)
        1 2 and 0 extras: ( )
    CLIPS>(print-args a b c d )
        a b and 2 extras: (c d)

    CLIPS>(print-args a )
    [ARGACCESS4] Function print-args expected at least 2
arguments.
```

У прикладі 2.2 за допомогою конструктора `deffunction` визначається функція `print-args`, яка приймає два обов'язкових параметра `?a` та `?b`, і має груповий параметр `$?c`.

Функція виводить на екран свої обов'язкові параметри, а також число полів у складеному параметрі та його вміст.

Після виклику функції інтерпретатор CLIPS послідовно виконує дії в порядку, заданому конструктором.

Функція повертає значення або обчислений вираз. Якщо остання дія не повернула ніякого результату, то функція, що виконується, також не поверне результат (як у наведеному вище прикладі). Якщо функція не виконує ніяких дій, то поверне значення, яке дорівнює `FALSE`. У разі виникнення помилки при виконанні чергової дії виконання функції буде перервано і повернутим значенням також буде `FALSE`.

Функції можуть бути само- і взаємнорекурсивними.

Саморекурсивна функція викликає сама себе зі списку своїх власних дій.

У прикладах 2.3 та 2.4 наведено реалізацію рекурсії на прикладі обчислення факторіала. Конструктор `deffunction` може викликати в своєму тілі інші конструктори `deffunction`, включаючи самих себе.

Приклад 2.3. Перший варіант програми обчислення факторіала від додатного цілого числа.

```
CLIPS> (deffunction factorial (?a)
(if (or (not(integerp ?a))(< ?a 0)) then
(printout t "Factorial error" crlf )
else
(if (= ?a 0) then 1
else
(* ?a (factorial (- ?a 1))))))
CLIPS> (factorial 4)
24
```

Приклад 2.4. Другий варіант програми обчислення факторіала від додатного цілого числа.

```
CLIPS> (deffunction factorial (?n)
(if (>= ?n 1)
then (* ?n (factorial (- ?n 1)))
else 1))
CLIPS> (factorial 4)
24
CLIPS>
```

Взаємна рекурсія між двома функціями вимагає попереднього оголошення однієї з цих функцій, для чого в CLIPS використовується конструктор `deffunction` з порожнім списком дій.

У прикладі 2.5 розглянуто конструктор `deffunction`, у якому обчислюється довжина гіпотенузи прямокутного трикутника за допомогою теореми Піфагора, де `a` та `b` – сторони, які утворюють прямий кут, а `c` – гіпотенуза:

```
CLIPS> (deffunction hyp (?a ?b)
(** (+ (* ?a ?a) (* ?b ?b)) 0.5))
CLIPS> (hyp 3 4)
5.0
CLIPS>
```

Параметри функції `?a` та `?b` використовуються для передачі у функцію значень довжини двох катетів трикутника, а функція `**` з другим аргументом `0.5` використовується для обчислення квадратного кореня.

У прикладі 2.6 розглянуто конструктор `deffunction`, який визначає чи є задане число простим:

```
(deffunction primep (?num)
 (loop-for-count (?i 2 (- ?num 1))
       (if (= ?num (* (div ?num ?i) ?i))
       then
       (return FALSE)))
 (return TRUE)).
```

Конструктор `deffunction` з ім'ям `primep` повертає значення `TRUE`, якщо параметр `?num` є простим числом, інакше – значення `FALSE`. У наведеному конструкторі використовується функція `loop-for-count` для ітерації по всіх числах від 2 до того числа, що на одиницю менше числа, яке перевіряється як претендент на просте число.

Функція `div` виконує цілочисельне ділення та дає в результаті цілочисельне значення, що повертається. Тому вираз `(div 5 2)` повертає 2, а не значення 2.5, яке було би повернуто у разі використання виразу (/ 5 2). Якщо вираз `(* (div ?num ?i) ?i)` повертає початкове значення `?num`, то число `?num` ділиться без залишку на `?i` і тому число `?num` не є простим.

У прикладі 2.7 функція `foo` попередньо оголошена і отже може бути викликана з функції `bar`. Остаточна реалізація функції `foo`, виконана конструктором після оголошення функції `bar`, така:

```
(deffunction foo ())
(deffunction bar () (foo))
(deffunction foo () (bar)).
```

Приклад 2.8 використання циклічних посилань друг на друга в конструкторах `deffunction`: у конструкторі `deffunction A` передбачено виклик конструктора `deffunction B`, який викликає конструктор `deffunction C`, а останній вміщує виклик конструктора `deffunction A`.

```
CLIPS> (deffunction B (?n))
CLIPS> (deffunction A (?n)
(if (<= ?n 0)
then 1
else (+ 2 (B (- ?n 1)))))
CLIPS> (deffunction C (?n))
CLIPS> (deffunction B (?n)
(if (<= ?n 0)
then 1
else (* 2 (C (- ?n 1)))))
CLIPS> (deffunction C (?n)
(if (<= ?n 0)
then 1
else (- 2 (A (- ?n 1)))))
CLIPS>
CLIPS> (C 7)
4
CLIPS> (A 7)
6
CLIPS> (B 7)
8
CLIPS>
```

Для відстеження роботи конструктора `deffunction` використовується команда `watch`, яка виводить інформаційне повідомлення, як далі показано для конструктора `C`. Позначення `DFN` вказує, що воно належить до конструктора `deffunction`, а символ «>>» показує, що виконується перехід у конструктор, а символ «<<» показує, що виконується вихід із конструктора `deffunction`. Символ `ED` (Evaluation Depth, Глибина вкладеності) показує, як вкладаються один в одного виклики конструктора `deffunction`. Останнім фрагментом інформації, який відображається в рядку, є фактичні параметри, які передаються в конструктор `deffunction`.

```
CLIPS> (watch deffunctions)
CLIPS> (C 7)
DFN >> C ED:1 (7)
DFN >> A ED:2 (6)
DFN >> B ED:3 (5)
DFN >> C ED:4 (4)
DFN >> A ED:5 (3)
DFN >> B ED:6 (2)
DFN >> C ED:7 (1)
DFN >> A ED:8 (0)
DFN << A ED:8 (0)
DFN << C ED:7 (1)
DFN << B ED:6 (2)
DFN << A ED:5 (3)
DFN << C ED:4 (4)
DFN << B ED:3 (5)
DFN << A ED:2 (6)
DFN << C ED:1 (7)
4
CLIPS>
```

2.2.2 Команди та функції для роботи з конструктором `deffunction`

CLIPS надає такі команди та функції для роботи з конструктором `deffunction`.

1. Команда `ppdeffunction` (скорочено від `pretty print deffunction` – структуроване виведення конструктора `deffunction`) виводить визначення заданої функції на екран і має такий синтаксис:

`(ppdeffunction` <ім'я–функції>).

2. Команда `list-deffunction` призначена для відображення в діалоговому вікні списку імен усіх визначених у системі функцій і має такий синтаксис:

`(list-deffunctions)`.

3. Команда `undeffunction` призначена для видалення функцій, визначених користувачем за допомогою конструкторів `deffunction` і має такий синтаксис:

`(undeffunction` <ім'я–функції>).

Як параметр <ім'я–функції> можна використовувати символ «*», під час застосування якого команда спробує видалити всі визначені користувачем

функції. Знищення функції закінчиться невдачею, якщо обрана функція в цей момент використовується або виконується, наприклад правилом.

4. Функція `get–deffunction–list` повертає багатозначне значення, яке вміщує список задекларованих конструкцій `deffunction`, і має такий синтаксис:

(`get–deffunction–list`).

2.2.3 Типи функцій CLIPS

В CLIPS користувач використовує такі функції:

а) ***процедурні функції,*** які реалізують можливості розгалуження, організації циклів у програмах, тобто призначені для керування потоком виконання дій:

1) `if` – оператор розгалуження;

2) `while` – цикл із передумовою;

3) `loop–for–count` – ітеративний цикл;

4) `prong` – об'єднання дій в одній логічній команді;

5) `prong$` – виконання набору дій над кожним елементом поля;

6) `return` – переривання функції, циклу, правила тощо;

7) `break` – завершення виконання функції `while`, `loop–for–count` або `prong$`, у якій вона безпосередньо використовується, для виклику передчасного завершення циклу у разі виявлення деякої умови, тобто дія як у `return`, але без повернення;

8) `switch` – оператор множинного розгалуження;

9) `bind` – створення та зв'язування змінних;

10) `halt` – зупинка виконання правил, які знаходяться в робочому списку правил;

б) ***логічні функції***, які повертають значення `TRUE` або `FALSE`:

1) функції булевої логіки: `and`, `or`, `not`;

2) функції порівняння чисел: $=$, $\neq$, $>$, $\geq$, $<$, $\leq$, `:=` (перевірка рівності першого та останнього елементів);

3) предикатні функції, призначення яких полягає в тестуванні свого єдиного аргументу на приналежність до того чи іншого типу:

– функції `integerp`, `floatp`, `numberp` перевіряють, чи належить аргумент до типу `integer` або `float`;

– функції `stringp`, `symbolp`, `lexemerp` перевіряють, чи належить аргумент до типу `string` або `symbol`;

– функція `pointerp` перевіряє, чи належить аргумент до типу `external–address`;

– функція `evenp` перевіряє ціле число на парність;

– функція `oddp` перевіряє ціле число на непарність;

– функція `multifildp` перевіряє, чи є аргумент складовим полем;

4) функції порівняння за типом та значенням:

– функція `eq` повертає значення `TRUE`, якщо її перший аргумент дорівнює другому і всім наступним аргументам (якщо вони присутні);

– функція `neq`, навпроти, повертає значення `FALSE`, якщо її перший аргумент дорівнює другому і наступним аргументам;

в) ***математичні функції:***

1) стандартні функції:

– `max`, `min`, `+`, `-`, `*`, `/`;

– `div` – цілочислене ділення;

– `abs` – абсолютне значення;

– `float` – перетворення в тип `float`;

– `integer` – перетворення в тип `integer`;

2) розширені функції:

– `sqrt` – добування кореня;

– `round` – округлення числа;

– `mod` – обчислення залишку від ділення;

3) тригонометричні функції: `sin`, `sinh`, `cos`, `cosh`, `tan`, `tanh`, `acos`, `acosh`, `acot`, `acoth`, `acsc`, `acsch`, `asec`, `asech`, `asin`, `asinh`, `atan`, `atanh`, `cot`, `coth`, `csc`, `csch`, `sec`, `sech`, `deg–grad` (перетворення з градусів у радіани), `rad-deg` (перетворення з радіанів у градуси);

4) логарифмічні функції: `log`, `log10`, `exp`, `pi`;

г) ***функції роботи з рядками:***

1) функція `str-cat` призначена для об'єднання рядків у значення типу `string`. Якщо використати таку функцію (`str-cat "foo" bar`), то відповідь буде такою: `"foobar"`;

2) функція `sym-cat` призначена для об'єднання рядків у значення типу `symbol`. Якщо використати таку функцію (`sym -cat "foo" bar`), то відповідь буде такою: `foobar`;

3) функція `sub-string` призначена для виділення підрядків;

4) функція `str-index` призначена для пошуку підрядка;

5) функція `eval` призначена для виконання рядка в CLIPS;

6) функція `build` призначена для виконання рядка як конструктора CLIPS;

7) функція `upcase` використовується для перетворення символів у символи верхнього регистру;

8) функція `lowcase` використовується для перетворення символів у символи нижнього регістру;

9) функція `str-compare` використовується для порівняння рядків;

10) функція `str-length` використовується для визначення довжини рядка;

11) функція `check-syntax` використовується для перевірки синтаксису рядка;

12) функція `string-to-field` використовується для повернення першого поля рядка;

д) ***функції роботи зі складовими величинами:***

1) функція `create$` використовується для створення складової величини. Наприклад, функція `(create$ (+ 3 4) (* 2 3) (/ 8 4))` повертає значення, яке є складовою величиною `(7 6 2.0)`;

2) функція `nth$` використовується для отримання елемента складової величини. Перший аргумент цієї функції має бути цілим числом, більшим або дорівнювати 1, який визначає індекс поля в складовій величині, що задана другим аргументом. Наприклад, функція `(nth$ 3 (create$ a b c d f))` повертає як відповідь значення `c`;

3) функція `member$` виконує пошук елемента складової величини та повертає індекс поля, якщо воно вміщується в складовій величині. Наприклад, функція `(member$ blue (create$ red 3 "text" 7.8 blue))` повертає відповідь 5;

4) функція `subset$` виконує перевірку, чи не є одна складова величина підмножиною іншої;

5) функція `delete$` виконує знищення елемента складової величини;

6) функція `explode$` призначена для створення складової величини з рядка;

7) функція `implode$` призначена для створення рядка зі складової величини;

8) функція `subseq$` призначена для вилучення підпослідовності зі складової величини;

9) функція `replace$` призначена для заміни елемента складової величини;

10) функція `insert$` призначена для додавання нових елементів у складову величину;

11) функція `first$` призначена для отримання першого елемента складової величини;

12) функція `rest$` призначена для отримання залишку складової величини;

13) функція `length$` призначена для визначення числа елементів складової величини;

14) функція `delete-member$` призначена для знищення елементів складової величини;

15) функція `replace-member$` призначена для заміни елементів складової величини;

е) ***команди введення–виведення*** використовують такі логічні імена пристроїв:

1) `stdin` – пристрій введення;
2) `stdout` – пристрій виведення;
3) `wclips` – пристрій, який використовується як довідковий;
4) `wdialog` – пристрій для відправлення користувачеві повідомлень;
5) `wdisplay` – пристрій для відображення правил, фактів та інше;
6) `werror` – пристрій виведення повідомлень про помилки;
7) `wwarning` – пристрій для виведення попереджень;
8) `wtrase` – пристрій для виведення налагоджувальної інформації;

ж) ***команди введення–виведення***:

1) команда `open` відкриває файл;
2) команда `close` закриває файл;
3) команда `printout` виконує виведення інформації на заданий пристрій;
4) команда `read` зчитує дані з заданого пристрою;
5) команда `readline` виконує введення рядка із заданого пристрою;
6) команда `format` виконує форматоване виведення на заданий пристрій;
7) команда `rename` перейменовує файл;
8) команда `remove` знищує файл;

и) ***основні команди для роботи з системою CLIPS (***близько двох десятків команд):

1) команда `load` завантажує конструктори із текстового файлу;
2) команда `load+` завантажує конструктори із текстового файлу без відображення;
3) команда `reset` очищає поточний список фактів із додаванням у список фактів факту `initial-fact`;
4) команда `clear` очищає поточного списку фактів;
5) команда `run` виконує завантажені конструктори;
6) команда `save` зберігає розроблені конструктори у текстовий файл;
7) команда `set-break` дозволяє зупиняти виконання програми перед тим, як відбудеться запуск будь-якого правила з вказаної групи правил;
8) команда `exit` здійснює вихід з середовища CLIPS.

2.3 Глобальні змінні

Глобальні змінні можна використовувати як для передачі інформації, яка використовується в правій частині правила та повинна активізувати зіставлення з шаблоном, так і як константи в правилах.

2.3.1 Конструктор `defglobal`

Для подання даних CLIPS використовує як глобальні змінні (`globals`), так і факти, а також об'єкти. На відміну від змінних, пов'язаних зі своїм значенням у лівій частині правила, глобальна змінна доступна всюди після свого створення (а не тільки в правилі, у якому вона отримала своє значення). Однак, на відміну від змінних більшості процедурних мов програмування, глобальні змінні в CLIPS слабо типізовані. Фактично змінна може приймати значення будь-якого примітивного типу CLIPS під час кожного нового присвоєння значення.

Імена глобальних змінних розпочинаються та закінчуються символом «*», наприклад, `?*x*`.

За допомогою конструктора `defglobal` у середовищі CLIPS можна оголосити глобальні змінні та присвоїти їм початкові значення у такий спосіб:

(`defglobal` [<ім'я–модуля>] <визначення–змінної>*)
< визначення–змінної > `::=` < ім'я–змінної> = <вираз>
< ім'я–змінної > `::=` `?*`<значення–типу–symbol>`*`.

CLIPS дозволяє використовувати будь-яку кількість конструкторів `defglobal`. Необов'язковий параметр <ім'я–модуля> вказує модуль, у якому мають бути визначені змінні. Якщо ім'я модуля не задано, то змінні будуть поміщені в поточний модуль.

Глобальні змінні застосовуються в будь-якому місці, де можуть бути використані змінні, створені в лівій частині правил за деякими винятками.

По-перше, глобальні змінні не можуть використовуватися як параметри в конструкторах `deffunction`, `defmethod` або обробниках повідомлень.

По-друге, глобальні змінні не можуть використовуватися для отримання нових значень у лівій частині правил.

Приклад 2.9 використання конструктора `defglobal`, який декларує змінні дійсного та текстового типу, а також змінну зі значенням `symbol`:

```
(defglobal
?*d* = 7.8
?*e* = "string"
?*f* = symbol).
```

Приклад 2.10 використання конструктора `defglobal`, після виконання якого з'являться 4 глобальні змінні: `x`, `y`, `z` та `q`:

```
CLIPS> (defglobal
?*x* = 3
?*y* = ?*x*
?*z* = (+ ?*x* ?*y*)
?*q* = (create$ a b c))
CLIPS> ?*x*
3
CLIPS> ?*y*
3
CLIPS> ?*z*
6
CLIPS> ?*q*
(a b c)
CLIPS>
```

Глобальній змінній `x` присвоюється значення 3, глобальній змінній `y` – значення, збережене в глобальній змінній `x` (тобто ціле значення, яке дорівнює 3), а глобальна змінна `z` – це сума значень `x` і `y` (тобто 6).

Необхідно звернути увагу на те, що змінна `y` не є покажчиком на змінну `x`, просто їхні значення в цей момент збігаються. Якщо змінити значення `x`, значення змінних `y` і `z`, незважаючи ні на що, залишаться такими, що дорівнюють 3 і 6 відповідно.

Після виконання команди `reset` усі глобальні змінні отримують початкові значення, які визначені в конструкторі.

2.3.2 Команди та функції для роботи з глобальними змінними

Для маніпулювання конструкторами `defglobal` передбачено декілька команд.

1. Команда `ppdefglobal` (скорочено від `pretty print defglobal`– структуроване виведення конструктора `defglobal`) виводить у діалогове вікно системи визначення заданої глобальної змінної. Ім'я глобальної змінної має бути задано без знаку питання і символів «*», тобто тільки ім'я змінної (`name`) для глобальної змінної `?*name*`:

`(ppdefglobal name)`.

2. Команда `list-defglobals` призначена для відображення в діалоговому вікні списку імен усіх визначених у системі глобальних змінних:

(`list-defglobals` [<ім'я–модуля>]).

Якщо необов'язковий параметр <ім'я–модуля> не вказано, то ця команда виводить імена глобальних змінних, визначених у поточному модулі. Якщо параметр містить ім'я конкретного модуля, команда `list-defglobal` виводить список змінних, визначених у заданому модулі. За використання символу «*» ця команда виведе в діалогове вікно імена всіх глобальних змінних, визначених в усіх модулях системи.

3. Команда `show-defglobals`, на відміну від команди `list-defglobals`, виводить у діалогове вікно CLIPS не тільки імена глобальних змінних, але й їхні значення. В іншому ці дві команди практично ідентичні:

(`show-defglobals` [< ім'я–модуля >]).

4. Команда `undefglobal` призначена для видалення визначених користувачем глобальних змінних:

(`undefglobal` <ім'я–глобальної–змінної>).

Як параметр <ім'я–глобальної–змінної> допускається використання символу «*». У цьому випадку команда спробує видалити всі визначені користувачем глобальні змінні. Якщо глобальна змінна вказана, наприклад, у визначенні функції, видалення цієї змінної закінчиться невдачею.

5. Команда `watch` використовується для відстеження роботи конструктора `defglobal`:

(`watch globals`).

6. Функція `get–defglobal–list` повертає багатозначне значення, яке вміщує список задекларованих конструкцій `defglobal`, і має такий синтаксис:

(`get–defglobal–list`) .

7. Функція `bind` дозволяє змінити значення змінної `defglobal`. Для цього достатньо замість вказання локальної змінної вказати глобальну змінну. Функція `bind` має такий синтаксис:

(`bind` <ім'я–змінної> <вираз>*).

Параметр <вираз> є необов'язковим, і якщо він не заданий, то змінній буде встановлено початкове значення, задане в конструкторі `defglobal`. У разі, якщо <вираз> було задано, то його значення буде обчислено і результат присвоєно змінній. Якщо було задано кілька виразів, всі вони будуть обчислені,

із їхніх результатів буде сформоване складове поле, яке буде присвоєно глобальній змінній.

Функція `bind` повертає значення `FALSE` в разі, якщо змінній з будь-якої причини не було присвоєно ніякого значення. В іншому випадку функція повертає значення, присвоєне змінній.

Оскільки змінні в CLIPS слабо типізовані, типи значень, що присвоюються одній і тій саме змінній, у різні моменти часу можуть не збігатися.

2.4 Родові функції

У мові CLIPS передбачена можливість не тільки визначати функції з використанням конструктора `deffunction`, але й оголошувати універсальні функції. *Універсальна функція* становить групу взаємопов'язаних функцій, які називають методами, у яких спільно використовується загальне ім'я. Насправді універсальна функція, яка вміщує більше одного методу, називається *перезавантаженою*, оскільки надає можливість посилатися більше, ніж на один метод. Кожен метод у групі має свою власну сигнатуру, яка характеризує кількість та типи параметрів, що приймає метод. Під час оброблення виклику універсальної функції у системі CLIPS аналізуються параметри та викликається на виконання метод із сигнатурою, який відповідає параметрам, якщо є такий метод. Цей процес називається «виклик перезавантаженого методу (generic dispatch)».

Родові функції, як і звичайні, можна створити безпосередньо в CLIPS. Спосіб виклику цих функцій такий самий, як і спосіб виклику звичайних функцій, їхня відмінність у можливості перезавантажуватися. Завдяки механізму перезавантаження родова функція може виконувати різні дії залежно від типу та числа аргументів. Наприклад, функція + може виконувати як операцію конкатенації рядків, так і просте арифметичне додавання чисел.

Родові функції можуть вміщувати як системні методи, так і методи, які визначені користувачем. Наприклад, перезавантажена функція + складається з двох методів:

– неявний метод, який є системною функцією, що обробляє арифметичне додавання;

– явний (визначений користувачем) – обробник додавання рядків.

CLIPS не дозволяє використовувати функції, які створені за допомогою конструктора `deffunction`, як методи родових функцій. Конструктор `deffunction` надає можливість додавання в CLIPS нових функцій без використання концепції перезавантаження. Родова функція, яка має тільки один метод, за своїм поводженням ідентична функції, яка створена за допомогою конструктора `deffunction`.

Родові функції можуть використовувати класи як аргументи своїх методів, але вони повинні забезпечувати видачу повідомлень для маніпуляції з

об’єктами таких класів. Родові функції дозволяють виконувати різні дії залежно від набору аргументів, заданих під час виклику функції.

Для визначення загального імені універсальної функції, яке використовується групою методів, застосовується конструктор `defgeneric`.

Конкретні методи для універсальної функції визначаються з використанням конструктора `defmethod`.

2.4.1 Створення родової функції

Родова функція складається з заголовку та декількох методів. Заголовок родової функції може бути або явно визначено користувачем, або неявно задекларовано визначенням методу. Оголошення методу складається з 6 елементів:

– ім’я, яке відображає, до якої основної функції належить метод;
– необов’язковий індекс;
– необов’язкові коментарі;
– набір обмежень для параметрів;
– необов’язковий груповий параметр для обробки змінного числа аргументів;
– послідовність дій або виразів, які будуть виконані в заданому порядку в момент виклику методу.

Для створення заголовку родової функції використовується конструктор `defgeneric`, а для створення кожного нового методу родової функції – конструктор `defmethod`.

Кожен параметр методу може бути визначено з деякими довільними комплексними обмеженнями або без них. Параметри можуть мати два типи обмежень: обмеження типу та обмеження запитом. Наприклад, метод

```
(defmethod foo ((?a INTEGER) (?b INTEGER (> ?a ?b))))
```

спочатку задовольняє два відповідних обмеження типу `INTEGER`, а потім обчислює обмеження запитом.

Якщо всі аргументи родової функції, які використовуються, є такими, що застосовані до обмежень методу, то метод є застосованим для цього набору аргументів.

Приклад 2.11 методу перезавантаження системної функції `+`:

```
(defmethod + ((?a STRING) (?b STRING)
str-cat ?a ?b))
(+ 1 2)
(+ "foo" "bar")
(+ "foo" "bar" "mam")).
```

Перше звертання до родової функції + викличе виконання системної функції + як неявний метод, який виконує арифметичне додавання. Другий виклик спричинить виконання явного методу родової функції, яка виконує конкатенацію рядків, тобто обидва аргументи є рядками. Третій виклик згенерує помилку, оскільки явний метод для конкатенації рядків приймає тільки два аргументи, а неявний метод для арифметичного додавання не приймає рядкові аргументи взагалі.

Приклад 2.12 версії методу функції +, яка знаходить полусуму будь-якої кількості парних цілих чисел:

```
(defmethod +
       (($?any INTEGER (evenp ?current-argument))
       (div (call-next-method) 2)).
```

У прикладі 2.12 використана спеціальна змінна `?current-argument` для посилання на окремі аргументи, які об'єднані груповим символом. Ця змінна існує тільки в обмеженні запитом і не має значення в тілі методу.

У прикладі 2.13

```
(defmethod foo
         (($?any (> (length$ ?any) 2)))
         TRUE)
    (foo 1 red 3)
```

функції > та `length$` викликаються 3 рази для кожного з трьох аргументів, тому що обмеження за типом та запитом для групового параметра використовуються до кожного аргументу, який згруповано груповим символом, навіть якщо для перевірки використовуються функції для роботи зі складовими значеннями.

2.4.2 Родове зв'язування

Родовим зв'язуванням називається процес, за якого CLIPS самостійно розпізнає виклик родової функції та використовує аргументи для пошуку й запуску відповідного методу. У момент виклику родової функції CLIPS обирає метод із найвищим пріоритетом, для якого задовольняються всі обмеження параметрів. Цей метод виконується, а його значення повертається як значення родової функції.

Явний, визначений користувачем, метод є таким, що застосовується до виклику родової функції за таких трьох умов:

– ім'я співпадає з ім'ям родової функції;

– метод приймає не менше аргументів, ніж родова функція;

– кожен аргумент родової функції задовольняє відповідним обмеженням параметрів методу.

Рядок із відповідними обмеженнями можна отримати за допомогою функції `get-function-restriction`. Визначення неявного методу можна продивитися функціями `list-defmethods` або `get-method-restrictions`.

CLIPS виконує метод із найвищим пріоритетом, якщо два або більше методів є такими, що застосовуються до деякого виклику родової функції. Пріоритет методу визначається в момент його створення порівнянням обмежень параметрів для пар методів. Метод із більшою кількістю заданих обмежень параметрів має більший пріоритет.

Приклад 2.14 процесу визначення пріоритету (за зменьшенням пріоритету) методів для декількох прикладів перезавантаження функції +:

– `(defmethod + ((?a INTEGER (> ?a 2)) (?b NUMBER)))`

метод, який має обмеження і типом, і запитом для класу `integer`;

– `(defmethod + ((?a INTEGER) (?b INTEGER)))`
`(defmethod + ((?a INTEGER) (?b NUMBER)))`

методи, які мають обмеження типу для класу `integer`;

–`(defmethod + ((?a NUMBER) (?b INTEGER (> ?b 2))))`

метод, який має обмеження типу для класу `number`.

Перший метод має більш високий пріоритет, тому що обмеження параметрів і типом, і запитом має більш високий пріоритет, ніж обмеження тільки типом. Два інших методи мають більший пріоритет, ніж третій метод, тому що `integer` є підкласом `number`.

Якщо один із методів родової функції викликається іншим, то такий метод називається прихованим. Зазвичай тільки один метод має бути застосований до конкретного виклику родової функції. Якщо для цього виклику існує більше одного методу, який застосовується, родове зв'язування виконає метод із найвищим пріоритетом. Такий підхід називається *декларативним* методом родового зв'язування.

За допомогою функцій `call-next-method` та `override-next-method` метод родової функції може викликати деякий інший метод цієї родової функції (приховати виклик). Такий підхід називається *імперативним* методом (після виклику деякого методу він відіграє роль родового зв'язування).

Значення, що повертається родовою функцією, є значенням, яке повертається методом із найвищим пріоритетом. Значення, яке повертається методом, є останньою дією, яка обчислена в діях цього методу.

Для відстеження роботи універсальних функцій використовується команда `watch`, яка має такий синтаксис:

`(watch_generic_functions)`
`(watch methods)`.

2.4.3 Команди та функції для роботи з конструктором defgeneric

У мові CLIPS передбачено декілька команд для маніпулювання конструкторами defgeneric.

1. Команда ppdefgeneric (скорочено від pretty print defgeneric – структуроване виведення конструктора defgeneric) призначена для відображення текстового представлення конструктора defgeneric.
2. Команда undefgeneric призначена для знищення конструктора defgeneric.
3. Команда list-defgeneric використовується для відображення списку конструкцій defgeneric.
4. Функція get-defgeneric-list повертає багатозначне значення, що вміщує список конструктора defgeneric.

2.4.4 Команди та функції для роботи з конструктором defmethod

У мові CLIPS передбачено декілька команд для маніпулювання конструкторами defmethod.

1. Команда ppdefmethod (скорочено від pretty print defmethod – структуроване виведення конструктора defmethod) призначена для відображення текстового представлення конструктора defmethod.
2. Команда undefmethod призначена для знищення конструктора defmethod.
3. Команда list-defmethod використовується для відображення списку конструкцій defmethod.
4. Функція get-defmethod-list повертає багатозначне значення, що вміщує список конструктора defmethod.
5. Функція call-specific-method призначена для виклику методів родових функцій.

3 ПРОГРАМУВАННЯ НА ПІДСТАВІ ПРАВИЛ

Евристичний механізм подання знань у CLIPS реалізується за допомогою фактів і правил.

3.1 Факти

Факти – одна з основних форм подання даних у CLIPS (існує також можливість подання даних у вигляді об'єктів (екземплярів, які створюються на основі класів, визначених за допомогою мови COOL), і глобальних змінних). Кожен факт становить певний набір даних, що зберігається в поточному списку фактів – робочої пам'яті системи. Список фактів становить універсальне сховище фактів і є частиною бази знань. Обсяг списку фактів обмежений тільки пам'яттю комп'ютера. Список фактів зберігається в оперативній пам'яті комп'ютера, але CLIPS надає можливість зберігати поточний список у файлі, а завантажувати його із раніше збереженого файлу.

У CLIPS фактом є список неподільних (атомарних) значень примітивних типів даних. CLIPS підтримує два типи фактів: упорядковані (`ordered facts`) та неупорядковані (`non-ordered facts` або `template facts`).

Прикладом упорядкованого факту є такий запис (`voltage is 220 volt`), а неупорядкованого, у якому порядок слотів не має значення, наступний: `(meeting (subject "AI") (chief "Petrova") (Room "218"))`.

Посилатися на дані, які вміщуються в факті, можливо або використовуючи строго задану позицію значення в списку даних для упорядкованого факту, або вказавши ім'я значення для шаблонів.

Факти можливо додавати, знищувати, змінювати та дублювати. Після додавання факту в список фактів йому присвоюється унікальний ідентифікатор, який називається індексом факту (`fact-index`). Індекс першого факту дорівнює 0, у подальшому індекс збільшується на одиницю за додавання кожного нового факту.

Ідентифікатор факту складається з символів: «`f`, `-`, індексу факту», наприклад, ідентифікатор `f-10` посилається на факт з індексом 10.

Команди змінення, знищення або дублювання вимагають указання визначеного факту. Факт можна задати або індексом факту або його адресою. Адреса факту – це зміна–вказівник, яка зберігає індекс факту.

Упорядковані факти складаються з поля, яке обов'язково є даними типу `symbol`, та наступною за ним послідовністю полів, розділених пробілами та обмеженими круглими дужками:

(значення–типу–`symbol` [поле]*).

Перше поле факту визначає так зване відношення або зв'язок факту (relation). Термін зв'язок означає, що цей факт належить деякому визначеному конструктором або неявно оголошеному шаблону. Оскільки упорядкований факт для подання інформації використовує строго задані позиції даних, то для доступу до неї користувач повинен знати, не тільки, які дані збережені в факті, а й яке поле містить ці дані, до того ж кількість полів у факті необмежена. Приклади упорядкованих фактів наведено далі:

```
(the pump is on)
(altitude is 1000 feet).
```

Неупорядковані факти (або шаблони) надають користувачеві можливість задавати абстрактну структуру факту шляхом призначення імені кожному полю.

3.1.1 Конструктори для роботи з фактами

1. Конструктор deftemplate

Для створення шаблонів, які згодом будуть застосовуватися для доступу до полів факту за іменем, використовується конструктор deftemplate, який утворює ядро програми CLIPS, оскільки вводить знання програміста в середовище CLIPS, і цим відрізняється від функцій та команд. Як і всі конструктори, конструктор deftemplate не повертає ніяке значення.

Під час створення неупорядкованого факту конструктор deftemplate задає ім'я шаблону та визначає послідовність із нуля або більше полів неупорядкованого факту, які називаються *слотами*. Слот складається з ім'я, заданого значенням типу symbol, та списку полів, які йдуть за ним. Як і факт, слот з обох боків обмежується круглими дужками. На відміну від упорядкованих фактів слот невпорядкованого факту може жорстко визначати тип своїх значень. Крім того, слоту можуть бути задані значення за замовчуванням. Синтаксис цього конструктора такий:

(deftemplate <ім'я–шаблону> [<коментарі>]
[<визначення–слота> *]).

Порядок слотів у неупорядкованих фактах не має значення.

Слоти не можна використовувати в упорядкованих фактах.

Приклад 3.1 конструктора deftemplate, який описує факт person:

```
CLIPS> (deftemplate person "Example"
(slot name)
(slot age))
CLIPS>
```

Приклад 3.2 використання конструктора `deftemplate`:

```
CLIPS> (deftemplate MyObject
(slot name)
(slot location)
(slot weight)
 (multislot contents))
CLIPS>
```

У прикладі 3.2 існує два шаблони:

– перший шаблон є визначеним шаблоном `initial-fact`, який не має слотів і завжди додається під час запуску системи;

– другий шаблон – це шаблон `MyObject`.

Ключове слово `multislot` використовується для визначення складового слота, який здатний зберігати не одиницю інформації одного з примітивних типів даних, а список подібних одиниць інформації необмеженого обсягу. Під час створення шаблону кожному полю можна призначити певні атрибути, що задають значення за замовчуванням або обмежити значення слота. Якщо встановлений атрибут за замовчуванням `default? DERIVE`, то значення будуть вилучатися з обмеження. Якщо використовується атрибут `default? NONE`, то значення поля має бути задано в момент виконання операції додавання факту.

Приклад 3.3 використання атрибутів обмеження:

```
(deftemplate object
        (slot name
                (type SYMBOL)
                (default ?DERIVE))
        (slot location
                (type SYMBOL)
                (default ?DERIVE)))
```

Для перегляду всіх визначених у поточній базі даних шаблонів можна використовувати команду `get-deftemplate-list` або спеціальний інструмент *Deftemplate Manager* (меню *Browse/Deftemplate Manager*).

У мові CLIPS передбачено низку атрибутів слота, які можуть бути задані під час визначення слотів конструктора `deftemplate`.

1) Атрибут `type` визначає типи даних, які можуть зберігатися в слоті: змінна `?Variable` або один чи декілька символів `symbol`, `string`, `lexeme`, `integer`, `float`, `number`, `instance-name`, `instance-address`, `instance`, `external-address`, `fact-address`. Якщо використовується

змінна ?Variable, то слот може вміщувати дані будь-якого типу. Використання специфікації типу lexeme еквівалентно завданню специфікацій symbol та string. Використання специфікації типу number еквівалентно завданню специфікацій integer та float, а використання специфікації типу instance еквівалентно завданню специфікацій instance-name та instance-address. Атрибут type регламентує перелік допустимих типів.

Приклад 3.4 використання конструктора deftemplate з ім'ям person, який обмежує значення, що зберігаються в слоті name, символами та значенням, а в слоті age – цілими числами:

```
(deftemplate person
       (multislot name (type symbol))
       (slot age (type integer))).
```

2) Атрибути допустимого значення для конкретного типу: allowed–symbols, allowed–strings, allowed–lexemes, allowed–integers, allowed–floats, allowed–numbers, allowed–instance-names та allowed–values. За кожним із цих атрибутів повинно йти або позначення змінної ?Variable, або список значень цього типу, що йде за префіксом allowed–. За замовчуванням атрибут допустимого значення для слотів має вигляд: allowed–values ?Variable.

Приклад 3.5 використання конструктора deftemplate з ім'ям person, у якому реалізована можливість обмежити перелік допустимих символів для слота gender:

```
(deftemplate person
       (multislot name (type symbol))
       (slot age (type integer))
       (slot gender (type symbol)
       (allowed-symbols male female))).
```

3) Атрибут range дозволяє задати мінімальні та максимальні допустимі числові значення.

Приклад 3.6 використання конструктора deftemplate з ім'ям person для запобігання можливості поміщати в слот age від'ємне значення віку:

```
(deftemplate person
       (multislot name (type symbol))
       (slot age (type integer) (range 0 ?Variable))).
```

4) Атрибут `cardinality` (кардинальність) дозволяє задавати мінімальну та максимальну кількість значень, які можуть зберігатися в конструкції `multislot`. За замовчуванням атрибут `cardinality` для будь-якого багатозначного значення для конструкції `multislot` має вигляд: (`cardinality ?Variable ?Variable`).

Приклад 3.7 використання конструктора `deftemplate`, у якому описано склад волейбольної команди з шести гравців та двох запасних гравців, крім того у конструкції `multislot` використовуються обмеження типу допустимого значення та діапазону:

```
(deftemplate volleyball-team
       (slot name (type string))
       (multislot players (type string)
            (cardinality 6 6))
       (multislot alternates (type string)
            (cardinality 0 2))).
```

5) Атрибут `default` дозволяє задавати значення, яке використовується за замовчуванням. Якщо в атрибуті `default` задано значення `?DERIVE`, то для цього слоту має бути введено логічним шляхом визначене значення, яке відповідає всім атрибутам слота. Якщо в атрибуті `default` задано позначення `?NONE`, то необхідно передбачити використання деякого значення для цього слота під час введення факта в список фактів, тобто в такому випадку значення, що використовується за замовчуванням, не передбачено.

Якщо для слота не задано атрибут, то передбачається, що цей атрибут має вигляд: (`default ?DERIVE`).

2. Конструктор `deffacts`

Конструктор дозволяє визначати список фактів, які будуть автоматично додаватися щоразу після виконання команди `reset`, яка очищає поточний список фактів та додає в список фактів факт `initial-fact`. Синтаксис конструктора `deffacts` такий:

(`deffacts` <ім'я–списку–фактів>[коментарі]
[<факт>*].

Додавання конструктора `deffacts` з ім'ям вже існуючого конструктора призведе до видалення попереднього конструктора.

У прикладі 3.8 за використання конструктора `deffacts` третій факт вміщує вираз для обчислення суми трьох констант:

```
(deffacts startup "refrigerator"
        (refrigerator light on)
        (refrigerator door open)
        (refrigerator temp (+5 10 15))).
```

Під час запуску та після виконання команди `clear` CLIPS автоматично конструює такі зумовлені шаблони і факти:

```
(deftemplate initial-fact)
(deffacts initial-fact
(initial-fact)).
```

Зумовлений факт `initial-fact` шаблону `initial-fact` надає зручний спосіб для запуску програм на мові CLIPS, тобто правила, які не мають умовних елементів, автоматично перетворюються в правила з умовою, яка перевіряє наявність факту `initial-fact`. Факт `initial-fact` можна обробляти так само, як і всі інші факти CLIPS, які додані користувачем або програмою за допомогою функції `assert`.

3.1.2 Функції та команди для роботи з фактами

Усі факти, які відомі системі CLIPS, групуються та зберігаються в списку фактів. Цей список фактів дозволяє обробляти факти, які надають інформацію.

1. Функція `assert` дозволяє додавати факти в список фактів поточної бази знань. Кожним викликом цієї функції можна додати довільне число фактів.

Синтаксис функції `assert` такий:

(`assert` <факт> +).

Перше поле факту обов'язково має бути типу `symbol`.

У разі вдалого додавання фактів у базу знань, функція повертає адресу останнього доданого факту.

У прикладі 3.9 використовуються і команди CLIPS, і функція `assert`.

Команду `clear`, що очищає поточний список фактів, можна набирати з клавіатури, або обрати пункт *Clear* меню *Execution*.

На відміну від команди `reset`, команда `clear` не додає в список фактів факт `initial-fact`, що є системним початковим фактом, який створюється в робочій пам'яті інтерпретатора CLIPS за командою `reset` до запуску програми на виконання.

```
CLIPS> (clear)
CLIPS> (deftemplate status (slot temp) (slot pressure (default low)))
CLIPS> (assert (color red))
<Fact-0>
CLIPS> (assert (color blue)
            (value (+ 3 4)))
<Fact-2>
CLIPS> (deftemplate status
                     (slot temp)
                     (slot pressure
                     (default low)))
CLIPS> (assert (status (temp high)))
<Fact-3>
CLIPS> (facts)
f-0     (color red)
f-1     (color blue)
f-2     (value 7)
f-3     (status (temp high) (pressure low))
For a total of 4 facts.
CLIPS>
```

Системний початковий факт, який незалежно від фактів у базі даних, активізує деяке правило, додає такі факти, які, зі свого боку, активують правила, умови яких не виконувалися в початковий момент. Отже, цей факт використовується для того, щоб почати обчислення, тому що в CLIPS–програмах поширеними правилами є такі, які додають факти в базу даних, або, навпаки, видаляють їх. Типовою є ситуація, коли під час старту програми в базі даних немає фактів, які відповідають хоча б одному правилу.

У прикладі 3.9 використання функції `assert` при ініціалізації факту `value` використовувався вираз, а слот `pressure` невпорядкованого факту `status` отримав значення за замовчуванням.

Команда `facts` виводить список фактів.

За замовчуванням CLIPS не дозволяє додати два однакових факти. Цю установку можна змінити за допомогою функції `set-fact-duplication` або використати меню *Execution*, команду *Options*, діалогове вікно *Execution Options*, і встановити прапорець *Fact Duplication*.

2. Функція `retract` видаляє факти з поточного списку фактів. Кожним викликом цієї функції можна видалити довільну кількість фактів.

Синтаксис функції `retract` такий:

`(retract` <визначення–факту> `+ | *)`.

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), або вираз, який обчислює цей індекс (наприклад, (+ 1 2) для факту з індексом f–3).

Якщо як аргумент функції `retract` використовувати символ «*», то з бази будуть видалені всі факти (`retract *`).

Для функції `retract` не має значення, що повертається.

Наприклад, за допомогою функції `assert` додаємо факти:

```
(assert (a) (b) (c) (d) (e) (f)),
```

а функцією `retract` видаляємо факти з парними індексами, використовуючи індекс факту безпосередньо (перший аргумент) і вираз, який обчислює індекс факту (другий і третій аргумент):

```
(retract 0 (+ 0 2) (+0 2 2)).
```

Для видалення всіх фактів використовується команда:

```
(retract *).
```

3. Функція `modify` використовується для зміни невпорядкованих фактів і дозволяє змінювати значення слотів таких фактів. За один виклик функція `modify` дозволяє змінювати тільки один факт. У разі успішного виконання функція повертає новий індекс модифікованого факту. Команда `modify` розглядається як команда `retract`, за якою йде команда `assert`.

Синтаксис функції `modify` такий:

```
(modify <визначення–факту>
                <нове–значення–слота> +).
```

Аргументом <визначення–факту> може бути або змінна, пов’язана з адресою факту за допомогою правила, або індекс факту без префікса (наприклад, 3 для факту з індексом f–3). Після визначення факту слідує список з одного або більше нових значень слотів зазначеного шаблону.

Приклад 3.10 використання функції `modify`:

```
CLIPS> (clear)
CLIPS> (deftemplate temperature (slot value))
CLIPS> (assert (temperature (value low)))
<Fact-0>
CLIPS> (facts)
f-0     (temperature (value low))
For a total of 1 fact.
CLIPS> (modify 0 (value high))
<Fact-1>
CLIPS> (facts)
f-1     (temperature (value high))
For a total of 1 fact.
CLIPS>
```

Під час виконання функції `modify` CLIPS спочатку видаляє старий факт із індексом f–0, а потім додається новий факт із індексом f–1, ідентичний попередньому, але з новим значенням заданого слота.

Під час роботи з упорядкованими фактами, якщо в список існуючих упорядкованих фактів раніше був доданий факт (`temperature is low`), який отримав індекс 0, то змінити його значення можна так:

```
(clear)
(assert (temperature is low))
(retract 0)
    (assert (temperature is high)).
```

4. Функція `duplicate` створює новий невпорядкований факт заданого шаблону та копіює в нього визначену користувачем групу полів вже існуючого факту того саме шаблону. Одним викликом цієї функції можна створити одну копію деякого заданого факту. У разі успішного виконання функція повертає індекс нового факту.

Синтаксис функції `duplicate` такий:

```
(duplicate <визначення–факту>
<нове–значення–слота>+).
```

Аргументом <визначення–факту> може бути або змінна, пов’язана з адресою факту за допомогою правила, або індекс факту без префікса.

Приклад 3.11 створення копії існуючого невпорядкованого факту:

```
CLIPS> (deftemplate car
          (slot name)
          (slot producer)
          (slot type)
          (slot max-speed))
CLIPS> (assert (car
                  (name scorpio)
                  (producer ford)
                  (type sedan)
                  (max-speed 180)))
<Fact-0>
CLIPS> (duplicate 0
                  (type off-road)
                  (max-speed 230))
<Fact-1>
CLIPS>
```

У прикладі 3.11 визначається шаблон, що описує властивості автомобіля, і додає факт – автомобіль Ford Scorpio з типом кузова «Седан» і максимальною швидкістю 180 км/год. Далі додається факт з інформацією про ще один автомобіль із подібними характеристиками – це позашляховик «Ford Scorpio» з максимальною швидкістю 230 км/год.

5. Функція `assert-string` приймає як єдиний аргумент символьний рядок, який є текстовим поданням факту, і додає його в список фактів. Функція працює як із впорядкованими, так і неупорядкованими фактами. Одним викликом функція додає тільки один факт.

Синтаксис функції `assert-string` такий:

`(assert-string` <рядковий–вираз>`).`

Аргумент <рядковий–вираз>) має бути укладений в дужки. Функція перетворює заданий <рядковий–вираз> у факт, розділяючи окремі слова на поля, з урахуванням визначених у системі на поточний момент шаблонів. Приклад використання функції такий:

```
(assert-string "(book-name\"CLIPS\")").
```

6. Функція `fact-existp` визначає, чи присутній у цей момент факт, заданий індексом або змінною покажчика, у базі знань системи. У разі, якщо факт присутній у списку фактів, функція повертає значення `TRUE`, інакше – значення `FALSE`.

Синтаксис функції `fact-existp` такий:

(fact-existp < визначення–факту>).

Приклад 3.12 використання функцій assert-string та fact-existp:

```
(clear)
(assert-string "(a\\b \"c\\\\d\")")
    (fact-existp 0)
(retract 0)
    (fact-existp 0).
```

7. Функція fact-relation для роботи з неупорядкованими фактами дозволяє отримати зв'язок (relation) існуючого факту з шаблоном. Зв'язок факту з шаблоном, який визначено за допомогою конструктора deftemplate, або неявно створеним шаблоном, визначається за першим полем факту. Це поле завжди є простим полем і використовується CLIPS як ім'я шаблону, з яким пов'язаний факт. Отже, функція fact-relation повертає перше поле факту або значення FALSE, якщо зазначений факт не знайдений.

Приклад 3.13 використання функції fact-relation:

```
CLIPS> (assert (car Ford))
(fact-relation 0)
(retract 0)
(fact-relation 0)
<Fact-1>
CLIPS> (fact-relation 0)
initial-fact
CLIPS> (fact-relation 1)
car
CLIPS>
```

У першому випадку функція fact-relation повертає значення car, а в другому – значення FALSE.

8. Функція fact-slot-names повертає список імен слотів у складовому полі. Для впорядкованих фактів функція повертає значення implied (той, що мається на увазі), тому що CLIPS подає впорядковані факти як неявно задані невпорядковані з одним складовим слотом. Використання функції fact-slot-names у прикладі 3.14 поверне значення (name producer type max-speed):

```
CLIPS> (deftemplate car
(slot name)
(slot producer)
(slot type)
(slot max-speed))
CLIPS> (assert (car
(name scorpio)
(producer ford)
(type sedan)
(max-speed 180)))
<Fact-0>
CLIPS> (fact-slot-names 0)
(name producer type max-speed)
CLIPS>
```

9. Функція `fact-slot-value` дозволяє отримувати значення слота деякого заданого факту. Якщо факт є впорядкованим, то для отримання значення неявно визначеного складового слота використовується значення `implied`. У прикладі 3.15 наведено використання функції `fact-slot-value`:

```
CLIPS> (deftemplate foo
(slot bar)
(multislot yak))
CLIPS> (assert (foo (bar 1) (yak 2 3 )))
<Fact-0>
CLIPS> (fact-slot-value 0 bar)
1
CLIPS> (fact-slot-value 0 yak)
(2 3)
CLIPS> (assert (another a b c))
<Fact-1>
CLIPS> (fact-slot-value 1 implied)
(a b c)
```

10. Команда `save-facts` зберігає факти з поточного списку фактів у текстовий файл із указанням обмеження області видимості: чи зберігаються факти, які присутні в цей момент у системі (ключове слово `visible`), чи зберігаються факти з поточного модуля (ключове слово `local`). Після обмеження області видимості може йти список визначених у системі шаблонів.

На кожний факт відводиться один рядок.

Розглянемо приклад 3.16, послідовність дій якого зберігається у файлі `f1`, який розташовано в поточному каталозі, всі факти, якого видимі в поточному модулі та пов’язані з шаблонами `template` та `simple-fact1`:

```
CLIPS> (deftemplate template
(slot a)
(slot b))
CLIPS> (assert (template (a 1) (b 2)))
<Fact-0>
CLIPS> (assert (simple-fact1) (simple-fact2))
<Fact-2>
CLIPS> (save-facts f1 local template simple-fact1)
TRUE
CLIPS>
```

У результаті успішного збереження списку фактів створено текстовий файл з таким змістом:

```
template (a 1) (b 2))
(simple-fact1).
```

11. Команда `load-facts` використовується для завантаження збережених раніше файлів. Якщо у файлі є факти, що пов'язані з явно створеними за допомогою конструктора `deftemplate` шаблонами, то під час завантаження всі необхідні шаблони мають бути вже визначені в системі. CLIPS дозволяє завантаження конструкторів із текстового файлу.

12. Команда `watch` використовується для відстеження фактів і є корисним засобом відлагодження програми. Синтаксис команди такий:

(`watch facts`).

3.2 Правила

Експертна система зможе виконувати роботу тільки в тому випадку, якщо в ній присутні не тільки факти, але й правила.

Правила в CLIPS використовують для подання евристик, які визначають набір дій, що виконуються у разі появи деякої ситуації. Визначення правила має такий загальний формат:

(`defrule` <ім'я–правила> [<коментарі>]
`(умова)` {синоніми: антецеденти в логіці,
ліва частина – LHS у термінах CLIPS}
=>
`(дія)` {синоніми: консеквенти в логіці,
права частина – RHS у термінах CLIPS}.

Користувач визначає набір правил, які разом працюють над розв'язанням деякої задачі. Правила складаються з передумов та наслідків. Передумови називаються також ЯКЩО–частиною правила, лівою частиною правила або LHS (left-hand side of rule). Наслідок називається ТО–частиною правила, правою частиною правила або RHS правила (right-hand side of rule).

У довідковій системі та документації із CLIPS для позначення передумов правила найчастіше використовується термін «`LHS of rule`», а для позначення слідства – «`RHS of rule`», тому в подальшому буде використовуватися ліва та права частина правила [3–5].

Передумови правила становлять набір умов (або умовних елементів), які повинні задовольнятися, для того щоб правило виконувалося. Передумови задовольняються залежно від наявності або відсутності деяких заданих фактів у списку фактів або деяких створених об'єктів, які є екземплярами класів, визначених користувачем. Одним із поширеніших типів умовних виразів у CLIPS є зразки (`patterns`, шаблони), які складаються з набору обмежувачів, що використовуються для визначення того, чи задовольняє деякий факт або об'єкт умовному елементу. Іншими словами, зразок задає деяку маску для фактів або об'єктів. Процес співставлення зразків фактам або об'єктам називається процесом зіставлення зразків (`pattern-matching`). CLIPS надає механізм, який називається механізмом логічного виводу (`inference engine`), який автоматично зіставляє зразки з поточним списком фактів та визначеними об'єктами в пошуках правил, які використовуються в певний момент. Якщо всі шаблони (зразки) правил узгоджуються з фактами, то правило активується та поміщається в робочий список правил – у колекцію активізованих правил. У робочому списку може знаходитися від нуля і більше правил.

Наслідки правила є набором деяких дій, які необхідно виконати у випадку, якщо правило застосовується до поточної ситуації. Отже, дії, які задані внаслідок правил, виконуються за командою механізму логічного виводу, якщо всі передумови правил були задовільнені. У випадку, якщо в певний момент можна виконати більше одного правила, то механізм логічного виводу використовує так звану стратегію розв'язання конфліктів, яка визначає, яке правило буде виконуватися. Після чого CLIPS виконує дії, які описані внаслідок вибраного правила, та починає вибір наступного правила. Цей процес продовжується доки список правил не закінчиться.

Для кращого розуміння сутності правил у CLIPS їх можна подати у вигляді оператора IF–THEN, який використовується в процедурних мовах програмування. Однак умови виразу IF–THEN у процедурних мовах обчислюються тоді, коли потік управління програмою безпосередньо потрапляє на цей вираз шляхом послідовного перебирання виразів і операторів, що складають програму. У CLIPS, на відміну від цього, механізм логічного виводу створює і постійно модифікує список правил, умови яких у конкретний момент задовільнені.

Ці правила запускаються на виконання механізмом логічного виводу.

Під час старту програми, що містить множину фактів і правил, інтерпретатор CLIPS запускає машину логічного виводу, яка з'ясовує, які з правил можна активізувати. Це виконується циклічно, а кожен цикл складається з трьох кроків:

– зіставлення фактів і правил;

– вибір правила, що підлягає активізації;

– виконання дій, запропонованих правилом.

Отже, правила, які взаємодіють із базою даних у вигляді фактів, вносять у неї функціональність і утворюють разом із нею базу знань.

Механізм логічного виводу виконує дві основні функції:

– перегляд існуючих фактів та правил, а також додавання в базу фактів нових фактів;

– визначення порядку перегляду й застосування правил. Порядок може бути прямим та зворотним.

В ЕС із прямим виводом за відомими фактами відшуковується висновок, що виходить із цих фактів. Якщо такий висновок вдається знайти, він заноситься в базу фактів. Прямі виводи часто застосовуються в системах діагностики, їх називають виводами, керованими даними.

У системах зі зворотним виводом спочатку висувається деяка гіпотеза про кінцеве судження, а потім механізм виводу намагається знайти факти, які могли б підтвердити або спростувати висунуту гіпотезу. Процес пошуку необхідних фактів може містити значну кількість кроків, до того ж можливо висувати нові гіпотези (цілі). Зворотні виводи управляються цілями.

Механізм логічного виводу визначає порядок застосування правил, а також установлює, чи є ще факти, які можуть бути змінені у випадку продовження роботи (за немонотонного виводу). У циклі логічного виводу здійснюються такі кроки:

1) зіставлення (уніфікація) – антецедент правила співставляється з фактами з бази фактів;

2) дозвіл конфліктного набору – вибір одного з декількох правил у тому випадку, якщо їх можна застосувати одночасно;

3) спрацьовання правил – у випадку збігу антецедента правила з фактами, відбувається спрацювання правила, тобто воно позначається як використане;

4) дія – додавання консеквента правила, що спрацювало, у базу фактів, тобто формування нового факту. Якщо механізм логічного виводу аналізує фрейм, у якому значення слотів відповідають фактам, то він запускає на виконання керуючий слот фрейму (наприклад, процедури обчислення).

Вихід із циклу механізм логічного виводу здійснює декількома способами:

– вибірка всіх правил бази знань;

– використання метаправил, тобто правил, які керують іншими правилами.

Отже, ЕС не має у своєму розпорядженні готове рішення в просторі станів, а знаходить його у процесі логічного виводу на підставі даних, які одержані від користувача.

3.2.1 Основний цикл виконання правил

Після того як у систему додані всі необхідні правила та приготовлені початкові списки фактів і об'єктів, CLIPS готовий виконувати правила. У традиційних мовах програмування точка входу, точка зупинки і послідовність обчислень визначаються програмістом. У CLIPS потік виконання програми абсолютно не вимагає чіткого визначення. Знання (правила) і дані (факти й об'єкти) розділені, і механізм логічного виводу, що надається CLIPS, застосовує дані до знань, формуючи список застосованих правил, після чого послідовно виконує їх. Цей процес називається основним циклом виконання правил (`basic cycle of rule execution`). Послідовність дій (кроки), які виконуються системою CLIPS у цьому циклі в момент виконання програми, така [6]:

1) якщо була досягнута межа виконання правил або не було встановлено поточний фокус, виконання переривається. В іншому випадку для виконання вибирається перше правило модуля, на якому був встановлений фокус. Якщо в поточному плані виконання немає правил, що потребують виконання, то фокус переміщується по стеку фокусів і встановлюється на наступний модуль у списку. Якщо стек фокусів порожній, виконання припиняється, в іншому разі крок 1) виконується ще один раз;

2) виконання дій, описаних у правій частині обраного правила. Використання функції `return` може змінювати положення фокусу в стеці фокусів. Кількість запусків цього правила збільшується на одиницю для визначення меж виконання правила;

3) у результаті виконання кроку 2) деякі правила можуть бути активовані або дезактивовані. Активовані правила (тобто правила, умови яких задовольняються в цей момент) поміщаються в план вирішення завдання модуля, у якому вони визначені. Розміщення в плані визначається пріоритетом правила (`salience`, значущість) і поточною стратегією вирішення конфліктів.

Дезактивовані правила видаляються з поточного плану виконання завдання. Якщо для правила встановлено режим перегляду активацій, то користувач отримає відповідне інформаційне повідомлення під час кожної активації або дезактивації правила (режим перегляду активацій можна встановити за допомогою діалогового вікна *Watch options*. Для цього необхідно обрати команду *Watch* в меню *Execution* і встановити прапорець *Activations*);

4) якщо встановлено режим динамічного пріоритету (`dynamic salience`), то для всіх правил із поточного плану розв'язання задачі обчислюються нові значення пріоритету. Після цього цикл повторюється з кроку 1).

3.2.2 Створення правил. Конструктор `defrule`

Для додавання нового правила в базу знань CLIPS надає спеціальний конструктор `defrule`. У загальному вигляді синтаксис цього конструктора можна подати так:

```
(defrule
   <ім'я–правила>
    [<коментарі>]
    [<визначення–властивості–правила>]
   <передумови> , ліва частина правила
=>
     <наслідки> , права частина правила ).
```

Ім'я правила має бути значенням типу `symbol`. Як ім'я правила не можна використовувати зарезервовані слова CLIPS, які були перераховані раніше. Визначення правила може містити оголошення властивостей правила, яке йде безпосередньо після імені правила і коментарів.

Приклад 3.17 розроблення правила:

```
(defrule search-black-audi
       (cars (model "Audi")
       (color Black))
   =>
       (printout t "Є чорний Audi!" crlf)
  ).
```

Це правило активується тоді, коли в робочій пам'яті з'явиться факт із атрибутами (model «Audi») та (color Black). Активація правила не означає його виконання, а означає розміщення правила в робочий список правил (або `agenda` – план розв'язання задачі) у CLIPS. Для виконання активованих правил необхідно виконання команди (`run`).

Ліва частина правила задається набором умовних елементів, який, зазвичай, складається з умов, застосованих до деяких зразків. Заданий набір зразків використовується системою для зіставлення з наявними фактами і об'єктами. Усі умови в лівій частині правила об'єднуються за допомогою неявного логічного оператора `and`. Права частина правила містить список дій, які виконуються у разі активації правила механізмом логічного виводу. Правило не має обмежень на кількість умовних елементів або дій.

Якщо в лівій частині правила не вказано жоден умовний елемент, то CLIPS надасть умову–зразок `initial-fact` або `initial-object`. Отже, правило активується щоразу у разі появи в базі знань факту `initial-fact` або `initial-object`.

Якщо в правій частині правила не визначена жодна дія, правило може бути активовано та виконано, але, водночас, нічого не станеться.

Приклад 3.18 є найпростішим прикладом CLIPS–програми:

```
CLIPS> (defrule
Kharkiv
=>
(printout t crlf crlf)
(printout t ********************************** crlf)
(printout t O.M. Beketov National University of crlf)
(printout t Urban Economy in Kharkiv crlf)
(printout t ********************************** crlf)
(printout t crlf crlf)
)
CLIPS> (reset)
CLIPS> (run)
```

У наведеній програмі:

– команда `clear` повністю очищає систему, тобто знищує всі правила, факти та інші об'єкти бази знань, які додані конструкторами, та приводить систему в початковий стан, необхідний для кожної нової програми;

– конструктор `defrule` додає нове правило з ім'ям `Kharkiv`;

– ліва частина правила відсутня, тому CLIPS автоматично формує передумови з одного умовного речення (`initial-fact`). Під час запуску програми механізм логічного виводу CLIPS буде шукати в списку фактів факт (`initial-fact`), і якщо він там буде знайдений, активує правило;

– права частина складається з декількох викликів функції `printout`. Параметр `t` задає стандартний потік виведення – екран. Абревіатура `crlf (caret return line feed)` означає перехід на новий рядок;

– команда `reset` очищує робочу пам'ять середовища CLIPS та заносить у нього факт (`initial-fact`), що є важливим для нормального функціонування програми;

– команда `run` запускає механізм логічного виводу, що розпочинає виконання програми.

3.2.3 Синтаксис `LHS` правила

Ліва частина правила містить список умовних елементів, які повинні задовольнятися, для того щоб правило було розміщено в плані розв'язання задачі.

Існує 8 типів умовних елементів, які використовуються в лівій частині правил:

CEs-зразки, test CEs, and CEs, or CEs, not CEs, exists CEs, forall CEs, logical CEs.

Зразки є найвикористовуванішим умовним елементом (CEs – conditional elements), який містить обмеження, що існує для визначення, чи задовольняє який-небудь елемент даних (факт або об'єкт) зразкам.

Умова test CEs використовується для оцінки виразу як частини процесу зіставлення.

Умова and CEs застосовується для визначення групи умов, кожна з яких має бути задовільнена.

Умова or CEs використовується для визначення однієї умови з деякої групи умов, яка має бути задовільнена.

Умова not CEs використовується для визначення однієї умови з деякої групи, яка не має бути задовільнена.

Умова exists CEs використовується для перевірки наявності принаймні одного збігу факту з деяким заданим зразком.

Умова forall CEs використовується для визначення, що деяка задана умова виконується для всіх заданих умовних елементів.

Умова logical CEs дозволяє виконати додавання фактів і створення об'єктів у правій частині правила, які пов'язані з фактами й об'єктами, що збіглися із заданим зразком у лівій частині правила (підтримка вірогідності фактів у базі знань).

Розглянемо більш детально кожен із типів умовних елементів.

1. Зразок (pattern CE).

Зразок (pattern CE) – умовний елемент, який складається зі списку обмежень полів, групових символів (wildcards) і змінних, які використовуються для пошуку множини фактів або об'єктів, що відповідні заданому зразку. Отже, зразок як би визначає маску, якій повинні відповідати дані. Такий умовний елемент задовольняється будь-яким фактом або об'єктом, які відповідають заданим обмеженням.

Обмеження полів – це набір обмежень, які використовуються для перевірки простих полів або слотів об'єктів. Обмеження полів можуть складатися не тільки з одного символьного обмеження, а можливе об'єднання декількох обмежень разом. На додаток до символьних обмежень CLIPS підтримує зв'язувальні обмеження, предикатні та обмеження, які повертають значення.

Групові символи використовуються для зіставлення полів у зразках і під час зіставлення зразків у ситуації, коли просте поле або група полів можуть приймати будь-які значення.

Груповий символ для простого поля записується за допомогою символу «?», який відповідає одному будь-якому значенню, яке збережене в заданому полі.

Груповий символ складового поля записується за допомогою знака «$?» і відповідає, можливо, порожній послідовності полів, збереженої в складовому полі.

Змінні використовуються для збереження значення поля, яке може бути згодом використано в лівій частині правила для іншого умовного елемента, або в правій частині як аргумент дії.

Перше поле будь-якого зразка обов'язково має бути значенням типу `symbol`. CLIPS використовує перше поле для визначення, чи є цей зразок упорядкованим фактом, шаблоном або об'єктом. Ключове слово `object` зарезервовано для створення зразків, які призначені для зіставлення з об'єктами.

Розглянемо обмеження полів у зразках.

1) Основні обмеження, які використовуються в зразках, – це обмеження, які визначають точну відповідність між полями факту та зразком. Такі обмеження називаються *символьними*. Вони складаються з таких констант, як дійсні та цілі числа, значення типу `symbol`, рядка або імені об'єктів. Ці обмеження не можуть вміщувати групових символів або змінних. Усі символьні обмеження під час зіставлення зразків повинні точно співпадати за всіма вказаними полями, інакше факт не буде вважатися таким, що підійшов цьому зразку.

Приклад 3.19 правил, які використовують як зразок факти (як упорядковані, так і шаблони) з символьними обмеженнями. Для цього необхідно ввести конструктори фактів і шаблонів:

```
CLIPS> (deffacts    data-facts
(data 1.0 blue "red")
(data 1 blue)
(data 1 blue red)
(data 1 blue RED)
(data 1 blue red 6.9))
CLIPS> (deftemplate person
(slot name)
(slot age)
(multislot friends))
CLIPS> (deffacts people
(person (name Joe) (age 20))
(person (name Bob) (age 20))
(person (name Jim) (age 25)))
CLIPS>
```

та визначення наступних правил із символьними обмеженнями, які визначають точну відповідність між полями факту та зразком:

```
(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)).
```

Після введення програми необхідно подати команди `reset` та `run`, після чого будуть активовані та виконані два правила `Find-data` та `Find-Bob-20`.

Прикладом правила `Find-data` є такий запис:

```
(defrule  Find - data
(data ? blue  red $?) =>).
```

У розглянутому прикладі у списку фактів присутні два факти, які підходять заданим шаблоном і здатні активувати конкретне правило:

```
(data 1 blue red)
(data 1 blue red 6.9).
```

Це відбулося тому, що зразки, які задані в лівій частині цих правил, знайшли в списку фактів дані, які повністю відповідають заданим символьним обмеженням.

2) CLIPS надає три *зв'язувальних обмеження,* які призначені для об'єднання окремих обмежень та змінних в одне ціле: обмеження `not`, яке позначається символом «~» (логічне `HI`), обмеження `and`, яке позначається символом «`&`» (логічне `I`) та обмеження `or`, яке позначається символом «`|`» (логічне `АБО`).

Обмеження `not` має найвищий пріоритет, далі йдуть обмеження `and` та обмеження `or`.

Обмеження `not` діє на одне обмеження або змінну, які безпосередньо йдуть за ним. Якщо обмеження, яке йде за обмеженням `not`, зіставляється з полем, то під дією обмеження `not` вдале зіставлення стає невдалим. Якщо обмеження, яке йде за обмеженням `not`, зіставляється з полем, то під дією обмеження `not` невдале зіставлення стає вдалим. Фактично, обмеження `not` заперечує результат використання наступного за ним обмеження.

Приклад 3.20 правила пошуку людей, які не мають каштанового кольору волосся, записаного за допомогою обмеження `not`:

```
(defrule person-without-brown-hair
    (person (name ?name) (hair ~brown))
=>
(printout t ?name "does not have brown hair" crlf)).
```

Обмеження `or` використовується для забезпечення можливості узгодити з полем шаблону одно або декілька допустимих значень. Обмеження `or` задовольняється, якщо будь-яке з двох сусідніх обмежень задовольняються.

Приклад 3.21 правила, яке дозволяє знайти за допомогою обмежувача `or` всіх людей, які мають волосся чорного або каштанового кольору:

```
(defrule black-or-brown-hair
    (person (name ?name)
    (hair brown | black))
=>
    (printout t ?name "has dark hair" crlf)).
```

Обмеження `and` задовольняється, якщо два сусідні обмеження задовольняються, та використовується в сукупності з іншими обмеженнями.

Приклад 3.22 правила, яке використовує і обмеження `or`, і обмеження `and` для визначення людей темного кольору волосся з виведенням на друк отриманої інформації:

```
(defrule black-or-brown-hair
    (person (name ?name)
    (hair ?color&brown | black))
=>
(printout t ?name "has"  ?color "hair" crlf)).
```

У цьому випадку змінна `?color` буде пов'язана з тим значенням кольору, який був зіставлений за допомогою обмеження `brown | black`.

Приклад 3.23 правила, яке використовує і обмеження `not`, і обмеження `and` для визначення людини ні з чорним, ні з каштановим кольором волосся. Цей колір виводиться на друк:

```
(defrule black-or-brown-hair
    (person (name ?name)
    (hair ?color&~brown&~black))
=>
(printout t ?name "has"  ?color "hair" crlf)).
```

3) *Предикатні обмеження поля*, які позначаються символом «:», дозволяють викликати предикатні функції (функції, які повертають значення

FALSE у разі невідповідності умовам та TRUE, якщо значення задовольняє умовам) впродовж процесу зіставлення зразків.

Якщо предикатна функція повертає значення TRUE, то обмеження задовольняється. Якщо предикатна функція повертає значення FALSE, то обмеження не задовольняється. Предикатні обмеження записуються за допомогою двокрапки та виклику відповідної предикатної функції, що йде за ним. Зазвичай *предикатні обмеження* використовуються разом із *зв'язувальними обмеженнями,* та під час зв'язування змінних, тобто якщо є змінна, яку необхідно зв'язати з деяким полем і водночас її протестувати, то необхідно з'єднати її з *предикатним обмеженням*.

Використання *предикатного обмеження поля* аналогічно виконанню перевірки за допомогою умовного елементу test, але в деяких випадках є ефективнішим.

Під час читання шаблон *предикатне обмеження* поля можна розглядати як «такий, що». Наприклад, обмеження поля ?size&:(> ?size 1) можна прочитати як «виконати прив'язку значення ?size такого, що ?size більше 1».

Однією зі сфер застосування предикатного обмеження поля є перевірка наявності помилок у даних.

Приклад 3.24 правила перевірки для визначення того, що елемент даних є числовим, перед його додаванням до проміжної суми:

```
(defrule add-sum
    (data-item ?value&: (numberp ?value))
    ?old-total <- (total ?total)
=>
    (retract ?old-total)
    (assert (total (+ ?total ?value)))).
```

Друге поле першого шаблону можна прочитати як «виконати зв'язування значення ?value такого, що ?value є числом».

Приклад 3.25 декларування правила Find-data з використанням *зв'язувальних обмежень,* та *предикатних обмежень:*

```
(defrule Find-data
(data ?x&: (floatp ?x) &: (> ?x 0) $?y ?z&: (stringp
?z))
=>
(printout t "x=" ?x  "y=" ?y  "z=" ?z  crlf)).
```

У цьому випадку шукається факт неявно створеного шаблону data, перше поле якого є дійсне число більше нуля, а останнє – рядок. У списку

фактів (приклад 3.19) такому правилу задовольняє тільки факт (data 1.0 blue "red").

4) В *обмеженнях, які повертають значення*, можливо використання значень, які повернуті деякими функціями. Виклик функції записується за допомогою знака «=» та вказаної за ним функції. Значення, що повертається функцією, об'єднується зі зразком так, щоб воно було *символьним обмеженням*. Функція обчислюється під час кожного зіставлення зразків, а не один раз під час визначення правила. Правило, наведене в прикладі 3.26, виводить на екран факти data, у яких значення другого поля в два рази більше значення першого, і це факти (data 1 2) та (data 2 4):

```
(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)).
```

5) Зразки можна зіставляти не тільки з фактами зі списку фактів, але й з екземплярами об'єктів, визначених користувачем класів на мові COOL. Такі зразки називають *зразками об'єктів*. Зразки можна зіставляти з об'єктами, специфікація яких визначена до створення зразка і які знаходяться в межах видимості поточного модуля. Будь-який клас, який має об'єкти, що відповідають зразку, не може бути знищено або змінено, доки не буде знищено зразок.

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, який є нащадком класу, який задано в обмеженнях, у випадку, якщо це не буде явно заборонено зразком.

Обмеження name використовується для визначення конкретного об'єкта із заданим ім'ям. Ім'я, яке задається в цьому обмеженні, має бути значенням типу instance-name, а не значенням типу symbol.

Обмеження для складових полів (такі як $?) не можна використовувати з обмеженнями is-a та name.

У прикладі 3.27 наведено 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, які належать різним класам.

Трете правило виконується у випадку, якщо буде знайдено об'єкт активного класу, що вміщує активний слот `width`, значення якого більше 20.

Четверте правило задовольняється будь-яким об'єктом активного класу, який вміщує активні слоти `width` та `height`, значення яких мають бути рівними.

6) Присвоювання змінній адреси конкретного факту або об'єкта називається *адресою зразка* (`pattern-address`).

Синтаксис адреси зразка такий:

<адреса–зразка> `::=` ? <ім'я–змінної > <– <зразок>.

Стрілка вліво «<–» є необхідною частиною синтаксису. Змінна, яка пов'язана з адресою факту або об'єкта, може порівнюватися з іншою змінною або використовуватися зовнішньою функцією.

Правило 3.28 знищення всіх фактів `data`:

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

2. Умовний елемент `test CEs` надає можливість накладення додаткових обмежень на слоти фактів або об'єкти. Цей умовний елемент не потребує зіставлення шаблону з одним із фактів у списку фактів, а просто обчислює значення. Самою зовнішньою функцією цього виразу має бути предикатна функція (за визначенням як предикатна функція може розглядатися будь-яка функція, яка повертає значення `TRUE` або `FALSE`). Якщо обчислення виразу

спричиняє отримання будь-якого значення, відміного від FALSE, то перевірка за допомогою умовного елемента test завершується вдало, інакше – завершується невдало. Запуск правила виконується тільки в тому випадку, якщо перевірка всіх його умовних елементів test, разом з усіма іншими шаблонами завершується вдало.

Приклад 3.29 застосування умовного елемента test:

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

Вираз test обчислюється кожен раз під час задовільнення інших умовних елементів. Це означає, що умовний елемент test буде обчислено більше одного разу, якщо оброблюваний вираз може бути задовільнений більше ніж однією групою даних. Використання умовного елемента test може стати причиною автоматичного додавання правилу деяких умовних виразів.

Приклад 3.30 правила, у якому виконується перевірка для пошуку трьох різних точок:

```
(defrule three-distinct-points
    ?point-1 <- (point (x ?x1) (y ?y1))
    ?point-2 <- (point (x ?x2) (y ?y2))
    ?point-3 <- (point (x ?x3) (y ?y3))
        (test (and (neq ?point-1 ?point-2)
                   (neq ?point-2 ?point-3)
                   (neq ?point-1 ?point-3)))
=>
    (assert (distinct-points (x1 ?x1) (y1 ?y1)
                             (x2 ?x2) (y2 ?y2)
                             (x3 ?x3) (y3 ?y3)))).
```

3. Умовний елемент and CEs.

Якщо все умовні елементи в лівій частині правил CLIPS об’єднані елементом and, то це означає, що всі умовні елементи, задані в лівій частині, повинні задовольнятися, щоб правило було активоване. За своїм призначенням умовний елемент and є протилежним умовному елементу or.

4. Умовний елемент or CEs дозволяє активувати правило будь-яким із декількох заданих умовних елементів. Якщо який-небудь з умовних елементів,

об’єднаних за допомогою or, задоволений, то і весь вираз or вважається задоволеним.

5. Умовний елемент not CEs задовольняється, тільки якщо умовний елемент, який він містить, не задовольняється. Цей умовний елемент може заперечувати тільки один вираз, тому в умовному елементі not використовується умовний елемент and.

Приклад 3.31, у якому необхідно вивести на екран імена всіх пар однолітків, демонструє застосування елемента not:

```
(deftemplate person
        (slot name)
        (slot age))

(deftemplate person-pair
        (slot name1)
        (slot name2)
        (slot age))

(deffacts people
        (person (name Joe) (age 20))
        (person (name Jim) (age 20))
        (person (name Bob) (age 34)))

(defrule Find-2-Coeval-Person
         (person (name ?x) (age ?z)
         (person (name ?y & ~?x) (age ?z)
(not (person-pair (name1 ?x) (name2 ?y) (age ?z)))
(not (person-pair (name1 ?y) (name2 ?x) (age ?z)))
=>
(printout t “name=” ?x “name=” ?y “age=” ?z crlf)
(assert (person-pair (name1 ?x) (name2 ?y) (age ?z)))).
```

Обмеження ?y&~?x забороняє виводити на екран пари імен типу Коля–Коля.

У фактах шаблону person-pair зберігається інформація про вже знайдені пари однолітків.

Умовні елементи (not (person-pair (name1 ?x) (name2 ?y) (age ?z))) та (not (person-pair (name1 ?y) (name2 ?x) (age ?z))) перевіряють наявність фактів типу person-pair та відстежують, чи була вже оброблена ця пара, чи її перестановка. Якщо ці факти відсутні, то це означає, що оброблення не відбулося. У цьому випадку правило активується, і виконуються дії, які описані в правій частині правила:

виводиться повідомлення про знайдену пару однолітків та додається відповідний факт person-pair, який стверджує, що ця пара вже була оброблена.

6. Умовний елемент exists CEs дозволяє визначити, чи існує хоча б один набір даних (фактів або правил), які задовольняють умовним елементам, що задані всередині елемента exists.

CLIPS автоматично замінює exists двома послідовними умовними елементами not.

Приклад 3.32 правила з exists:

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

та перетворене правило з двома not:

```
(defrule example
    (not (not (and (a ?x) (b ?x)))) =>).
```

Приклад 3.33 використання умовного елемента exists, у якому визначається шаблон героя, що має складове поле з ім'ям героя та просте поле, що вміщує статус «не занятий» за замовчуванням. Конструктор deffacts визначає трьох вільних героїв та поточну мету – спасіння планети. Правило перевіряє, чи є наразі ця мета, і чи є вільний герой. Якщо всі умовні елементи правила задовільнені, то правило повідомляє про спасіння планети:

```
(deftemplate hero
        (multislot name)
        (slot status (default unoccupied)))

(deffacts goal-and-heroes
    (goal save-the-world)

    (hero (name Defying Man))
    (hero (name Stupendouse Man))
    (hero (name Incredible Man)))

(defrule save-the-world
    (goal save-the-world)
    (exists (hero (status unoccupied)))
=>
    (printout t "The day is saved." crlf)).
```

Результат виконання прикладу 3.33 використання умовного елемента `exists` наведено далі:

```
Defining deftemplate: hero
Defining deffacts: goal-and-heroes
Defining defrule: save-the-world +j+j+j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
The day is saved.
CLIPS>
```

7. Умовний елемент `forall CEs` дозволяє визначити, що деяка задана умова виконується для всіх заданих умовних елементів. CLIPS автоматично замінює `forall` комбінацією умовних елементів `not` і `and`.

Приклад 3.34 правила з `forall`:

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

та перетворене правило:

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

Приклад 3.35 використовує правило `all-students-passed` для визначення проходження студентами тестів за визначеними дисциплінами за допомогою умовного елемента з `forall`:

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

8. Умовний елемент `logical CEs` надає механізм підтримки вірогідності для створених правилом даних (фактів або об'єктів), які відповідають зразкам. Цей умовний елемент дозволяє вказати, що існування деякого факту залежить від існування іншого факта або групи фактів. Дані, створені в правій частині правила, можуть мати логічну залежність від даних,

які задовольнили зразки в лівій частині правила. Така залежність називається *логічною підтримкою*. Умовний елемент `logical` групує зразки, так само як це робить умовний елемент `and`.

Приклад 3.36 використання умовного елемента `logical`:

```
(defrule ok
         (logical (a))
         (logical (b))
         (c)
=>
         (assert (d))).
```

За командою `run` правило `ok`, яке активоване фактами `a`, `b` і `c`, додає новий факт `d`, який має логічну підтримку (залежить) від фактів `a`, `b` і `c`.

Якщо правило не містить умовних елементів в своїй лівій частині, то до передумов правила автоматично додається зразок `initial-fact`.

Приклад 3.37 роботи правила без умов:

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

та перетворене правило без умов:

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

3.2.4 Команди та функції для роботи з правилами

1. Команда `ppdefrule` (скорочено від `pretty print defrule`) дозволяє переглянути визначення правила в тому вигляді, у якому воно було створено за допомогою конструктора `defrule`:

(`ppdefrule` <ім'я–правила>).

2. Команда `list-defrules` виводить повний список правил, які присутні в CLIPS у цей момент:

(`list-defrules` <ім'я–правил>).

3. Команда `undefrule` призначена для видалення правила:

(`undefrule` <ім'я–правила>).

4. Команди `Lood` і `Lood*` відображають процес завантаження конструктора.

5. Команда `agenda` дозволяє переглянути план виконання завдання, тобто список правил, які знаходяться в робочому списку правил.

6. Команда `matches` виводить інформацію про всі можливі набори даних, які здатні активувати правило.

7. Команда `run` використовується для запуску CLIPS–програм.

3.2.5 Властивості правил

Властивості правил дозволяють задавати характеристики правил до опису лівої частини правила. Для завдання властивості правила використовується ключове слово `declare`. Одне правило повинно мати тільки одне визначення властивості, яке задане за допомогою `declare`:

<визначення – властивості – правила> `::=` (`declare` <властивість – правила>)

< властивості – правила >`::=`(`salience` <цілечислений вираз >)|
(`auto-focus` `TRUE` | `FALSE`).

1. Властивість правила `salience` (значущість) дозволяє користувачеві призначити пріоритет для своїх правил. Оголошений пріоритет може бути виразом, який має цілочисельне значення від –10 000 до +10 000. Вираз, що представляє пріоритет правила, може використовувати глобальні змінні та функції. Якщо пріоритет правила чітко незаданий, йому присвоюється значення за замовчуванням, що дорівнює 0.

Значення пріоритету може бути обчислено в трьох випадках:

– у разі додавання нового правила;

– у разі активації правила;

– на кожному кроці основного циклу виконання правил.

За замовчуванням значення пріоритету обчислюється тільки під час додавання правила. Для зміни цієї установки можна використовувати команду `set-salience-evaluation`.

Кожен метод обчислення пріоритету містить у собі попередній метод, тобто якщо пріоритет обчислюється на кожному кроці основного циклу виконання правил, то він обчислюється і за активації правила, а також за його додавання в систему.

Приклад 3.38 оголошення значень значущості правил для примусового забезпечення запуску правил у послідовності `fire-first`, `fire-second`, незалежно від того, у якому порядку виконувалося введення фактів, які активували ці правила, у список фактів:

```
(defrule fire-first
    (declare (salience 30))
    (priority first)
=>
    (printout t "Print first" crlf))

(defrule fire-second
    (declare (salience 20))
    (priority second)
=>
    (printout t "Print second" crlf)).
```

2. Властивість правила `auto-focus` дозволяє автоматично виконувати команду `focus` за кожної активації правила. Якщо властивість `auto-focus` має значення `TRUE`, то команда `focus` у модулі, у якому визначено це правило, автоматично виконується у разі активації правила.

3.2.6 Стратегії вирішення конфліктів

CLIPS підтримує сім різних стратегій вирішення конфліктів: стратегія глибини (`depth strategy`), стратегія ширини (`breadth strategy`), стратегія спрощення (`simplicity strategy`), стратегія ускладнення (`complexity strategy`), LEX стратегія (`LEX strategy`), MEA стратегія (`MEA strategy`) і випадкова стратегія (`random strategy`). За замовчуванням у CLIPS встановлена стратегія глибини. Поточна стратегія може бути встановлена командою `set-strategy`, яка переупорядковує поточний план розв'язання задачі, базуючись на новій стратегії.

1. Стратегія глибини. Згідно з цією стратегією тільки що активоване правило розміщується вище всіх правил із таким саме пріоритетом.

Наприклад, припустимо, що факт А активував правила 1 та 2 і факт Б активував правило 3 та правило 4. Тоді, якщо факт А доданий перед фактом Б у плані вирішення завдання, то правила 3 та 4 будуть розташовуватися нижче, ніж правила 1 і 2. Однак позиція правила 1 щодо правила 2 і правила 3 щодо правила 4 буде довільною.

2. Стратегія ширини. Згідно з цією стратегією тільки що активоване правило розміщується нижче всіх правил із таким саме пріоритетом.

Наприклад, припустимо, що факт А активував правила 1 та 2 і факт Б активував правила 3 та 4. Тоді, якщо факт А доданий перед фактом Б в плані вирішення завдання, то правила 1 і 2 будуть розташовуватися нижче, ніж правила 3 та 4. Однак позиція правила 1 щодо правила 2 і правила 3 щодо правила 4 буде довільною.

3. Стратегія спрощення. Згідно з цією стратегією між усіма правилами з однаковим пріоритетом правила, які тільки що активовані, розміщуються вище

всіх активованих правил із 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 змінною додає до визначеності одиницю. Кожен виклик функції в лівій частині правила, який є частиною умовних елементів test CEs, також додає до визначеності одиницю. Логічні функції and, or та not не збільшують визначеність правила, але їхні аргументи можуть це зробити. Виклики функцій, зроблені всередині функцій, не збільшують визначеність правила.

Приклад 3.39 правила з визначеністю, що дорівнює 5:

```
(defrule example
 (item ?x ?y ?x)
 (test (and (numberp ?x) (> ?x (+ 10 ?y)) (< ?x
100)))
=> ).
```

4. Стратегія ускладнення. Згідно з цією стратегією між усіма правилами з однаковим пріоритетом правила, які тільки що активовані, розміщуються вище всіх активованих правил із рівною або меншою визначеністю.

5. LEX стратегія. Згідно з цією стратегією для визначення місця активованого правила в плані вирішення завдання використовується «новизна» зразка, який активував правило.

CLIPS маркує кожен факт або об'єкт тимчасовим тегом для відображення відносної новизни кожного факту або об'єкта в системі. Зразки, які асоційовані з кожною активацією правила, сортуються відповідно до зменшення тегів для визначення місця розташування правила.

Активація правила, яка виконана більш новими зразками, розташовується перед активацією, здійсненою більш пізніми зразками. Для визначення порядку розміщення двох активацій правил поодинці порівнюються відсортовані тимчасові теги для цих двох активацій, починаючи з найбільшого тимчасового тега. Порівняння триває поки не залишається одна активація з найбільшим тимчасовим тегом. Ця активація розміщується вище всіх інших у плані вирішення завдання.

Якщо активація деякого правила виконана великою кількістю зразків, ніж активація іншого правила, і всі тимчасові теги, що порівнюються, однакові, то активація іншого правила з великою кількістю тимчасових тегів розміщується перед активацією з меншою кількістю тимчасових тегів. Якщо дві активації мають однакову кількість тимчасових тегів і їхні значення дорівнюють, то правило з більшою визначеністю розміщується перед активацією з меншою визначеністю. Умовний елемент not у CLIPS має псевдотимчасовий тег, який також використовується в цій стратегії вирішення конфліктів. Тимчасовий тег умовного елемента not завжди менше, ніж тимчасовий тег зразка.

У прикладі 3.40 розглянуто наступні шість активацій правил, наведені в LEX–порядку (кома наприкінці рядка активації означає наявність логічного елемента not). Необхідно зазначити, що тимчасові теги фактів не обов'язково дорівнюють індексу, але якщо індекс факту більше, то більше і його тимчасовий тег. Для цього прикладу було прийнято, що тимчасові теги дорівнюють індексам:

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.

Далі показані тіж самі активації з індексами фактів у тому саме порядку, у якому вони порівнюються 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.

6. MEA стратегія. Основна відмінність цієї стратегії від стратегії LEX у тому, що в цій стратегії не проводиться сортування зразків, що активували правило. Як і в стратегії LEX, умовний елемент not має псевдотимчасовий тег.

7. Випадкова стратегія. Згідно з цією стратегією кожній активації призначається випадкове число, яке використовується для визначення місця розташування серед активацій з однаковим пріоритетом.

Це випадкове число зберігається під час зміни стратегій так, що той саме порядок відтворюється за наступної установки випадкової стратегії (серед активацій в плані вирішення завдання, коли стратегія замінена на початкову).

4 ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Для об'єктно-орієнтованого програмування характерні такі основні можливості: абстрактність, інкапсуляція, наслідування, поліформізм, динамічне зв'язування [6].

Абстракція – це спосіб подання даних на певному рівні деякої проблемної області. Створення нового класу реалізує можливість абстрактного подання нового типу даних. Слоти та обробники повідомлень цього класу визначають властивості та поводження цілої групи об'єктів, які належать до цього класу.

Інкапсуляція – це процес, що дозволяє приховувати деталі реалізації об'єкта за допомогою деякого визначеного для цього класу зовнішнього інтерфейсу. Інкапсуляція реалізується в CLIPS вимогою обов'язково використовувати повідомлення під час роботи з об'єктами визначених користувачем класів. Обробники повідомлень класу становлять доступний користувачеві інтерфейс, який приховує реалізацію класу.

Наслідування дозволяє визначати класи, що використовують визначення інших класів і мають усі їхні та деякі свої властивості. COOL підтримує множинне наслідування. Це означає, що деякий клас може мати всі властивості вказаного одного або декількох суперкласів. Об'єкт, який є екземпляром нового класу, наслідує всі властивості (слоти) та поводження (обробник повідомлень) кожного класу зі списку передування класів (`class precedence list`). Список передування класів побудовано з використанням ієрархії наслідування.

Поліформізм – це властивість, завдяки якій різні COOL–об'єкти по-різному реагують на одні й ті саме повідомлення. На практиці це здійснюється приєднанням до різних класів обробників одного й того саме повідомлення, але з різними послідовностями дій, які виконуються.

Динамічне зв'язування є можливістю обирати певний обробник повідомлень об'єкта під час виконання програми. CLIPS підтримує можливість динамічного зв'язування за допомогою функції `send`, яка призначена для посилання повідомлень об'єкту. Виклик цієї функції виконується в процесі роботи програми. Отже, визначення обробника, який виконується в той чи інший момент, також відбувається в процесі виконання програми.

4.1 Класи

Клас становить своєрідний шаблон, який визначає загальні властивості та поводження об'єктів – екземплярів цього класу. Об'єкти можуть належати класам, що визначені користувачем, або деяким системним класам.

Мова COOL надає 17 зумовлених системних класів: `OBJECT`, `USER`, `INITIAL-OBJECT`, `PRIMITIVE`, `NUMBER`, `INTEGER`, `FLOAT`, `INSTANCE`, `INSTANCE-NAME`, `INSTANCE-ADDRESS`, `ADDRESS`, `FACT-ADDRESS`, `EXTERNAL-ADDRESS`, `MULTIFIELD`, `LEXEME`, `SYMBOL`, `STRING`.

Діаграма наслідування цих класів наведена на рисунку 4.1 [1].

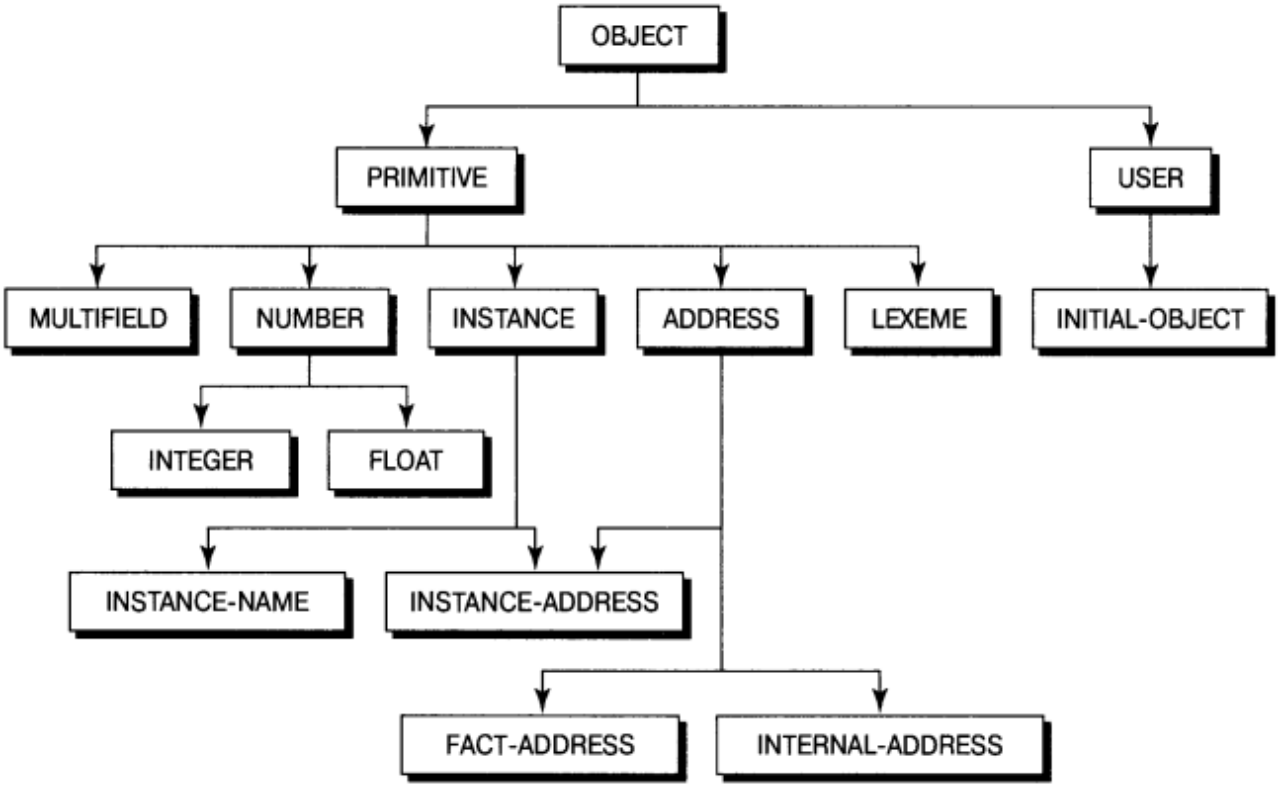

Рисунок 4.1 – Діаграма наслідування системних класів

Усі перераховані системні класи, за винятком `INITIAL-OBJECT`, є абстрактними, що означає їхнє використання тільки для наслідування і не використання для створення екземпляра об'єкта цього класу. Жоден із цих класів не має слотів і, за винятком класу `USER`, жоден не має обробників повідомлень.

Клас `OBJECT` є суперкласом для всіх інших класів разом із класами, визначеними користувачем.

Усі класи, визначені користувачем, повинні наслідуватися прямо чи опосередковано від класу `USER`, оскільки цей клас має всі стандартні системні обробники повідомлень, наприклад, обробники повідомлень для ініціалізації або знищення об'єкта до нього вже приєднані.

У мові COOL екземпляри (або об'єкти) використовуються як ще один спосіб подання даних. Атрибути екземпляра задаються конструктором `defclass`. З використанням конструктора `defmessage-handler` із класами може бути пов'язаний процедурний код, який оформлений у вигляді обробників повідомлень. Механізм наслідування дозволяє використовувати в класі слоти та обробники повідомлень, які пов'язані з іншим класом.

4.1.1 Конструктор defclass

Використання екземплярів та класів замість фактів та конструктора deftemplate надає низку переваг [1]:

– наслідування. Конструктор defclass може наслідувати інформацію від одного або декількох класів, що дозволяє створювати більш структуровані, модульні визначення даних. Усі класи, визначені користувачем, повинні наслідуватися принаймні від одного класу. Для цього COOL надає системні класи для використання як базові в нових класах. Для всіх нових класів COOL до списку суперкласів використовує спеціальний алгоритм отримання списку передування класів. Якщо один клас наслідується від іншого класу, то перший клас називається *підкласом* класу, а другий – *суперкласом* першого. Кожен визначений користувачем клас повинен мати хоча б один прямий суперклас, тобто один клас повинен з'явитися в частині is-a конструктора defclass. Наприклад, запис (defclass A (is-a USER)) ілюструє створення класу A, який наслідує свої властивості безпосередньо від класу USER, до того ж список передування класів для класу A такий: A USER OBJECT;

– за об'єктами можна закріпити процедурну інформацію за допомогою обробників повідомлень. Для створення екземпляра класу використовується функція make-instance;

– зіставлення з шаблоном на підставі об'єктів забезпечує більшу гнучкість ніж зіставлення з шаблоном на підставі фактів.

Для створення користувацького класу в середовищі CLIPS використовується конструктор defclass, який визначає властивості (слоти) та поводження (обробники повідомлень) класу об'єктів. Синтаксис конструктора defclass такий:

```
(defclass <class-name> [коментарі]
       (is-a <superclass-name>)
       <slot-definition>*).
```

Цей конструктор складається з таких елементів:

– ім'я класу;

– список суперкласів, від яких новий клас наслідує слоти та обробники повідомлень. Клас, який визначається користувачем, повинен наслідувати інформацію або від іншого визначеного користувачем класу, або від класу USER. Класом, від якого зрештою наслідують інформацію всі класи, які визначаються користувачем, є системний клас USER;

– список слотів, визначених у новому класі. Слот – це місце для зберігання значень, які асоціюються з об'єктом визначеного користувачем класу. Кожен екземпляр класу має свою копію набору слотів, які визначені в його класі безпосередньо, а також копію будь-яких слотів, які отримані наслідуванням. Іменем слота може бути будь-яке значення типу symbol за

винятком ключових слів `is-a` та `name`, які зарезервовані для використання в зразках (шаблонах) об'єкта.

Ключове слово `is-a` обмежує перелік екземплярів, які узгоджуються з шаблоном, екземплярами тих класів, які задовольняють обмеженню `is-a`.

Ключове слово `name` використовується для узгодження з конкретними екземплярами.

Приклад 4.1 наслідування слотів:

```
(defclass A
 (is-a USER)
            (slot  fooA)
            (slot  barA))

(defclass B
 (is-a A)
            (slot  fooB)
            (slot  fooB)).
```

Список передування для класу A такий: A `USER OBJECT`. Екземпляр класу A буде мати два слоти: `fooA` та `barA`.

Список передування для класу B такий: B A `USER OBJECT`. Екземпляр класу B буде мати чотири слоти: `fooB fooB fooA` та `barA`.

Приклад 4.2 демонструє класи, які наслідують інформацію від іншого класу, що забезпечує спільний доступ до інформації:

```
(defclass PERSON
 (is-a USER)
      (slot full-name)
      (slot age)
      (slot eye-color)
      (slot hair-color))

(defclass EMPLOYER
 (is-a PERSON)
      (slot job-position)
      (slot employer)
      (slot salary))

(defclass STUDENT
 (is-a PERSON)
      (slot university)
      (slot major)
      (slot GPA)).
```

Атрибути класу PERSON наслідуються і в класі EMPLOYER, і в класі STUDENT. Функцією (make-instance [Nike] of STUDENT) можна створити екземпляр Nike, який вміщує тільки слоти, що належать до його класу, у цьому випадку до класу STUDENT, який є підкласом класу PERSON. Функцією send можна викликати обробник повідомлень print, який визначено системою, для роботи з екземпляром класу: (send [Nike] print). Результатом роботи CLIPS будуть слоти university, major, GPA, які характеризують студента Nike.

Слот може вміщувати значення або простого (slot, single-slot), або складового поля (myltislot). Значеннями складового слота можна маніпулювати за допомогою стандартних функцій для складових полів, таких як nth$ та length$, починаючи з моменту присвоєння їм деякого значення. COOL надає також функції для установки складових полів, наприклад, slot–insert$.

Для визначення слотів конструктора defclass можна використовувати такі атрибути слота з конструктора deftemplate: type, range, cardinality, allowed–symbols, allowed–strings, allowed–lexemes, allowed–integers, allowed–floats, allowed–numbers, allowed–instance-names, allowed–values, default, default-dynamic.

Для визначення властивостей слотів використовуються атрибути слотів, що описують різні особливості слота, які притаманні всім об’єктам, що вміщують цей слот. Атрибути слота конструктора defclass називають також *фасетами* (facets або гранями слота). За допомогою граней можна встановити такі властивості слотів: значення за замовчуванням, місце зберігання, доступ, розташування при наслідуванні, джерело граней, активність при зіставленні зразків, видимість для обробників повідомлень підкласу, автоматичне створення обробників повідомлень для доступу до слотів, ім’я повідомлення, що посилається, для установки слота або обмеження на значення слота.

Приклад 4.3 використання атрибутів слотів:

```
(defclass person
  (is-a USER)
  (slot full-name
        (type string))
        (slot age
        (type (integer)
        (range 0 120))
  (slot eye-color
        (type symbol)
        (allowed-values brown blue green)
        (default brown))
  (slot hair-color
```

```
        (type symbol)
        (allowed-values black brown red blonde)
        (default brown)))
```

Грані `default` та `default-dynamic` використовуються для задання початкових значень, які присвоюються слотам під час створення екземпляра класу або його ініціалізації.

За замовчуванням значення грані `default` дорівнює (`default ?DERIVE`). Якщо як вираз використовується значення `?NONE`, то слоту не буде присвоєно значення за замовчуванням.

Атрибут `pattern-match` називається атрибутом активності під час зіставлення зразків і має значення `reactive` та `non-reactive`. Значення `reactive` визначає, що змінення слота активує процес зіставлення зразків (використовується за замовчуванням), а значення `non-reactive` вказує, що змінення слота не спричинить активацію процесу зіставлення зразків.

Для управління доступом до слота можна використовувати атрибути слотів `access` та `create-accessor`. Атрибут `access` безпосередньо обмежує тип доступу, який допускається використовувати до слота. Цьому атрибуту можна присвоїти такі значення:

– значення за замовчуванням `read-write` забезпечує безпосередньо доступ для читання або запису за допомогою обробників класів;

– значення `read-only` виконує вибірку значення слота з використанням скороченного позначення;

– значення `initialize-only` аналогічно використанню атрибута `read-only`, за винятком того, що можливо задавати значення під час створення екземпляра.

Для управління автоматичним створенням обробників `get-` та `put-` для слотів класу використовується атрибут `create-accessor`, якому можна присвоїти такі значення:

– значення `read-write`, яке передбачено за замовчуванням, для створення і обробника `get-`, і обробника `put-`;

– значення `read` для створення тільки обробника `get-`;

– значення `write` для створення тільки обробника `put-`;

– значення `?NONE` не передбачає створення ні обробника `get-`, ні обробника `put-` .

Приклад 4.4 демонстрації цих атрибутів для підрахунку загальної вартості заказу:

```
(defclass ORDER
 (is-a USER)
      (slot ID (access initialize-only)
           (default-dynamic (gensym)))
      (slot total-price (create-accessor read)
           (default 0.0))
```

```
        (slot order-price
              (default 0.0))
        (slot sales-tax (default 0.0))).
```

Значення слота `ID` може бути задано тільки під час створення екземпляра. Якщо для цього слота значення залишиться незаданим, то необхідне значення буде вироблятися динамічно шляхом виклику функції `gensym`. Атрибуту `create-accessor` слота `total-price` присвоюється значення `read`. Отже, обробник `get-total-price` буде створюватися, а обробник `put-total-price` – не буде створюватися.

Загальна вартість обчислюється або задається в коді обробника класу `compute-total-price`, що створюється:

```
(defmessage-handler ORDER compute-total-price()
        (bind ?self:total-price
              (* ?self:order-price
                    (+ 1 ?self:sales-tax)))).
```

CLIPS надає такі команди та функції для роботи з конструктором `defclass`.

1. Команда `describe-class` виводить на екран інформацію про заданий клас.
2. Команда `list-defclass` відображає поточний список конструктора `defclass`.
3. Команда `browse-classes` відображає відносини між класом та його підкласами.
4. Команда `ppdefclass` відображає текстове представлення конструктора `defclass`.
5. Команда `undefclass` використовується для знищення конструктора `defclass`.

4.1.2 Типи класів

Класи розділяються на:

1) абстрактні (`abstract`) та конкретні (`concrete`) класи. Абстактні класи призначені тільки для наслідування, створення екземплярів абстрактних класів неможливе. Конкретні класи можна використовувати як для наслідування, так і для створення екземплярів об’єктів цих класів. За замовчування класи є конкретними. Для вказання того, чи повинен клас бути абстрактним, чи конкретним, використовується атрибут класу `role`, який має бути вказаний після атрибута класу `is-a`, але перед будь-яким визначенням слотів. Синтаксис атрибута `role` такий:

< роль–класу>::= (role concrete | abstract).

Приклад 4.5 демонструє наслідування атрибута ролі класу:

```
(defclass A (is-a USER)
           (role  concrete))
(defclass B (is-a USER))
(defclass C (is-a A B))
(defclass D (is-a B A)).
```

Клас A є конкретним класом, оскільки це чітко задано атрибутом ролі. Клас B є абстрактним, оскільки його перший суперклас USER є абстрактним. Клас C є конкретним класом, тому що його перший суперклас (A) є конкретним класом, а клас D є абстрактним класом, тому що його перший суперклас (B) у списку наслідування є абстрактним;

2) активні та неактивні класи. За допомогою атрибута активності класу можна задати поводження об'єкта цього класу під час проведення зіставлення зразків (шаблонів):

<активність–класу>::=(pattern-match reactive|non-reactive).

Атрибут зіставлення з шаблоном pattern-match дозволяє вказати, що або слот, або клас не може брати участь у зіставленні з шаблоном. Якщо цьому атрибуту присвоюється значення reactive, яке передбачене за замовчуванням, то зазначений слот або клас має властивість активувати зіставлення з шаблоном у лівій частині правила. Якщо цьому атрибуту присвоюється значення non-reactive, то вказаний слот або клас не активує зіставлення з шаблоном у лівій частині правила.

Приклад 4.6 використання атрибута pattern-match на рівні класу:

```
(defclass SUM
  (is-a USER)
  (pattern-match non-reactive)
       (slot total)).
```

Атрибут класу pattern-match необхідно задати після атрибутів класу is-a та role, але до будь-яких визначень слотів.

Атрибут класу pattern-match не має вплив чітко на атрибут слотів pattern-match, тому клас може наслідувати слоти з атрибутом reactive від класу з атрибутом non-reactive.

Об'єкти активного класу можна використовувати для зіставлення зразків під час виконання правил. Об'єкти неактивного класу не можуть використовуватися для зіставлення зразків і не беруть участь у визначенні списків класів, які використовуються для зіставлення зразків.

Приклад 4.7 наслідування атрибута активності класу:

```
(defclass A (is-a USER)
           (role  concrete)
           (pattern-match reactive)
(defclass B (is-a USER)
            (role  concrete))
(defclass C (is-a A B))
(defclass D (is-a B A)).
```

Клас A є активним класом, оскільки це чітко задано атрибутом активності класу. Клас B є неактивним, оскільки він наслідується від абстрактного, а отже, і неактивного класу `USER`. Клас C є активним класом, тому що його перший суперклас (A) є активним класом, а клас D є неактивним класом, тому що його перший суперклас (B) у списку наслідування є неактивним.

4.1.3 Конструктор `defmessage-handler`

За класами можна закріпити не тільки дані, але й процедурну інформацію. Процедури, які входять у склад класів, називаються обробниками повідомлень. Згідно з визначеннями мови COOL до оголошення кожного класу автоматично додаються обробники повідомлень, створені системою: `print`, `delete`, `put-`, `get-`. Користувач має можливість визначити власні обробники повідомлень за допомогою конструктора `defmessage-handler`, який задає поводження об'єкта певного класу у відповідь на отримання визначеного повідомлення. Маніпулювання об'єктом виконується передаванням йому повідомлення за допомогою функції `send`. Конструктор `defmessage-handler` призначено для створення обробника повідомлень, який фактично задає поводження об'єкта цього класу у відповідь на отримання визначеного повідомлення. Результатом передання повідомлення можуть бути або обчисленні значення, або деякі дії обробника.

Конструктор `defmessage-handler` складається з 7 елементів:

– ім'я класу, до якого додається обробник;

– ім'я повідомлення, на яке буде відгукуватися обробник;

– необов'язковий тип обробника повідомлень (за замовчуванням `primary`);

– необов'язкові коментарі;

– список параметрів, які мають бути передані обробнику в повідомленні;

– необов'язковий символ групових параметрів для вказання, що обробник може мати змінне число аргументів;

– послідовність дій, які будуть виконуватися в заданому порядку в момент виклику обробника.

Значення, яке повертається обробником, є результатом обчислення останнього виразу в тілі обробника.

Кожен клас зі списку передування класів об'єкта може мати свої обробники для повідомлень. У цьому випадку об'єкт класу і всі його суперкласи розподіляють роботу із оброблення повідомлень між собою. Кожен обробник обробляє ту частину повідомлення, яка відповідає цьому класу. Обробники повідомлень можуть перекриватися обробниками класів-нащадків чотирма способами: `primary` , `before`, `after`, `around`.

Тип обробника повідомлень задається після імені обробника, до того ж кожен клас може мати обробники кожного типу. Типи обробників повідомлень мають такі функції:

– тип обробника `primary` здійснює основне оброблення повідомлення і використовується як основний обробник, який призначено для формування відповіді на повідомлення;

– тип обробника `before` визначає обробник повідомлень, який має бути викликаний на виконання перед обробником повідомлень `primary`, тобто виконує допоміжне оброблення повідомлення перед викликом основного обробника;

– тип обробника `after` виконує допоміжне оброблення повідомлення після виклику основного обробника;

– тип обробника `around` готує середовище для виконання інших обробників. Обробники повідомлень такого типу називаються також *оболонками*, оскільки вони охоплюють своїм кодом код обробників повідомлень інших типів, які виконуються до та після них. Обробник повідомлень типу `around` становить свого роду комбінацію обробників типу `before` та `after`, яка дозволяє також аварійно перервати оброблення повідомлень у процесі передачі повідомлення.

Приклад 4.8 використання обробника повідомлень типу `before` в класі `MY-ORDER`:

```
(defmessage-handler MY-ORDER get-total-price
       before ()
 (send ?self compute-total-price)).
```

У цьому прикладі насамперед викликається обробник типу `before` класу `MY-ORDER`, а потім обробник типу `primary` класу `ORDER` (див. приклад 4.4).

Приклад 4.9 обробника повідомлень типу `around` для повідомлення `compute-total-price` (див. приклад 4.4):

```
(defmessage-handler MY-ORDER compute-total-price()
            around ()
      (if (or (not (numberp
            (send ?self get-order-price)))
      (not (numberp
            (send ?self get-sales-tax))))
```

```
        then
        (return))
    (call-next-handler)).
```

У цьому обробнику типу `around` насамперед перевіряється, чи є значення слотів `order-price` та `sales-tax` числовими. У разі від'ємного результату цієї перевірки в цьому обробнику виконується повернення і не виконуються будь-які інші дії. У цей момент не викликається жоден із обробників типу `primary`, `before` або `after` для повідомлення `compute-total-price`. Якщо значення обох слотів `order-price` та `sales-tax` є числовими, то викликається функція `call-next-handler` та інші обробники повідомлень. У цьому випадку має бути викликаний тільки обробник повідомлень `compute-total-price` типу `primary` класу `ORDER` (див. приклад 4.4).

Функція `call-next-handler` викликає наступний обробник повідомлень, який був перекритий поточним обробником повідомлень. Під час виклику цієї функції не потрібно задавати будь-які параметри. Наступному обробнику передаються параметри, з якими було викликано поточний обробник.

Обробники повідомлень однозначно ідентифікуються класом, ім'ям та типом і ніколи не викликаються безпосередньо. Коли користувач посилає повідомлення, CLIPS обирає та упорядковує обробники повідомлень, які можна застосувати і які приєднані до об'єкта, та потім виконує їх. Цей процес називається *зв'язуванням повідомлень*.

Приклад 4.10 процесу зв'язування повідомлень:

```
(defclass A (is-a USER)
  (role concrete))

(defmessage-handler A delete before ()
(printout t "Deleting an instance of the class A…"
crlf))

(defmessage-handler USER delete after ()
(printout t "System completed deletion of an
instance." crlf)).
```

Клас A є прямим нащадком класу `USER` і не має слотів. Один обробник повідомлення `delete` типу `before` належить безпосередньо до класу A, а другий обробник повідомлення `delete` типу `after` - до класу `USER`. Крім цих двох обробників повідомлення `delete`, у кожного об'єкта нащадка класу `USER` існує ще один обробник `primary`, який знищує об'єкт.

Спочатку викликається обробник повідомлення `delete` типу `before` класу A, потім системний обробник `primary`, який знищує об'єкт, а після нього обробник повідомлення `delete` типу `after` класу `USER`.

Слотами об'єкта можна маніпулювати за допомогою повідомлень–акцесорів та напряму, минаючи механізм повідомлень, завдяки тому, що обробники повідомлення є частиною об'єкта, який інкапсулює дані та методи їхнього оброблення.

Для доступу до слотів активного екземпляра з дій обробників повідомлень використовується така конструкція:

`?self:`<ім'я–слоту>.

Приклад 4.11 обробника повідомлень, який напряму маніпулює зі слотами об'єкта:

```
(defclass A (is-a USER)
       (role concrete)
       (slot foo (default 1))
       (slot bar (default 2)))

(defmessage-handler A print-all-slots ()
       (printout t ?self:foo " " ?self:bar crlf)).
```

Обробник повідомлень `print-all-slots` виводить на екран поточний зміст слотів `foo` та `bar`, крім того, значення, які зберігаються в цих слотах, передаються напряму, а не за допомогою повідомлень.

Для встановлення нового значення слотів в обробниках використовується функція `bind`:

(`bind ?self:`<ім'я–слота> <значення>*.

Для прикладу 4.11 установка значення слота в обробниках повідомлень класу буде мати вигляд:

```
(defmessage-handler A set-foo (?value)
       (bind ?self:foo ?value)).
```

Приклад 4.12 розроблення обробника повідомлень `compute-area`, який призначено для обчислення площі двох фігур:

```
(defclass RECTANGLE
       (is-a USER)
             (slot height)
             (slot width))
```

```
(defclass CIRCLE
       (is-a USER)
            (slot radius))
(defmessage-handler RECTANGLE compute-area
       ()
       (* (send ?self get-height)
       (send ?self get-width)))
(defmessage-handler CIRCLE compute-area
       ()
       (* (pi)
       (send ?self get-radius)
       (send ?self get- radius)))
(definstances figures
 (rectangle-1 of RECTANGLE (height 2) (width 4))
 (circle-1 of CIRCLE (radius 3)))
```

У цьому прикладі визначаються два класи, RECTANGLE та CIRCLE, з відповідними слотами. За кожним класом закріплено обробник повідомлень compute-area, який призначено для обчислення площі кожної фігури. Для класу RECTANGLE площа екземпляра RECTANGLE обчислюється множенням висоти, яка задана в екземплярі, на ширину. Для класу CIRCLE площа екземпляра CIRCLE є добутоком числа Пі, яке повертається функцією pi, на значення радіуса, що задане в екземплярі, піднесене до ступеня, що дорівнює двум. Розроблені два обробника повідомлень використовують змінну ?self, яка автоматично визначається для кожного обробника повідомлень. Під час виклику обробника повідомлень змінній ?self присвоюється значення адреси того екземпляра, якому передається повідомлення. Ця змінна може використовуватися для передачі в екземпляр повідомлень для виборки значень слотів height, width та radius. Для виборки або установки значення будь-якого слота у вигляді виразу (send ?self get-<slot-name>) можна використовувати вираз ?self:<slot-name>. Тоді описані раніше обробники повідомлень compute-area можна перевизначити у такий спосіб:

```
(defmessage-handler RECTANGLE compute-area
       ()
       (* ?send:height ?send:width))

(defmessage-handler CIRCLE compute-area
       ()
       (* (pi) ?self:radius ?self:radius))
```

Після визначення обробників повідомлень є змога відправляти повідомлення compute-area в екземпляри класів RECTANGLE та CIRCLE

для отримання інформації про площу фігури, яка задана цим екземпляром, наприклад, у такий спосіб: (send [circle-1] compute-area).

У цьому прикладі кожному екземпляру передається теж саме повідомлення, але, обчислюючи площу фігури, яка задана за його допомогою, кожний екземпляр відповідає по-різному. Така здатність різних екземплярів відповідати на те саме повідомлення в характерній для нього формі, називається *поліморфізмом*.

Команда override-next-handler надає можливість перекривати параметри, які передаються обробнику повідомлень і має такий синтаксис:

(override-next-handler <expression>*).

У цьому визначенні кожен вираз <expression> становить параметр, який передається як заміна існуючого параметра в наступний обробник повідомлень.

Приклад 4.13 використання команди override-next-handler для оброблення заказу на товари в іноземній валюті:

```
(defclass FOREIGN-ORDER
 (is-a ORDER)
       (slot exchange-rate (default 1.0)))
(defmessage-handler FOREIGN-ORDER get-order-price
            around()
       (* ?self:exchange-rate (call-next-handler)))
(defmessage-handler FOREIGN-ORDER put-order-price
            around(?value)
       (override-next-handler
(/ ?value ?self:exchange-rate)).
```

У цьому прикладі створено конструктор defclass з ім'ям FOREIGN-ORDER, і цей клас наслідує свої властивості від класу ORDER (див. приклад 4.4) та забезпечує автоматичне перетворення вартостей, заданих в іноземній валюті та доларах США. Клас FOREIGN-ORDER має слот exchange-rate, який використовується для представлення валютного курсу, який зв'язує іноземну валюту і долари США.

В обробнику get-order-price типу around класу FOREIGN-ORDER викликається обробник get-order-price типу primary класу ORDER за допомогою команди override-next-handler. Потім значення, що повертається, помножується на значення слота exchange-rate та виконується повернення отриманого значення. В обробнику put-order-price типу around класу FOREIGN-ORDER викликається обробник put-order-price типу primary класу ORDER за допомогою команди override-next-handler, але замість передачі значення, яке відповідає

вартості в іноземній валюті, це значення ділиться на валютний курс і замість нього передається отримане значення, яке представлене в доларах США.

Для прикладу можна створити екземпляр класу [order6] із заданими значеннями слотів командою:

```
(make-instance order6 of FOREIGN-ORDER
  (ID #006)
  (exchange-rate 2)
  (sales-tax .10)).
```

Командою (send [order6] get-order-price) виконується операція виборки значення слота, а команда (send [order6] put-order-price 40.00) дозволяє змінити значення слота введенням нового значення в іноземній валюті.

4.1.4 Системні обробники повідомлень

CLIPS підтримує поняття *daemons* (демони) – це ділянки програмного коду, які неявно виконуються у разі виникнення деякого зумовленого системного повідомлення. На практиці демони реалізуються за допомогою обробників системних повідомлень before або after, які добавлені користувачем.

Системний клас USER вміщує 7 зумовлених обробників повідомлень: init, delete, print, direct-modify, message-modify, direct-duplicate, message-duplicate. Усі класи, похідні від класу USER, наслідують ці обробники повідомлень. Зумовлені обробники повідомлень не можна знищити або змінити [6].

1. Ініціалізація об’єкта. Під час створення будь-який об’єкт, який є нащадком класу USER, отримує повідомлення init, яке обробляється зумовленим системним обробником. Систаксис системного обробника init такий:

```
(defmessage-handler USER init primary()).
```

Цей обробник відповідає за ініціалізацію об’єкта значенням за замовчуванням відразу після його створення. Повідомлення init об’єкту посилають функції make-instance та initialize-instance. Користувач не повинен самостійно надсилати це повідомлення. Обробник цього повідомлення використовує функцію init-slot.

Приклад 4.14 створення демона системного обробника init:

```
(defclass A (is-a USER)
       (role concrete))
       (defmessage-handler A init before ()
```

```
(printout t "Initializing a new instance of class A..." crlf)).
```

До класу A приєднано демон–обробник before, який виконується завжди перед ініціалізацією об'єкта, тобто отримання їм системного повідомлення init.

2. Знищення об'єкта. Під час знищення об'єкт класу, який є нащадком системного класу USER, отримує повідомлення delete, яке обробляється зумовленим системним обробником. Синтаксис системного обробника delete такий:

```
(defmessage-handler USER delete primary()).
```

Цей обробник відповідає за знищення об'єкта з системи. Користувач повинен самостійно надіслати повідомлення delete об'єкту, який він хоче знищити. Обробник повертає значення TRUE, якщо об'єкт знищено, інакше – значення FALSE.

3. Відображення об'єкту. Для відображення змісту слотів об'єкта призначено повідомлення print, яке також має зумовлений системний обробник. Синтаксис системного обробника print такий:

```
(defmessage-handler USER print primary()).
```

Цей обробник виводить назву об'єкта, його клас та поточне значення всіх слотів.

4. Змінення об'єкта. CLIPS надає два зумовлених системних обробника для змінення значень слотів об'єкта: direct-modify та message-modify. Синтаксис системних обробників direct-modify та message-modify такий:

```
(defmessage-handler USER direct-modify primary
(?slot-override-expressions))

(defmessage-handler USER message-modify primary
(?slot-override-expressions)).
```

Обробник direct-modify дозволяє змінювати значення слотів об'єкта без використання повідомлення put-. Цей обробник використовується функціями modify-instance та active- modify-instance.

Для кожного слота, який визначено в конструкторі defclass, система CLIPS автоматично визначає обробники повідомлень слота з префіксами get- та put-, які використовуються для виборки та задання значень слота. Дійсні імена обробників повідомлень формуються в результаті додавання до цих префіксів імені слота. Наприклад, якщо є слот age в конструкторі defclass з

ім’ям person, то автоматично створюються для цього класу такі обробники повідомлень: get-age та put-age. Обробники повідомлень get-не мають параметрів та повертають значення слота.

Обробник message-modify дозволяє змінювати значення слотів об’єкта за допомогою посилання повідомлень put- для кожного слота. Він використовується функціями modify-instance та active-modify-instance.

5. Копіювання об’єкта. CLIPS надає два зумовлених системних обробника для створення копій об’єкта: direct-duplicate та message-duplicate. Синтаксис системних обробників direct-modify та message-modify такий:

```
(defmessage-handler USER direct-duplicate primary
(?new-instance-name ?slot-override-expressions))

(defmessage-handler USER message-duplicate primary
(?new-instance-name ?slot-override-expressions)).
```

Обробник direct-duplicate копіює об’єкт без використання повідомлення put- для установки значень заданих слотів. Значення слота з первинного об’єкта безпосередньо копіюється в задані слоти нового об’єкта. Якщо ім’я нового об’єкта співпадає з вже існуючим іменем об’єкта, то існуючий об’єкт знищується без використання повідомлення. Це повідомлення використовується функціями duplicate-instance та active-duplicate-instance.

Обробник message-duplicate виконує копіювання об’єкта з використанням повідомлення put-. Значення слота з первинного об’єкта безпосередньо копіюється в задані слоти нового об’єкта з використанням повідомлень put- та get-. Якщо ім’я нового об’єкта співпадає з вже існуючим іменем об’єкта, то існуючий об’єкт знищується за допомогою повідомлення delete. Після створення нового об’єкта йому надсилається повідомлення init. Це повідомлення використовується функціями message-duplicate-instance та active-message-duplicate-instance.

CLIPS надає такі команди та функції для роботи з конструктором defmessage-handler.

1. Команда list-defmessage-handler відображає поточний список конструктора defmessage-handler, у якій можна використовувати ключеве слово inherit, яке вказує, що в список мають бути включені успадковані обробники повідомлень.

2. Команда ppdefmessage-handler відображає текстове подання конструктора defmessage-handler.

3. Команда undefmessage-handler знищує конструктор defmessage-handler.

4. Функція `get-defmessage-handler` повертає багатозначне значення, яке вміщує список конструктора `defmessage-handler`, що належить до вказаного класу.

5. Команда `watch` з параметрами `messages` або `message-handlers` використовується для відстеження процесу передачі повідомлень та функціонування обробника повідомлень.

4.2 Об'єкти

Об'єкти дозволяють об'єднати дані за способом їхнього оброблення. Об'єкти CLIPS можна використовувати в правилах і функціях як дані так само, як факти або змінні [6].

У CLIPS об'єктами є тільки об'єкти класів, визначених користувачем, та об'єкти, які представляють дані примітивних типів CLIPS, на відміну від інших об'єктно-орієнтованих мов, у яких об'єктами є абсолютно всі програмні елементи.

Об'єкт CLIPS складається з двох основних частин: властивості об'єкта та його поводження. Властивості об'єкта визначаються в термінах слотів, а поводження обумовлюється обробниками повідомлень, які є приєднаною до класу послідовністю дій із заданим ім'ям. Будь-які маніпуляції з об'єктом можна виконати тільки за допомогою повідомлень. Наприклад, у випадку об'єкта Rolls-Royce, який є екземпляром класу CAR, для того щоб запустити двигун, користувач повинен надіслати об'єкту повідомлення start–engine за допомогою функції `send`. Те, як об'єкт Rolls-Royce прореагує на це повідомлення, залежить від визначення обробника повідомлень start–engine, який пов'язано з класом CAR. Призначення обробника повідомлень насправді еквівалентно призначенню будь-якої функції – повертати результат або виконувати дії.

Клас становить своєрідний шаблон, який визначає загальні властивості та поводження об'єктів – екземплярів класу. Об'єкти можуть належати класам, які визначені користувачем, або деяким системним класам.

Об'єкти поділяються на дві категорії: *об'єкти, які зберігають примітивні типи даних,* і *об'єкти класів, визначених користувачем*.

Об'єкти примітивних типів неявно створюються та знищуються системою CLIPS у місцях, де це потрібно. За посиланням на такий об'єкт можна отримати значення відповідного типу, які зберігаються в ньому. Об'єкти примітивних типів не мають слотів та імен. Класи, які визначають ці об'єкти, є зумовленими класами CLIPS. Функціональність зумовлених класів, які визначають об'єкти примитивних типів, подібна функціональності класів, визначених користувачем, за винятком того, що до таких класів не можна приєднати обробник повідомлень. Це робить не дуже зручним використання таких класів в об'єктно-орієнтованому програмуванні. Основним призначенням об'єктів примітивних типів є використання їх у визначенні родових функцій,

які використовують ці об'єкти як свої аргументи та, за заданим набором, у момент виклику, визначають, який метод родової функції має бути викликаний.

Для посилання на *об'єкт класу, визначеного користувачем*, необхідно використати ім'я або адресу об'єкта. Такі об'єкти створюються або знищуються за допомогою повідомлень або спеціальних системних функцій. Властивості об'єкта класу, визначеного користувачем, визначаються набором слотів, які задані під час визначення класу. Слот об'єкта має ім'я та може вміщувати просте або складове значення. Усі об'єкти одного класу мають однакові набори слотів, але можуть вміщувати в цих слотах різні значення.

Основна різниця між слотами об'єкта та неупорядкованого факту полягає в можливості наслідування, яке дозволяє використовувати в класі деякі властивості та поводження, які визначені в іншому класі. Об'єктно-орієнтована частина мови CLIPS (COOL) підтримує множинне наслідування, за якого клас може наслідувати слоти та обробники повідомлень безпосередньо від декількох класів.

Робота з програмними елементами CLIPS, які не є об'єктами, здійснюється за допомогою викликів відповідних функцій.

Робота з об'єктами в CLIPS, який реалізує концепцію інкапсуляції даних, здійснюється за допомогою посилання їм повідомлень.

Для цього існує системна функція `send`, яка вміщує як параметр об'єкт призначення для повідомлення, само повідомлення та будь-які аргументи, які передаються обробнику. Синтаксис функції `send` такий:

(`send` < об'єкт > <ім'я–повідомлення> [< аргументи>].

4.2.1 Створення об'єкта

Для створення та ініціалізації нового об'єкта використовується функція `make-instance`, яка неявно надсилає ініціалізовані повідомлення кожному новому об'єкту після його розміщення в пам'яті. Ця функція також перевизначає значення слотів для зміни будь-якого значення за замовчуванням. Вона автоматично призупиняє процес зіставлення зразків у правилах для всіх автивних об'єктів доти, поки процес створення об'єкта не буде завершено.

Функція `active-make-instance` також дозволяє створювати нові об'єкти визначених користувачем класів, але не викликає затримки процесу зіставлення зразків.

У разі успіху функція `make-instance` повертає ім'я створеного об'єкта, інакше – значення `FALSE`.

Під час виконання функція `make-instance` здійснює такі дії:

1) створює новий неініціалізований об'єкт заданого класу із заданим ім'ям;

2) значення всіх слотів, які перевизначаються, обчислюються та встановлюються за допомогою повідомлення `put-` , тобто:

(send <ім’я–об’єкта> put-<ім’я–слота> <вираз>*;

3) новий об’єкт отримує повідомлення:

init: (send <ім’я–об’єкта> init).

Зумовлений системний обробник цього повідомлення викличе функцію init-slot, яка використовує значення за замовчуванням із визначення класу для всіх слотів, які не були перевизначені. Установка слотів класу за замовчуванням розміщується безпосередньо в слот без використання повідомлень.

4.2.2 Конструктор definstances

Для ініціалізації об’єктів використовується конструктор definstances, який дозволяє створювати набір об’єктів, що додають у базу знань CLIPS під час кожного очищення системи. Цей конструктор є еквівалентним конструктору deffacts, але є таким, що використовується до екземплярів. Під час виконання команди reset система CLIPS очищується, тобто всім екземплярам передається повідомлення delete, а потім у список додаються всі об’єкти, які задані конструкторами definstances. CLIPS вміщує один зумовлений системний конструктор definstances, який викликає додавання в систему об’єкта initial-object.

Зазвичай для створення об’єктів використовують функцію make-instance.

Приклад 4.15 використання конструктора definstances:

```
(definstances  people
       (Nike of PERSON (full-name “Nike Petrov”)
            (age 23))
       (of PERSON (full-name “John Doe”)
            (hair-color black))).
```

Приклад 4.16 синтаксису зумовленого класу та екземпляр цього класу:

```
(defclass INITIAL-OBJECT
       (is-a USER)
       (role concrete)
       (pattern-match reactive))
(definstances  initial-object
       (initial-object of INITIAL-OBJECT)).
```

Клас INITIAL-OBJECT є зумовленим класом, прямим нащадком класу USER. Клас INITIAL-OBJECT не може бути знищено, проте об'єкт цього класу initial-object можна знищити.

Зразки можна зіставляти з екземплярами об'єктів, так саме як і зразки зіставляються з фактами зі списку фактів (см п. 3.2.3). Синтаксис зразків об'єктів є такий:

<зразок об'єкта> ::= (object <атрибути–обмеження>)
<атрибути–обмеження> ::= (is-a <обмеження>)|
(name <обмеження>)|
(slot <обмеження>).

Обмеження is-a (є) використовується для визначення таких обмежень класу, як «Чи є цей об'єкт екземпляром заданого класу?». Обмеження name використовується для визначення конкретного об'єкта із заданим ім'ям.

Для маніпулювання конструкторами definstances передбачені такі команди та функції.

1. Команда list-definstances відображає поточний список конструктора definstances.

2. Команда ppdefdefinstances відображає текстове подання конструктора definstances.

3. Команда undefinstances використовується для знищення конструктора definstances.

4. Функція get-definstances-list повертає багатозначне значення, яке вміщує список конструкторів definstances.

4.2.3 Переініціалізація існуючих об'єктів

Для читання, установки значень слотів та знищення об'єкта використовуються такі функції:

1. Функція initialize-instance надає можливість повторної ініціалізації існуючих об'єктів значеннями за замовчуванням, що задані в визначенні класу, або новими перевизначеннями слотів. Значення функції, яке повертається, – це ім'я об'єкта в разі успішного виконання операції або значення FALSE у випадку помилки.

Синтаксис функції initialize-instance такий:

(initialize-instance <об'єкт> <перевизначення–слота>*).

Параметр <об'єкт> має бути ім'ям об'єкта, адресою об'єкта або рядком. Функція initialize-instance автоматично призупиняє процес зіставлення всіх активних об'єктів доти, поки операція не буде завершена.

Під час виконання функція initialize-instance здійснює такі дії:

а) обчислюються всі задані перевизначення слотів, після чого результати розміщуються у відповідні слоти за допомогою повідомлення:

(send <об'єкт> put-<ім'я–слота> <значення>);

б) зумовлений системний обробник цього повідомлення викликає функцію init-slot. Ця функція використовує за замовчуванням значення із визначення класу для всіх слотів, які не були перевизначені. Установки слотів класу за замовчуванням розміщюються в слот без використання повідомлень.

2. Обробники повідомлень для читання змісту слотів можуть використовувати як повідомлення, так і прямий доступ до слотів об'єкта. CLIPS надає декілька функцій, які також можуть неявно оперувати з об'єктом за допомогою повідомлень і можуть бути викликані тільки обробниками, наприклад, такими як dynamic-get.

3. Для установки значень слотів у CLIPS використовуються обробники повідомлень, які використовують як повідомлення, так і прямий доступ до слотів об'єкта. У випадку прямого доступу до слотів для установки нового значення використовується функція bind.

4. Для знищення об'єкта з системи використовується повідомлення delete.

4.2.4 Змінення та дублювання об'єктів

У мові CLIPS передбачено декілька команд для роботи з екземплярами, які надають функціональні можливості для модифікації та дублювання.

Для змінення об'єктів призначені 4 функції: modify-instance, active-modify-instance, message-modify-instance, active-message-modify-instance. Перші дві функції дозволяють обновляти слоти об'єктів без виклику повідомлень put-, дві інші використовують повідомлення. Кожна з цих функцій повертає значення TRUE у разі успіху та FALSE – в іншому випадку.

У разі використання команд active-modify-instance та active-message-modify-instance зіставлення з шаблоном об'єкта виконується тільки після того, як будуть внесені зміни в кожен слот, тобто всі модифікації слотів в екземплярі були виконані до зіставлення з шаблоном об'єкта.

Приклад 4.17 модифікації екземпляра командою modify-instance. Спочатку створити клас з ім'ям PERSON:

```
(defclass PERSON
 (is-a USER)
    (slot first-name)
    (slot last-name)).
```

Далі створити екземпляр [p1] класу PERSON функцією make-instance:

```
(make-instance [p1] of PERSON
     (first-name "Nike")
     (last-name "Kharkiv")).
```

Змінити значення екземпляра [p1] класу PERSON функцією modify-instance:

```
(modify-instance  [p1]
     (first-name "Mish")
     (last-name "Lviv")).
```

Приклад 4.18 використання функції message-modify-instance, яка має такий саме синтаксис, як і функції modify-instance, але у разі її використання змінення значень слотів виконується за допомогою засобів передачі повідомлень:

```
(message-modify-instance[p1]
     (first-name "Nike")
     (last-name "Kharkiv")).
```

Для дублювання об'єктів призначені 4 функції: duplicate-instance, active-duplicate-instance, message-duplicate-instance, active-message-duplicate-instance. Ці функції дозволяють дублювати об'єкти та обновлювати слоти об'єктів без виклику повідомлень put-. Кожна з цих функцій повертає ім'я нового об'єкта у разі успіху (значення TRUE) та значення FALSE – в іншому випадку.

За допомогою цих функцій замість модифікації екземпляра, який передано як параметр, створюється дублікат екземпляра, до якого використовуються операції перекриття значень слотів. Значенням, яке повертається цими функціями, є дубльований екземпляр.

Приклад 4.19 використання команд дублювання:

```
(duplicate-instance [p1]
(first-name ""Mish"))

(message-duplicate-instance [p1] to [p3]
(first-name ""Kate")).
```

4.3 Модулі

CLIPS надає можливість розбиття бази знань на окремі незалежні модулі. Для створення таких модулів існує конструктор `defmodule`. За допомогою модулів можна групувати разом окремі елементи бази знань і керувати процесом доступу до цих елементів під час розв'язання певної задачі. Області видимості в CLIPS строго ієрархічні й однонаправлені: якщо модуль А може бачити дані модуля В, це не означає, що модуль В може бачити дані модуля А.

За допомогою управління з обмеженням доступу до даних, що містяться в різних модулях, під час вирішення складних завдань модулі можуть реалізовувати концепцію дошки оголошень (`blackboard strategy` – стратегія вирішення завдань із використанням різнорідних джерел знань, які взаємодіють через загальне інформаційне поле). У цьому випадку окремий модуль дозволяє правилам з інших модулів бачити строго певний набір фактів і об'єктів. Крім того, модулі використовуються для управління потоком обчислення правил [6].

За замовчуванням у CLIPS визначається єдиний модуль `MAIN`.

4.3.1 Створення модуля

Для створення модуля використовується конструктор `defmodule`, який має такий синтаксис:

```
(defmodule   <ім'я–модуля>
             [<коментарі>]
   <специфікації–імпорту–експорту>*)
             < специфікації–імпорту–експорту> ::=
             (export <елемент–специфікації>)  |
             (import <ім'я–модуля> <елемент–специфікації>)
   <елемент– специфікації > : := ?ALL|
                                 ?NONE|
                                 <конструктор> ?ALL|
                                 <конструктор> ?NONE|
                                 <конструктор> <ім'я–конструктора>
    <конструкція>::=deftemplate|defclass|
                    defglobal|deffunction|defgeneric.
```

Явне завдання модуля виконується за допомогою імені модуля, розділеного з ім'ям конструкції за допомогою символу подвійної двокрапки «`::`». Ім'я модуля і символ «`::`» називаються специфікатором модуля (`module specifier`).

Неявне завдання модуля виконується за допомогою установки поточного активного модуля. Поточний модуль змінюється під час кожного визначення нового модуля або під час виконання функції `set-current-module`.

Після створення модуль не може бути перевизначений або видалений (за винятком системного модуля MAIN, який користувач може один раз перевизначити). Єдиний спосіб видалити існуючий модуль – виконати команду clear. Під час запуску системи і виклику команди clear CLIPS автоматично створює визначений системний модуль (defmodule MAIN).

Усі зумовлені системні класи належать системному модулю MAIN, який є поточним за замовчуванням.

Для визначення, у який модуль буде розміщена конструкція мови, створена відповідним конструктором, у конструкторі необхідно вказати ім'я модуля. Конструктори deffacts, deftemplate, defrule, deffunction, defgeneric, defclass, definstances для визначення імені модуля дозволяють включати його в ім'я відповідної конструкції.

Конструктор defglobal приймає ім'я модуля в спеціально відведене для цього поле, яке йде за ключовим словом defglobal.

Конструктор defmessage-handler приймає ім'я модуля як частину визначення класу, з яким зв'язується повідомлення.

Конструктор defmethod приймає ім'я модуля як частину визначення родової функції, якій належить цей метод.

Ім'я поточного модуля можна отримати за допомогою функції get-current-module.

У синтаксисі конструктора defmodule передбачені ключеві слова, які визначають тип специфікації експорту/імпорту.

Під час використання формату атрибута export ?ALL у модуль експортуються всі видимі в ньому конструкції, а формат атрибуту export ?NONE показує, що ніякі конструкції не експортуються, і цей формат є форматом за замовчуванням для конструктора defmodule.

Атрибут export може бути заданим більш ніж один раз у визначенні defmodule для вказання різних типів конструкцій, що експортуються, наприклад, конструкції конструктора deftemplate або інші конструкції процедурного й об'єктно-орієнтованого програмування мови CLIPS.

Атрибут import має формати (import <ім'я–модуля>?ALL) та (import <ім'я–модуля>?NONE), які забезпечують імпорт зазначених конструкцій з вказанням модуля, з якого імпортуються конструкції.

У прикладі 4.20 наведено використання модулів:

```
(clear)
(defmodule A)
(defmodule B)
(defrule foo =>)
(defrule A :: bar =>)
       (list-defrules)
       (set-current-module B)
       (list-defrules).
```

Після визначення нового модуля він стає поточним. Відповідно, правило `foo` було додано в поточний модуль B, оскільки під час його створення модуль не було вказано явно, а правило `bar` додане в модуль A, що явно вказано в конструкторі.

Повідомлення, які виникли після визначення правил, повідомляють про визначення в нових модулях фактів `initial-fact`, які необхідні для безумовних правил. Після цього, перемикаючи поточний активний модуль за допомогою команди `set-current-module` та використовуючи команду `list-defrules`, можна впевнитися, що правила знаходяться в тих модулях, у яких вони мають знаходитися.

У прикладі 4.21 модуль A експортує всі видимі в ньому конструкції, модуль B – усі конструкції `deftemplate`, а модуль C – три окремі конструкції `defglobal`:

```
(defmodule A (export ?ALL))
(defmodule B (export deftemplate ?ALL))
(defmodule C (export defglobal foo bar abc)).
```

Специфікація імпорту у визначені модуля використовується для визначення, які конструкції з інших модулів можна використовувати в цьому модулі. Імпортуватися можуть лише такі конструкції: `deftemplate`, `defclass`, `deffunction`, `defgeneric`, `defglobal`.

У прикладі 4.22 модуль A імпортує всі видимі конструкції модуля D, модуль B – усі конструкції `deftemplate` з модуля D, а модуль C – три окремі конструкції `defglobal`, які також визначені в модулі D:

```
(defmodule A (import D ?ALL))
(defmodule B (import D deftemplate ?ALL))
(defmodule C (import D defglobal foo bar abc)).
```

Під час використання ключового слова `?NONE` модуль не буде імпортувати/експортувати або ніякі конструкції, або не буде імпортувати/експортувати ніякі конструкції заданого типу.

У CLIPS факти й об’єкти належать не тому модулю, у якому вони були створені, а тому, у якому було визначено відповідний конструктор `deftemplate` або `defclass`. Отже, факти й об’єкти стають видимими в тих модулях, які імпортують відповідні конструкції `deftemplate` або `defclass`. Це дозволяє розбивати базу знань так, щоб правила або інші конструкції могли бачити тільки необхідні їм факти й об’єкти.

4.3.2 Модулі та управління виконанням правил

Кожен модуль має свій власний процес зіставлення зразків для своїх правил і свій план вирішення завдання. За командою `(run)` починає

виконуватися план вирішення завдання модуля, на який наразі встановлено фокус. Команди (reset) і (clear) автоматично встановлюють фокус на модуль MAIN.

Приклад 4.23 управління виконанням програми та прийняття рішення про те, у якому модулі необхідно обрати робочий список правил для їхнього виконання [1]. На першому кроці необхідно надати визначення для модулів DETECTION, ISOLATION, RECOVERY та їхніх конструкторів deftemplate:

```
(defmodule DETECTION
       (export deftemplate fault))
(deftemplate DETECTION::fault
       (slot component))
(defmodule ISOLATION
       (export deftemplate possible-failure))
(deftemplate ISOLATION::possible-failure
       (slot component))
(defmodule RECOVERY
       (import DETECTION deftemplate fault)
(import ISOLATION deftemplate possible-failure)).
```

На другому кроці необхідно ввести факти fault у списки фактів модулів DETECTION та RECOVERY, а факти possible-failure – у списки фактів модулів RECOVERY та ISOLATION:

```
CLIPS> (deffacts DETECTION::start
       (fault (component A)))
CLIPS> (deffacts ISOLATION::start
       (possible-failure (component B)))
CLIPS> (deffacts RECOVERY::start
       (fault (component C))
       (possible-failure (component D))).
```

Після вставлення фактів fault у список фактів модулів DETECTION та RECOVERY ці факти є видимими в обох модулях. Аналогічно факти possible-failure, які вставляються в модулі ISOLATION та RECOVERY, є видимими в обох модулях.

На третьому кроці записуються правила. Усі конструктори defrule мають бути активовані за допомогою фактів fault та possible-failure, які введені в список фактів:

```
(defrule DETECTION::rule-1
       (fault (component A|C))
=>)
(defrule ISOLATION::rule-2
```

```
        (possible-failure (component B|D))
=>)
(defrule RECOVERY::rule-3
        (fault (component A|C))
        (possible-failure (component B|D))
=>).
```

Після подачі команди `agenda` відтвориться робочий список правил модуля `RECOVERY`, оскільки останне задане правило було поміщено в цей модуль.

План виконання завдання – це список усіх правил, що мають задовольнити умовам за деякого, поточного стану списку фактів і об'єктів, які ще не були виконані. Виконання плану подібно стеку (верхнє правило плану завжди буде виконано першим). Після активації нового правила воно розміщується в плані вирішення завдання, керуючись такими факторами:

1) тільки активоване правило розміщується вище всіх правил із меншим пріоритетом і нижче всіх правил з більшим пріоритетом;

2) серед правил з однаковим пріоритетом використовується поточна стратегія вирішення конфліктів для визначення розміщення серед інших правил з однаковим пріоритетом;

3) якщо правило активовано разом із кількома іншими правилами додаванням або вилученням деякого факту і за допомогою кроків 1) і 2) не можна визначити порядок правила в плані вирішення завдання, то правило у довільний спосіб упорядковується разом з іншими правилами, які були активовані.

Необхідно зазначити, що в цьому випадку порядок, у якому правила були додані в систему, справляє довільний ефект на вирішення конфлікту, який великою мірою залежить від поточної реалізації правил.

Виконання правил триває, поки в плані вирішення завдання не залишиться правил, які можна застосувати, а інший модуль не отримає фокус, або права частина одного з правил, що виконуються, не викличе функцію (`return`). Після того як у плані вирішення завдання модуля, що має фокус, закінчуються правила, поточний модуль видаляється зі стеку фокусів (`focus stack`) і наступний модуль, що знаходиться в стеку, отримує фокус. Перед виконанням правила поточним стає модуль, у якому це правило визначено. Керувати стеком фокусів можна за допомогою команди (`focus` <ім'я–модуля>+).

Поточний стан стеку фокусів можна продивитися за допомогою функції `get-focus-stack` або команди `list-focus-stack`. Команда `clear-focus-stack` знищує всі модулі зі стеку фокусів. Функція `get-focus` повертає ім'я модуля, який знаходиться в поточному фокусі, або значення `FALSE`, якщо стек фокусів є пустим, а функція `pop-focus` знищує поточний фокус зі стеку фокусів та повертає ім'я модуля, який був у поточному фокусі, або значення `FALSE`, якщо стек фокусів є пустим.

5 ПРАКТИЧНІ ЗАНЯТТЯ

ПРАКТИЧНЕ ЗАНЯТТЯ 1
РОБОТА З ПРОСТОЮ БАЗОЮ ЗНАНЬ

Мета заняття – ознайомлення зі створенням бази даних та бази знань із використанням конструкторів.

Завдання до виконання практичного заняття

1. Розробити базу знань, яка складається з двох програмних модулів, для підрахунку кількості учасників конференції, які прибули з різних міст країни.
2. Підібрати резистор для ділянки електричної схеми радіоелектронного пристрою.

Загальні положення

Для функціонування експертної системи важливим є наявність бази знань. Останнім часом усе частіше термін «система, яка базується на знаннях» (knowledge–base system) використовується як синонім терміна «експертна система» [8].

Знання з предметної області називається базою знань. База знань експертної системи містить факти (дані) і правила (способи подання знань). Механізм виводу містить: інтерпретатор, який визначає, як застосовувати правила для виводу нових знань, та диспетчерів, що встановлюють порядок застосування цих правил. Знання або, за термінологією спеціалістів, база знань експертної системи складається з великої кількості професійних правил різного ступеня спільності. Розв'язуючи задачу, експертна система вибирає правила в порядку зниження їхньої спільності, що відтворює алгоритм міркувань спеціаліста у подібній ситуації від мети до конкретних дій [9].

У CLIPS база даних може представляти деяку предметну область. Вихідний або поточний стан будь-якої проблеми можна моделювати в просторі або в часі, і поводження будь-якої системи або будь-якої сутності можна описати за допомогою множини записів у вигляді списків [4–5].

Відмінною особливістю CLIPS є конструктори для створення баз знань, які не повертають жодних значень на відміну від функцій:

`deffunction` – визначення функцій;
`deftemplate` – визначення шаблону факту;
`deffacts` – визначення фактів;
`defrule` – визначення правил;
`defglobal` – визначення глобальних змінних;
`defmodule` – визначення модулів;
`defclass` – визначення класів;
`defintances` – визначення об'єктів за заданим шаблоном;
`defmessage-handler` – визначення повідомлень для об'єктів;

`defgeneric` – створення заголовків родової функції;
`defmethod` – визначення методу родової функції.

CLIPS надає можливість для набуття, зберігання та оброблення фактів і правил. Кожен факт у базі даних є записом у вигляді списку. Список може містити одне або кілька полів, які приймають символьні або числові значення, або бути порожнім.

У CLIPS є два види фактів: упорядковані та шаблонні. Шаблонні факти мають шаблон, що задається конструктором `deftemplate`. Упорядковані не мають явної конструкції `deftemplate`. Шаблонний факт має поля, які називаються слотами і оголошуються конструкцією `slot`. Факти розміщуються в робочій пам'яті, а нові факти поміщаються в робочу пам'ять функцією `assert`.

Знання предметної області подаються в CLIPS у вигляді правил, де ліва частина правила – це умова його спрацьовування, а права частина – це ті дії, які повинні виконатися в разі виконання умов. Якщо кожна умова в лівій частині правила знаходить себе серед фактів, то відбувається активація правила і виконуються всі дії, записані в його правій частині. В іншому випадку правило не активується. Правила в CLIPS оголошуються за допомогою конструктора `defrule` та складаються з передумов та наслідків.

Основна ідея створення ЕС полягає в поданні знань у вигляді такої форми:

Правило 1:
`ЯКЩО` (виконуються умови 1)
`ТОДІ` (виконати дії 1)
Правило 2:
`ЯКЩО` (виконуються умови 2)
`ТОДІ` (виконати дії 2)
...

Таке подання близьке до людського мислення і відрізняється від програм, написаних на традиційних алгоритмічних мовах, де дії впорядковані та виконуються строго за алгоритмом.

Порядок виконання практичного заняття

Приклад 1

Дана предметна область, що представляє учасників конференції, які приїхали з різних міст країни [10]. Необхідно підрахувати кількість учасників конференції, що є представниками кожного міста.

Процедура регістрації учасників складається з уведення відомостей про учасників у базу даних, у якій на кожного учасника виділяється один запис (факт), що складається зі списку з трьома полями:

– перше поле має символьне значення `rep`, що означає скорочення від `representative` (представник). Загалом це значення може бути будь-яким, а поле може бути відсутнім;

– друге поле списку зберігає прізвище учасника;

– третє поле призначено для введення міста, з якого учасник прибув.

1. Створити базу даних у вигляді текстового ASCII–файлу з ім'ям `rep.clp`, використовуючи будь-який текстовий редактор. Вміст фактів бази даних з ім'ям `rep` буде таким:

```
(deffacts rep
     (rep Alejnov Odesa)
     (rep Ladak Kharkiv)
     (rep Slobodjanjuk Kiev)
     (rep Klitka Odesa)
     (rep Bojko Kiev)
     (rep Pustovit Kharkiv)
     (rep Spokojnij Odesa)
     (rep Shamis Odesa)
     (rep Lobovko Kiev)
     (rep Zadorozhna Kharkiv)
     (rep Javorskij Kharkiv)).
```

2. Розробити програму та зберігти її у вигляді текстового ASCII–файлу зі стандартним для CLIPS–програми розширенням `.clp` та ім'ям `represent`:

```
(defglobal ?*odesa* = 0)
(defglobal ?*kiev* = 0)
(defglobal ?*kharkiv* = 0)

(defrule start
     (initial-fact)
=>
     (printout t crlf «REPRESENTATIVES» crlf))

(defrule odesa
     (rep ?Odesa)
=>
          (bind ?*odesa* (+ ?*odesa* 1)))

(defrule kiev
     (rep ?Kiev)
=>
          (bind ?*kiev* (+ ?*kiev* 1)))
```

```
(defrule kharkiv
    (rep ?Kharkiv)
=>
    (bind ?*kharkiv* (+ ?*kharkiv* 1)))

(defrule result
    (declare (salience -1))
        (initial-fact)
=>
    (printout t «from Odesa:» ?*odessa* crlf)
    (printout t «from Kiev:» ?*kiev* crlf)
    (printout t «from Kharkiv:» ?*kharkiv* crlf)).
```

Алгоритм розв'язання задачі, яка дозволяє підрахувати кількість учасників–представників кожного міста, такий:

1) задекларувати за допомогою конструктора `defglobal` три глобальні змінні `?*odesa*`, `?*kiev*` та `?*kharkiv*` , які є лічильниками;

2) записати правило з ім'ям `start`, ліва частина якого становить запис (`initial-fact`), яким позначається системний початковий факт, який створюється в робочій пам'яті інтерпретатора CLIPS за командою (`reset`) до запуску програми на виконання командою (`run`).

Системний початковий факт необхідно використовувати через те, що в CLIPS–програмах поширеними правилами є такі, які додають факти в базу даних, або, навпаки, видаляють їх. Типовою є ситуація, коли під час старту програми в базі даних немає фактів, які відповідають хоча б одному правилу. У цьому випадку програма нічого не виконує. Для того щоб розпочати обчислення і використовується системний початковий факт, який, незалежно від фактів у базі даних, активує деяке правило, додає такі факти, які, в свою чергу, активують правила, умови яких не виконувалися в початковий момент.

У цій програмі системний початковий факт (`initial-fact`) запускає правило `start`, яке активується незалежно від фактів у файлі `rep.clp`, і є присутнім в програмі тільки з однією метою – вивести заголовок «`REPRESENTATIVES`». Для цього в правій частині правила `start` викликається вбудована функція `printout` з ключем `t`, що виводить на стандартний пристрій виводу (монітор) заголовок, укладений в лапки. Комбінація символів `crlf` потрібна для переведення курсору на наступний рядок:

```
(defrule start
    (initial-fact)
        =>
    (printout t crlf «REPRESENTATIVES» crlf));
```

3) задати лічильник і послідовно проглядати списки в записах файлу `rep.clp` для знаходження третього поля списку, яке має значення `Kiev`. У разі знаходження такого значення зміст відповідного лічильника збільшиться на одиницю. Для цього використовується функція `bind`, яка є аналогом оператора присвоювання: зміст змінної `?*kiev*` збільшується на одиницю та результат зберігається в цій саме змінній.

Правило

```
(defrule kiev
         (rep ?Kiev)
             =>
         (bind ?*kiev* (+ ?*kiev* 1)))
```

активується в тому випадку, коли в базі даних знаходиться факт `(rep ? Kiev)`, у якому символ «?» у другому полі цього списку означає символ універсальної підстановки та замінює будь-яке прізвище. Це означає, що правило `kiev` активується стільки разів, скільки разів факт `(rep ? Kiev)` присутній в базі даних. Разом із тим стільки саме разів виконуються дії, які вміщуються в правій частині правила;

4) задати лічильники аналогічно для значення `Odesa` та `Kharkiv`;

5) обрахувати кількість учасників конференції з кожного міста;

6) вивести на друк отримані результати підрахунку кількості учасників конференції, які прибули з різних міст країни:

```
(defrule result
   (declare (salience -1))
      (initial-fact)
          =>
      (printout t «from Odesa:» ?*odesa* crlf)
      (printout t «from Kiev:» ?*kiev* crlf)
      (printout t «from Kharkiv:» ?* kharkiv * crlf)).
```

У CLIPS існує кілька стратегій черговості виконання правил, а самі правила можуть мати пріоритет, який задається вбудованою функцією `declare` з параметром `salience` (значущість). Цей параметр може приймати цілочисельні значення від –10 000 до +10 000. За замовчуванням для всіх правил величина `salience` дорівнює нулю. Якщо в правилі `result` не вказати пріоритет, воно буде конфліктувати з правилом `start` за черговість виконання, оскільки у цих правил однакова ліва частина. Для усунення конфлікту в правилі `result` пріоритет вказано явно і зі знаком мінус, у зв'язку з чим це правило виконується останнім;

7) завантажити командою `(load)` файл `rep.clp`;

8) завантажити командою `(load)` файл `represent.clp`;

9) подати команду `(reset)` (Ctrl+E);
10) подати команду `(run)` (Ctrl+R) для запуску програми на виконання.
Результат виконання програми наведено на рисунку 5.1.

```
CLIPS> (reset)
CLIPS> (run)

REPRESENTATIVES
from Odesa: 4
from Kiev: 3
from Kharkiv: 4
CLIPS>
```

Рисунок 5.1 – Результат виконання програми для підрахунку учасників конференції

Повідомлення інтерпретатора `TRUE` означає, що у файлі немає синтаксичних помилок і команда завантаження виконана коректно.

У цьому випадку в інтерпретаторі CLIPS знаходяться два файли. Перший, з ім'ям `rep.clp`, є базою даних. Другий, з ім'ям `represent.clp`, містить відомості (правила) про те, як ці дані можуть бути використані. Отже, разом файли утворюють базу знань, яка містить принаймні два знання. Перше знання – загальний склад учасників конференції, який можна подивитися, не виходячи з інтерпретатора, за командою (`facts`). Друге знання – кількість учасників від кожного міста.

У розглянутому прикладі база знань складається з двох програмних модулів. Однак можна використовувати одну програму, збережену в одному файлі.

У цьому прикладі евристичний механізм подання знань використовується разом із процедурним.

Приклад 2

Необхідно підібрати резистор для ділянки електричної схеми радіоелектронного пристрою. Резистор характеризується опором, який визначається за виміряним або розрахованим значенням електричного струму, що проходить через резистор, і падінням напруги на ньому [11].

1. Розробити програму з ім'ям `resistor.clp`, яка розв'язує цю задачу і яка, на відміну від попереднього прикладу, збережена в одному файлі:

```
(deffacts resistors ; База даних резисторів
    (resistor Ra 2)
    (resistor Rb 5)
    (resistor Rc 7))

(deffunction om ; Функція от(x,y)
```

```
        (?x ?y)
            (div ?y ?x))
    (defrule input ; Початкове правило
        (initial-fact)
    =>
        (printout t crlf "Input current value: ")
                (bind ?i (read))
                 ; Введення числового значення струму
        (printout t "Input voltage value: ")
                (bind ?u (read))
                 ; Введення числового значення напруги
        (assert (numbers ?i ?u)))
    (defrule take ; Підбір типу резистора з бази даних
             (numbers ?i ?u)
             (resistor ?r =(om ?i ?u))
    =>
    (printout t crlf "You must take resistor « ?r»." crlf
crlf)
    (reset)
    (halt))
    (defrule nothing ; Якщо в базі даних немає відповідного резистора
        (numbers ?i ?u)
        (resistor ?r ~=(om ?i ?u))
    =>
    (printout t crlf "There is nothing for You in my
database!" crlf  crlf)
    (reset)
    (halt)).
```

Програма складається з декількох частин: бази даних з ім'ям `resistors`, оголошення користувацької функції `om` і трьох правил з іменами `input`, `take` і `nothing`.

У базі даних містяться відомості про резистори, які подані у вигляді списків, що складаються з трьох полів:

– перше поле має значення `resistor`, яке відображає тип радіодеталі;

– друге поле списку містить тип резистора;

– останнє поле зберігає значення опору, наприклад, `(Resistor Ra 2)`.

Функція `om` використовується для представлення процедурного знання – закону Ома:

```
    (deffunction om ; Функція от(x,y)
        ( ?x ?y)
            (div ?y ?x)).
```

Правило input призначене для введення початкових даних. Це правило активується системним початковим фактом і потребує від користувача введення струму і напруги:

```
(defrule input ; Початкове правило
     (initial-fact)
=>
     (printout t crlf "Input current value: ")
              (bind ?i (read))

     (printout t "Input strait value: ")
              (bind ?u (read)).
```

Вбудована функція read повертає значення, які введені зі стандартного пристрою вводу (клавіатури), і зберігає їх у змінних ?i та ?u.

У правій частині правила виконується ще одна дія. Команда assert додає в робочу пам'ять інтерпретатора CLIPS факт (numbers ?i ?u) для того, щоб можна було звертатися до локальних змінних ?i та ?u, які є пов'язаними з правилом input, з інших правил програми:

```
     (assert (numbers ?i ?u))).
```

У наступних двох правилах користувачеві або пропонується тип відповідного резистора (правило take), або повідомляється про відсутність такого (правило nothing).

Ліва частина правила take

```
(defrule take; Підбір типу резистора з бази даних
     (numbers ?i ?u)
          (resistor ?r =(om ?i ?u))
=>
(printout t crlf "You must take resistor « ?r».” crlf
crlf)
(reset)
(halt))
```

складається з двох умов, тому правило активується, якщо обидві умови будуть виконані.

Перша умова виконується, оскільки відповідний факт уже створений правилом input.

Друга умова виконується, якщо буде точно відповідати будь-якому факту (списку) у базі даних. Перше поле умови питань не викликає. У другому полі умови знаходиться змінна ?r, яка може прийняти значення Ra, або Rb, або Rc

залежно від змісту третього поля умови, у якому здійснюється виклик функції om і зберігається значення, яке повертається функцією. Якщо значення, яке повертається, дорівнює 7, то умова виконується, змінна ?r прийме значення Rc, правило активується і виведе на екран монітора пропозицію вибрати резистор Rc. Якщо функцією om повертається значення, яке дорівнює 5, то користувачеві буде запропоновано резистор Rb.

У лівій частині правила nothing є модифікація:

```
(defrule nothing ; Якщо в базі даних немає відповідного резистора
              (numbers ?i ?u)
      (resistor ?r ~=(om ?i ?u))
=>
(printout t crlf "There is nothing for You in my database!" crlf  crlf)
(reset)
(halt)).
```

У третьому полі другої умови перед викликом функції om стоїть символ «~», який означає логічне заперечення. Отже, умова виконається і правило активується, якщо значення, яке повертається функцією om, буде не 2, не 5 і не 7.

2. Очистити командою (clear) робочій простір CLIPS від даних попереднього завдання.

3. Завантажити файл resistor.clp, і запустити програму на виконання.

Результат виконання програми для визначення резистора для ділянки електричної схеми радіоелектронного пристрою наведено на рисунку 5.2.

```
CLIPS> (reset)
CLIPS> (run)

Input current value: 5
Input strait value: 10

You must take resistor Ra.
```

Рисунок 5.2 – Результат виконання програми для визначення резистора для ділянки електричної схеми радіоелектронного пристрою

Отже, файл resistor.clp є базою знань, оскільки містить і базу даних, і відомості (правила) про те, як дані можуть бути використані.

Ця база має три знання:

– перше знання – загальний список резисторів із зазначенням типу й опору;

– друге знання – закон Ома:

– третє знання – пропонований тип резистора.

Контрольні питання

1. Дати визначення бази даних.
2. Дати визначення бази знань.
3. Назвати та охарактеризувати конструктори CLIPS.
4. З яких елементів складається правило CLIPS?
5. Охарактеризувати призначення функції `assert`.
6. Охарактеризувати призначення функції `retract`.
7. Охарактеризувати призначення функції `modify`.
8. Які процедурні функції використовуються в CLIPS, їхнє призначення.
9. Які процедурні функції реалізують можливості розгалуження, організації циклів у програмі?
10. Охарактеризувати призначення функції `bind`.

ПРАКТИЧНЕ ЗАНЯТТЯ 2
ЕКСПЕРТНА СИСТЕМА CIOS

Мета заняття – використання об’єктно-орієнтованої мови CLIPS та родових функцій.

Завдання до виконання практичного заняття

1. Ознайомитися з об’єктно-орієнтованими можливостями мови COOL та родовими функціями.
2. Ознайомитися з поданням логічних елементів та їхніх зв’язків.
3. Розробити ЕС CIOS, яка призначена для побудови й оптимізації таблиць істинності заданих логічних схем.

Загальні положення

Експертна система CIOS призначена для побудови й оптимізації таблиць істинності заданих логічних схем і ґрунтується на об’єктно-орієнтованих можливостях мови COOL. В ЕС розроблена ієрархія класів, об’єкти яких є елементами логічної схеми, завдяки чому ЕС CIOS містить невелику кількість правил. Водночас велика частина обробки інформації відбувається в об’єктах завдяки використанню обробників повідомлень різних типів. Крім об’єктно–орієнтованих можливостей CLIPS експертна система CIOS використовує також родові функції для зв’язування логічних елементів різних типів [6].

Логічною схемою називається об’єднання деяких примітивних логічних елементів. У таблиці 5.1 наведено список і опис логичних елементів.

Таблиця 5.1 – Примітивні логичні елементи

Логічний елемент	Опис роботи логічного елемента
SOURCE	Джерело – елемент, що становить вхід логічної схеми. Має один вихід і не має входів. Вважається, що початковий сигнал з'являється в цьому елементі та відразу передається наступному логічному елементу
LED	Індикатор – елемент, що моделює, вихід логічної схеми. Є протилежністю джерелу, має один вхід і не має виходів. Вважається, що після отримання сигналу індикатор просто відбиває його
SPLITTER	Роздільник – елемент призначений для розподілу сигналу. Має один вхід і два логічних виходи. Отриманий сигнал без зміни передається на обидва виходи роздільника
NOT	Логічне НІ – елемент, що має один вхід і один вихід. Змінює отриманий сигнал на протилежний
AND	Логічне І – елемент, що має два входи та один вихід. Повертає логічну 1, якщо обидва отриманих сигнали дорівнюють логічній 1. В іншому випадку повертає логічний 0
OR	Логічне АБО – елемент, що має два входи та один вихід. Повертає логічну 1, якщо хоча б один із отриманих сигналів дорівнює логічній 1. В іншому випадку повертає логічний 0
NAND	Заперечення логічного І – елемент має два входи та один вихід. Повертає логічний 0, якщо обидва отриманих сигнали дорівнюють логічній 1. В іншому випадку повертає логічну 1
XOR	Виключаюче АБО – елемент має два входи та один вихід. Повертає логічну 1, якщо отримані сигнали не рівні. В іншому випадку повертає логічний 0

Під час складання логічної схеми з примітивних елементів об'язковими є такі правила:

– вихід кожного логічного елемента пов'язаний з одним і тільки одним входом іншого елемента;

– вхід кожного логічного елемента пов'язаний з одним і тільки одним виходом іншого елемента.

Таблицею істинності називається таблиця, що містить набори сигналів джерел логічної схеми й оброблений результат, який отриманий на індикаторах схеми для кожного набору.

На рисунку 5.3 наведена проста логічна схема, таблицею істинності якої є таблиця 5.2.

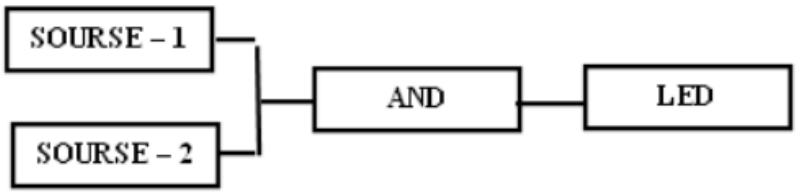

Рисунок 5.3 – Проста логічна схема

Таблиця 5.2 – Таблиця істиності простої схеми

Логічні елементи		
SOURCE–1	SOURCE–2	LED–1
0	0	0
0	1	0
1	0	0
1	1	1

Очевидно, що можливі ситуації, коли зміна значення одного з входів не впливає на результат за постійних значень сигналів на інших входах. Тоді таблицю можна оптимізувати шляхом об’єднання цих рядків таблиці, замінивши значення, що не впливає на результат входу символом (*).

У наведеному прикладі у таблиці 5.2 можна об’єднати перший та другий рядок, оскільки за будь-якого значення, що подається на джерело SOURCE–2, у разі, якщо на джерело SOURCE–1 подається логічний 0, результат буде дорівнювати логічному 0. Отже, оптимізована таблиця набуде вигляду, який відповідний таблиці 5.3.

Таблиця 5.3 – Оптимізована таблиця істиності простої схеми

Логічні елементи		
SOURCE–1	SOURCE–2	LED–1
0	*	0
1	0	0
1	1	1

Порядок виконання практичного заняття

Необхідно розробити та протестувати ЕС, яка призначена для побудови й оптимізації таблиць істинності заданих логічних схем, наведених на рисунках 5.4–5.6 [12].

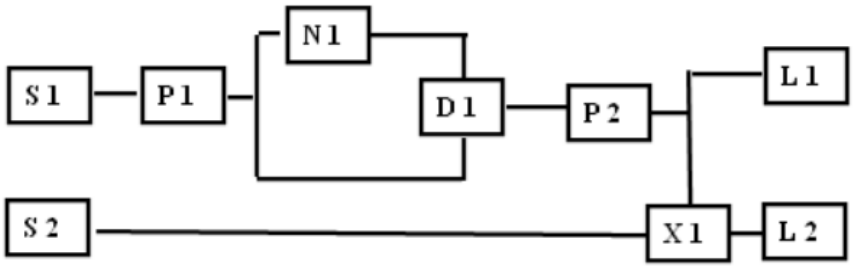

Рисунок 5.4 – Логічна схема 1

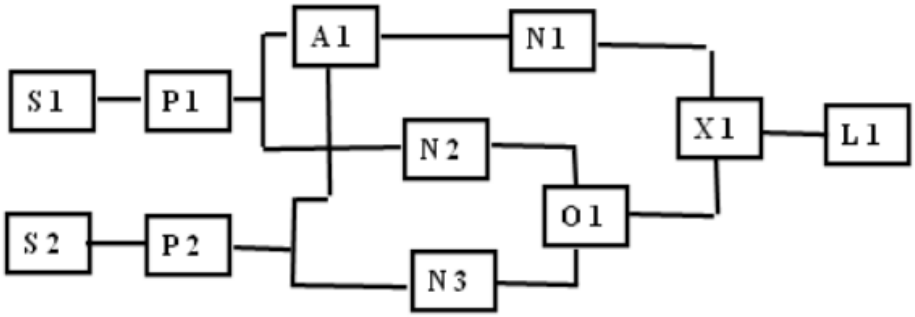

Рисунок 5.5 – Логічна схема 2

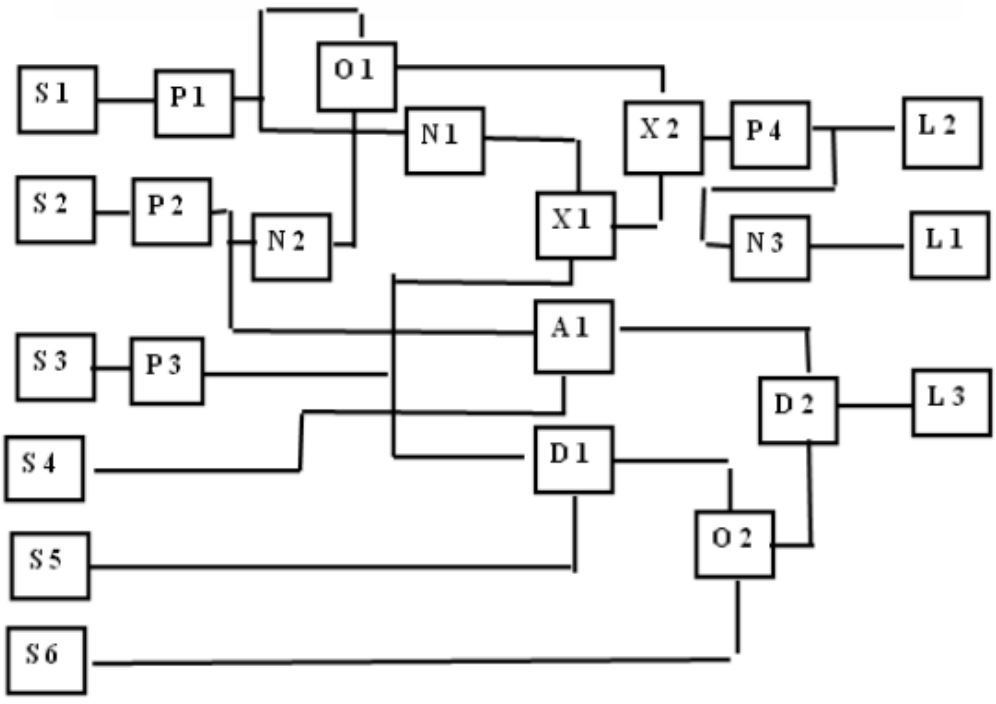

Рисунок 5.6 – Логічна схема 3

У таблиці 5.4 наведені скорочення назв логічних елементів (рис. 5.4–5.6).

Таблиця 5.4 – Скорочення назв логічних елементів

Назва логічного елемента	Скорощення, що використовується
SOURCE	S
LED	L
SPLITTER	P
NOT	N
AND	A
OR	O
NAND	D
XOR	X

Для вирішення завдання побудови й оптимізації таблиць істинності логічних схем доцільно використовувати такий алгоритм:

1) ініціалізація заданої логічної схеми і встановлення необхідних з'єднань між її елементами;

2) отримання результату роботи логічної схеми у разі, якщо на всі її джерела подаються значення, що дорівнюють логічному 0;

3) послідовне перебирання комбінацій вхідних сигналів для всіх джерел логічної схеми й обчислення результатів її роботи.

Перебирання комбінацій вхідних сигналів здійснюється за допомогою циклічного (рефлексного) коду Грея. Цей код є особливим методом подання двійкових чисел, у якому кожне наступне число відрізняється від попереднього тільки на один символ. Наприклад, коди Грея для чисел від 0 до 7 будуть: 0 = 000; 1 = 001; 2 = 011; 3 = 010; 4 = 110; 5 = 111; 6 = 101; 7 = 100.

Використання коду Грея дозволяє значно оптимізувати роботу експертної системи за часом виконання, тому що для отримання результату обробки кожного наступного варіанта вхідних сигналів необхідно змінити значення тільки одного джерела;

4) пошук у процесі отримання результатів роботи логічної схеми для комбінацій вхідних сигналів двох рядків таблиці, які можна об'єднати згідно з алгоритмом оптимізації, наведеним у таблиці 5.3;

5) виведення на екран знайдену таблицю істинності після отримання результатів роботи логічної схеми для всіх комбінацій вхідних сигналів і їхньої оптимізації.

1. Розробити відповідний клас для подання кожного логічного елемента, об'єкт якого буде самостійно виконувати оброблення сигналу, що надходить, і передавати отриманий результат на вхід елемента, який з'єднано з виходом.

Складена у такий спосіб логічна схема буде самостійно обчислювати результати своєї роботи під час потрапляння відповідних сигналів на вхід джерел. Для реалізації класів, що становлять логічні елементи, необхідно

скористатися можливістю множинного наслідування. Потрібно зауважити, що кожен логічний елемент є компонентом, який має деяку кількість входів (від 0 до 2) і виходів (від 0 до 2). Грунтуючись на цьому зауваженні, створюється набір класів і обробників повідомлень.

Далі наведені класи, які використовуються для створення логічних елементів:

```
(defclass COMPONENT
    (is-a USER)
        (slot ID# (create-accessor write))).
```

Клас COMPONENT є абстрактним нащадком класу USER. Цей клас містить єдиний слот ID# для ідентифікаційного номера компонента і визначає для нього акцесор запису. Клас буде суперкласом для всіх класів логічних елементів:

```
(defclass NO-OUTPUT
    (is-a USER)
(slot number-of-outputs (access read-only)
        (default 0)
        (create-accessor read) ) )
(defmessage-handler NO-OUTPUT compute-output ( ) )

(defclass ONE-OUTPUT
         (is-a NO-OUTPUT)
    (slot number-of-outputs (access read-only)
        (default 1)
        (create-accessor read) )
     (slot output-1 (default UNDEFINED)
        (create-accessor write) )
    (slot output-1-link (default GROUND)
        (create-accessor write) )
        (slot output-1-link-pin (default 1)
        (create-accessor write) ) )
(defmessage-handler  ONE-OUTPUT  put-output-1  after
(?value)
 (send ?self:output-1-link
 (sym-cat put-input- ?self:output-1-link-pin)
?value))

(defclass TWO-OUTPUT
    (is-a ONE-OUTPUT)
    (slot number-of-outputs (access read-only)
        (default 2)
```

```
            (create-accessor read) )
        (slot output-2
            (default UNDEFINED)
            (create-accessor write) )
        (slot output-2-link
            (default GROUND)
            (create-accessor write) )
        (slot output-2-link-pin
            (default 1)
            (create-accessor write) ) )
    (defmessage-handler TWO-OUTPUT put-output-2 after
(?value)
    (send ?self:output-2-link
    (sym-cat put-input- ?self:output-2-link-pin)
    ?value)).
```

Набір класів NO-OUTPUT, ONE-OUTPUT і TWO-OUTPUT призначений для реалізації властивостей елемента без виходу, з одним виходом і двома виходами відповідно. Кожен наступний клас успадковується від попереднього і додає до нього реалізацію одного логічного виходу.

Клас NO-OUTPUT має єдиний слот number-of-outputs зі значенням за замовчуванням 0 і доступний тільки для читання. Цей слот призначений для зберігання числа виходів елемента, успадкованого від цього класу. Крім того, клас NO-OUTPUT визначає обробник compute-output, який успадковує і перевизначає всі логічні елементи. Саме завдяки наявності цього обробника об'єкти класів логічних елементів зможуть самостійно визначати результат обробки отриманих сигналів.

Клас ONE-OUTPUT перевизначає слот number-of-outputs і призначає йому значення за замовчуванням, що дорівнює 1. Крім того, клас визначає набір слотів, які призначені для опису логічного виходу елемента, і зв'язку цього виходу з подальшими елементами схеми.

Слот output-1 призначено для зберігання значення, що утворюється на виході елемента. За замовчуванням слот має значення undefined.

Слот output-1-link містить ім'я об'єкта, з яким пов'язаний цей вихід, і за замовчуванням має значення ground.

Слот output-1-link-pin містить номер конкретного логічного входу елемента, з яким пов'язаний цей вихід, і за замовчуванням дорівнює 1.

Клас ONE-OUTPUT визначає after-обробник put-outout-1, який відразу після присвоєння значення слоту output-1 передає це значення на вхід елемента, пов'язаного з цим виходом.

Клас TWO-OUTPUT теж перевизначає слот number-of-outputs і присвоює йому значення за замовчуванням, що дорівнює 2, а також додає набір

слотів, призначених для опису другого логічного виходу: output-2, output-2-link і output-2-link-pin і after-обробник put-outout-2.

Потрібно звернути увагу на той факт, що слоти output-1, output-1-link і output-l-link-pin, і обробник put-outout-1 не перевизначаються, а успадковуються від класу ONE-OUTPUT:

```
(defclass NO-INPUT
    (is-a USER)
    (slot number-of-inputs
        (access read-only)
        (default 0)
        (create-accessor read)))

(defclass ONE-INPUT
    (is-a NO-INPUT)
    (slot number-of-inputs (access read-only)
        (default 1)
        (create-accessor read))
    (slot input-1 (default UNDEFINED)
        (visibility public)
        (create-accessor read-write))
    (slot input-1-link
        (default GROUND)
        (create-accessor write))
    (slot input-1-link-pin
        (default 1)
        (create-accessor write)))

(defmessage-handler ONE-INPUT put-input-1 after
(?value)
    (send ?self compute-output))

(defclass TWO-INPUT (is-a ONE-INPUT)
    (slot number-of-inputs (access read-only)
        (default 2)
        (create-accessor read) )
    (slot input-2 (default UNDEFINED)
        (visibility public)
        (create-accessor write) )
    (slot input-2-link (default GROUND)
        (create-accessor write) )
    (slot input-2-link-pin
        (default 1)
        (create-accessor write) ) )
```

```
    (defmessage-handler TWO-INPUT put-input-2 after
(?value)
    (send ?self compute-output)).
```

Набір класів NO-INPUT, ONE-INPUT і TWO-INPUT призначений для реалізації властивостей елемента без входу, з одним входом і двома входами відповідно. Кожен наступний клас успадковується від попереднього і додає до нього реалізацію ще одного логічного входу. Призначення слотів, які описують логічний вхід елементів input-x, input-x-link, input-x-link-pin, ідентично призначенням слотів, що використовуються в класах ONE-OUTPUT і TWO-OUTPUT.

Кардинальна відмінність класів, що реалізують логіку входів елемента, полягає в тому, що замість обробника put-outout-x класи реалізують after-обробник put-inout-x. Цей обробник викликає обробник повідомлення compute-output відразу після отримання нового значення.

2. Розробити класи, які реалізують логіку елементів SOURCE і LED:

```
(defclass SOURCE
    (is-a NO-INPUT ONE-OUTPUT COMPONENT)
        (role concrete)
    (slot output-1
        (default UNDEFINED)
        (create-accessor write)) )

(defclass LED
    (is-a ONE-INPUT NO-OUTPUT COMPONENT)
        (role concrete)).
```

Клас SOURCE реалізує поводження джерела. Цей логічний елемент має один вихід і не має входів.

Клас LED є реалізацією логічного індикатора, має один вхід і не має виходів.

3. Розробити функцію, яка буде вирішувати проблему збору результату обробки вхідних сигналів з усіх індикаторів логічної схеми:

```
(deffunction LED-response ()
    (bind ?response (create$))
    (do-for-all-instances ((?led LED)) TRUE
    (bind ?response (create$ ?response
    (send ?led get-input-1))))
?response).
```

Ця функція збирає складене поле зі значень, що зберігаються в усіх об'єктах класу LED поточної логічної схеми, за допомогою функції do-for-all-instances.

4. Розробити клас, який реалізує логіку елемента NOT-GATE, і його реалізація буде нагадувати класи SOURCE і LED, за винятком, того, що інші логічні елементи повинні перевизначати обробник compute-output, який буде проводити обробку сигналу, отриманого на вході елемента:

```
(defclass NOT-GATE
    (is-a ONE-INPUT ONE-OUTPUT COMPONENT)
    (role concrete))

(deffunction not# (?x) (- 1 ?x))

(defmessage-handler NOT-GATE compute-output ()
    (if (integerp ?self:input-1) then
    (send ?self put-output-1 (not# ?self:input-1))))
```

У реалізації обробника compute-output використовується функція not#, яка і виконує необхідні обчислення.

Визначення класу NOT-GATE компактно, однак, завдяки використанню продуманої ієрархії класів і множинного наслідування, клас має всю необхідну функціональність.

5. Розробити клас, який реалізує логіку елемента AND-GATE:

```
(defclass AND-GATE
    (is-a TWO-INPUT ONE-OUTPUT COMPONENT)
    (role concrete))

(deffunction and# (?x ?y)
    (if (and (!= ?x 0) (!= ?y 0)) then 1 else 0))

(defmessage-handler AND-GATE compute-output ()
    (if (and (integerp ?self:input-1)
    (integerp ?self:input-2) ) then
    (send ?self put-output-1
    (and#       ?self:input-1       ?self:input-2))))
```

6. Розробити клас, який реалізує логіку елемента OR-GATE:

```
(defclass OR-GATE
    (is-a TWO-INPUT ONE-OUTPUT COMPONENT)
```

```
    (role concrete))

(deffunction or# (?x ?y)
    (if (or (!= ?x 0) (!= ?y 0)) then 1 else 0))

(defmessage-handler OR-GATE compute-output ()
    (if (and (integerp ?self:input-1)
    (integerp ?self:input-2) ) then
    (send ?self put-output-1
    (or# ?self:input-1 ?self:input-2))).
```

7. Розробити клас, який реалізує логіку елемента `NAND-GATE`:

```
(defclass NAND-GATE
        (is-a TWO-INPUT ONE-OUTPUT COMPONENT)
        (role concrete))

(deffunction nand# (?x ?y)
(if (not (and (!= ?x 0) (!= ?y 0))) then 1 else 0))

(defmessage-handler NAND-GATE compute-output ()
        (if (and (integerp ?self:input-1)
        (integerp ?self:input-2 )) then
        (send ?self put-output-1
        (nand# ?self:input-1 ?self:input-2))).
```

8. Розробити клас, який реалізує логіку елемента `XOR-GATE`:

```
(defclass XOR-GATE
        (is-a TWO-INPUT ONE-OUTPUT COMPONENT)
        (role concrete))

(deffunction xor# (?x ?y)
        (if (or (and (= ?x 1) (= ?y 0))
        (and (= ?x 0) (= ?y 1))) then 1 else 0))

(defmessage-handler XOR-GATE compute-output ()
        (if (and (integerp ?self:input-1)
        (integerp ?self:input-2)) then
        (send ?self put-output-1
        (xor# ?self:input-1 ?self:input-2))).
```

9. Розробити клас, який реалізує логіку елемента `SPLITTER`:

```
(defclass SPLITTER
        (is-a ONE-INPUT TWO-OUTPUT COMPONENT)
        (role concrete))

(defmessage-handler SPLITTER compute-output ()
        (if (integerp ?self:input-1) then
        (send ?self put-output-1 ?self:input-1)
        (send ?self put-output-2 ?self:input-1))).
```

10. Розробити конструктор definstances, який призначено для встановлення зв'язку між логічними елементами та об'єднання всіх логічних елементів у логічну схему, наведену на рисунку 5.4.

```
(definstances circuit
        (S-1 of SOURCE)
        (S-2 of SOURCE)
        (P-1 of SPLITTER)
        (P-2 of SPLITTER)
        (N-1 of NOT-GATE)
        (O-1 of OR-GATE)
        (X-1 of XOR-GATE)
        (L-1 of LED)
        (L-2 of LED)).
```

11. Розробити родову функцію connect з методами, які враховують відмінність об'єктів, що з'єднуються, отриманих як параметри. Ця родова функція розробляється замість кількох функцій для кожного типу з'єднання:

```
(defgeneric connect)
(defmethod connect ((?out ONE-OUTPUT)
                    (?in ONE-INPUT))
        (send ?out put-output-1-link ?in)
        (send ?out put-output-1-link-pin 1)
        (send ?in put-input-1-link ?out)
        (send ?in put-input-1-link-pin 1))
(defmethod connect ((?out ONE-OUTPUT)
                  (?in TWO-INPUT) (?in-pin INTEGER))
    (send ?out put-output-1-link ?in)
    (send ?out put-output-1-link-pin ?in-pin)
(send ?in  (sym-cat put-input- ?in-pin -link) ?out)
(send ?in  (sym-cat put-input- ?in-pin -link-pin) 1))

(defmethod connect ((?out TWO-OUTPUT)
                  (?out-pin INTEGER) (?in ONE-INPUT))
```

```
    (send ?out (sym-cat put-output- ?out-pin -link) ?in)
(send ?out (sym-cat put-output- ?out-pin -link-pin) 1)
    (send ?in put-input-1-link ?out)
    (send ?in put-input-1-link-pin ?out-pin))

    (defmethod connect ((?out TWO-OUTPUT)
    (?out-pin INTEGER) (?in TWO-INPUT) (?in-pin INTEGER))
    (send ?out (sym-cat put-output- ?out-pin -link) ?in)
    (send ?out (sym-cat put-output- ?out-pin -link-pin)
    ?in-pin)
    (send ?in  (sym-cat put-input- ?in-pin -link) ?out)
    (send ?in   (sym-cat put-input- ?in-pin -link-pin)
?out-pin)).
```

12. Написати процедуру ініціалізації логічної схеми, наведеної на рисунку 5.4, тому що наведені методи родової функції `connect` враховували всі можливі типи з'єднань логічних аргументів:

```
(deffunction connect-circuit()
        (connect [S-1] [P-1])
        (connect [S-2] [X-1] 2)
        (connect [P-1] 1 [N-1])
        (connect [P-1] 2 [O-1] 2)
        (connect [N-1]  [O-1] 1)
        (connect [O-1] [P-2])
        (connect [P-2] 1 [L-1])
        (connect [P-2] 2 [X-1] 1)
        (connect [X-1]   [L-2])).
```

Попередня об'ява функції `connect-circuit` має вигляд:

```
(deffunction connect-circuit ()).
```

Застосування описаного вище методу зв'язування елементів у логічну схему дозволили розділити дані та методи їхнього оброблення. Кожна окрема схема може бути розташована в окремому файлі та підключатися до ЕС за допомогою команди `load`.

13. Створити глобальні змінні, які разом з описаними раніше класами, об'єктами, функціями і родовими функціями дозволяють реалізувати логіку ЕС:

```
(defglobal ?*gray-code*    =(create$)
    ?*sources* =(create$)
    ?*max-iterations* = 0).
```

Змінна max-iterations використовується для зберігання максимальної кількості ітерацій під час перебирання комбінацій вхідних сигналів системи.

У змінній sources зберігаються імена всіх джерел поточної логічної схеми.

Змінна gray-code призначена для зберігання поточного коду Грея.

14. Розробити функцію change-which-bit, яка визначає номер елемента в коді Грея, значення якого необхідно змінити для перебирання всіх можливих варіантів вхідних сигналів:

```
(deffunction change-which-bit (?x)
    (bind ?i 1)
    (while (and (evenp ?x) (!= ?x 0)) do
    (bind ?x (div ?x 2))
    (bind ?i (+ ?i 1)))
    ?i).
```

15. Розробити правило, яке ініціалізує завантажену логічну схему (рис. 5.4).

```
(defrule startup
=>
    (connect-circuit)
(bind ?*sources* (find-all-instances ((?x SOURCE)) TRUE))
    (do-for-all-instances ((?x SOURCE)) TRUE
    (bind ?*gray-code* (create$ ?*gray-code* 0) ) )
(bind ?*max-iterations* (round (** 2 (length
?*sources*))))
    (assert (current-iteration 0))).
```

Правило startup не має явних умовних елементів, тому буде активовано першим фактом initial-fact.

Після запуску правила викликається функція connect-circuit, яка ініціалізує поточну логічну схему, після цього правило отримує імена всіх елементів джерел в змінну sources.

У подальшому правило startup створює нульовий код Грея для початку процесу перебору всіх варіантів вхідних сигналів, обчислює максимальну кількість ітерацій і додає факт із номером поточної ітерації.

16. Розробити правило compute-response-1st-time, яке запускає ітераційний процес перебирання комбінацій вхідних сигналів:

```
(defrule compute-response-1st-time
    ?f <- (current-iteration 0)
```

```
    =>
    (do-for-all-instances ((?source SOURCE)) TRUE
    (send ?source put-output-1 0))
    (assert (result ?*gray-code* =(str-implode (LED-
response))))
    (retract ?f)
    (assert (current-iteration 1))).
```

Це правило призначене для першого кроку перебирання вхідних сигналів. Воно передає нульовий сигнал усіма джерелами логічної схеми, що спричиняє автоматичне обчислення результату. Далі отриманий результат зберігається у факті вигляді: (*result поточний – стан – джерел поточний – стан – індикаторів*).

Після цього правило видаляє факт (`current-iteration 0`), який і запускає правило і додає факт (`current-iteration 1`).

17. Розробити правило `compute-response-other-times`, логіка роботи якого подібна до роботи правила `compute-response-1st-time`, за винятком того, що на початку за допомогою виклику функції `change-which-bit` воно визначає, яке з джерел схеми повинно поміняти вхідний сигнал, і змінює його:

```
    (defrule compute-response-other-times
    ?f <- (current-iteration ?n&~0&:(< ?n ?*max-
iterations*))
    =>
        (bind ?pos (change-which-bit ?n))
        (bind ?nv (- 1 (nth ?pos ?*gray-code*)))
    (bind ?*gray-code* (replace$ ?*gray-code* ?pos ?pos
?nv))
        (send (nth ?pos ?*sources*) put-output-1 ?nv)
    (assert (result ?*gray-code* =(str-implode (LED-
response))))
        (retract ?f)
        (assert (current-iteration =(+ ?n 1)))).
```

Після цього правило отримує результат роботи логічної схеми і зберігає його в відповідному факті. Крім того, правило збільшує число ітерацій на одиницю.

Правило виконується, поки число ітерацій не досягне граничного значення.

Ще однією умовою виконання правила є те, що одна ітерація вже була здійснена (число ітерацій не дорівнює 0).

18. Розробити правило `merge-responses`, яке призначене для оптимізації обчислюваної таблиці істинності і має більш високий пріоритет, тому виконується відразу після того, як у системі з'являться два факти, що задовольняють заданій масці:

```
(defrule merge-responses
    (declare (salience 10))
    ?fl <- (result $?b ?x $?e ?response)
    ?f2 <- (result $?b ~?x $?e ?response)
=>
    (retract ?fl ?f2)
    (assert (result ?b * ?e ?response))).
```

19. Розробити правило `print-header`, яке призначено для виведення на екран заголовка таблиці істинності:

```
(defrule print-header
    (declare (salience -10) )
=>
    (do-for-all-instances ((?x SOURCE)) TRUE
    (format t " %3s " (sym-cat ?x) ) )
    (printout t " | ")
    (do-for-all-instances ((?x LED)) TRUE
    (format t " %3s " ( sym-cat ?x) ) )
    (format t "%n")
    (do-for-all-instances ((?x SOURCE)) TRUE
    (printout t "-----"))
    (printout t "-+-")
    (do-for-all-instances ((?x LED)) TRUE
    (printout t "-----"))
    (format t "%n")
    (assert (print-results))).
```

Це правило додає в систему факт `print-results`, що активує правило `print-result`.

Заголовок містить список усіх джерел системи та індикаторів, які розділені вертикальною рискою. Крім того, для більшої зручності сприйняття правило відокремлює заголовок таблиці від її змісту додатковим рядком.

Це правило має більш низький пріоритет, ніж інші правила експертної системи, і не має явних умовних елементів, тому виконується тільки після завершення перебирання комбінацій вхідних сигналів логічної схеми.

20. Розробити правило `print-result`, яке виводить на екран оптимізовану таблицю істинності, сортуючи водночас її рядки:

```
(defrule print-result
    (print-results)
    ?f <- (result $?input ?response)
    (not (result $?input-2 ?response-2&:
    (< (str-compare ?response-2 ?response) 0) ) )
=>
    (retract ?f)
    (while (neq ?input (create$)) do
    (printout t "  " (nth 1 ?input) "  ")
        (bind ?input (rest$ ?input)))
    (printout t " | ")
    (bind ?response (str-explode ?response))
    (while (neq ?response (create$)) do
    (printout t "  " (nth 1 ?response) "  ")
    (bind ?response (rest$ ?response)))
(printout t crlf)).
```

Для тестування роботи ЕС необхідно створити три файли, які відповідають логічним схемам (рис. 5.4–5.6) і зберігти їх під ім'ям SCHEMA1.CLP, SCHEMA2.CLP, SCHEMA3.CLP.

1. Ввести у файл з ім'ям SCHEMA1.CLP такі дані:

```
(definstances circuit
   (S-1 of SOURCE)
   (S-2 of SOURCE)
   (P-1 of SPLITTER)
   (P-2 of SPLITTER)
   (N-1 of NOT-GATE)
   (O-1 of OR-GATE)
   (X-1 of XOR-GATE)
   (L-1 of LED)
   (L-2 of LED)
    )

(deffunction connect-circuit()
   (connect [S-1] [P-1])
   (connect [S-2] [X-1] 2)
   (connect [P-1] 1 [N-1])
   (connect [P-1] 2 [O-1] 2)
   (connect [N-1]  [O-1] 1)
   (connect [O-1] [P-2])
   (connect [P-2] 1 [L-1])
   (connect [P-2] 2 [X-1] 1)
   (connect [X-1] [L-2])).
```

2. Запустити розроблену ЕС та створений файл SCHEMA1.CLP. Експертна система сформує наступну таблицю істинності, яка наведена на рисунку 5.7, згідно з логічною схемою 1 (рис. 5.4):

```
CLIPS> (reset)
CLIPS> (run)
 S-1  S-2  |  L-1  L-2
-----------+-----------
  *    1   |   1    0
  *    0   |   1    1
CLIPS>
```

Рисунок 5.7 – Таблиця істинності логічної схеми 1

3. Ввести у файл із ім’ям SCHEMA2.CLP такі дані:

```
(definstances circuit
    (S-1 of SOURCE)
    (S-2 of SOURCE)
    (P-1 of SPLITTER)
    (P-2 of SPLITTER)
    (A-1 of AND-GATE)
    (N-1 of NOT-GATE)
    (N-2 of NOT-GATE)
    (N-3 of NOT-GATE)
    (O-1 of OR-GATE)
    (X-1 of XOR-GATE)
    (L-1 of LED)
    )

(deffunction connect-circuit()
    (connect [S-1] [P-1])
    (connect [S-2] [P-2])
    (connect [P-1] 1 [A-1] 1)
    (connect [P-1] 2 [N-2])
    (connect [P-2] 1 [A-1] 2)
    (connect [P-2] 2 [N-3])
    (connect [A-1] [N-1])
    (connect [N-2] [O-1] 1)
    (connect [N-3] [O-1] 2)
    (connect [N-1] [X-1] 1)
    (connect [O-1] [X-1] 2)
    (connect [X-1] [L-1])) .
```

ЕС сформує таблицю істинності, наведену на рисунку 5.8, згідно з логічною схемою 2 (рис. 5.5).

```
CLIPS> (reset)
CLIPS> (run)
 S-1  S-2  |  L-1
-----------+------
  *    *   |   0
CLIPS>
```

Рисунок 5.8 – Таблиця істинності логічної схеми 2

4. Ввести у файл із ім’ям SCHEMA3.CLP такі дані:

```
(definstances circuit
    (S-1 of SOURCE)
    (S-2 of SOURCE)
    (S-3 of SOURCE)
    (S-4 of SOURCE)
    (S-5 of SOURCE)
    (S-6 of SOURCE)
    (P-1 of SPLITTER)
    (P-2 of SPLITTER)
    (P-3 of SPLITTER)
    (P-4 of SPLITTER)
    (N-1 of NOT-GATE)
    (N-2 of NOT-GATE)
    (N-3 of NOT-GATE)
    (O-1 of OR-GATE)
    (O-2 of OR-GATE)
    (X-1 of XOR-GATE)
    (X-2 of XOR-GATE)
    (A-1 of AND-GATE)
    (D-1 of NAND-GATE)
    (D-2 of NAND-GATE)
    (L-1 of LED)
    (L-2 of LED)
    (L-3 of LED)
)

(deffunction connect-circuit()
    (connect [S-1] [P-1])
    (connect [S-2] [P-2])
    (connect [S-3] [P-3])
    (connect [S-4] [A-1] 2)
    (connect [S-5] [D-1] 2)
    (connect [S-6] [O-2] 2)
```

```
(connect [P-1] 1 [O-1] 1)
(connect [P-1] 2 [N-1])
(connect [P-2] 1 [N-2])
(connect [P-2] 2 [A-1] 1)
(connect [P-3] 1 [X-1] 2)
(connect [P-3] 2 [D-1] 1)
(connect [N-1]  [X-1] 1)
(connect [N-2]  [O-1] 2)
(connect [O-1] [X-2] 1)
(connect [X-1] [X-2] 2)
(connect [A-1] [D-2] 1)
(connect [D-1] [O-2] 1)
(connect [X-2] [P-4])
(connect [O-2] [D-2] 2)
(connect [P-4] 1 [N-3])
(connect [P-4] 2 [L-2])
(connect [D-2] [L-3])
(connect [N-3] [L-1])
).
```

ЕС сформує таблицю істинності, наведену на рисунку 5.9, згідно з логічною схемою 3 (рис. 5.6).

```
CLIPS> (reset)
CLIPS> (run)
 S-1  S-2  S-3  S-4  S-5  S-6  |  L-1  L-2  L-3
------------------------------+---------------
  *    1    0    1    *    *   |   0    1    0
  1    0    0    1    *    *   |   0    1    1
  0    0    1    *    *    *   |   0    1    1
  0    1    0    0    *    *   |   0    1    1
  1    *    0    0    *    *   |   0    1    1
  *    1    1    1    0    0   |   1    0    0
  *    1    1    1    *    1   |   1    0    0
  *    1    1    0    0    0   |   1    0    1
  *    1    1    *    1    0   |   1    0    1
  1    0    1    *    *    *   |   1    0    1
  *    1    1    0    *    1   |   1    0    1
  0    0    0    *    *    *   |   1    0    1
CLIPS>
```

Рисунок 5.9 – Таблиця істинності логічної схеми 3

Контрольні питання

1. Яке призначення та синтаксис класу `COMPONENT` ЕС CIOS?

2. Для реалізації властивостей яких елементів призначено набір класів `NO-OUTPUT`, `ONE-OUTPUT` і `TWO-OUTPUT` ?

3. Для реалізації властивостей яких елементів призначено набір класів `NO-INPUT`, `ONE-INPUT` і `TWO-INPUT` ?

4. Поводження якого логічного елемента реалізує розроблений клас `SOURCE`? Скільки входів та виходів має цей логічний елемент?

5. Поводження якого логічного елемента реалізує розроблений клас `LED`? Скільки входів та виходів має цей логічний елемент?

6. Охарактеризувати розроблені в ЕС CIOS класи для реалізації логічних елементів `NOT-GATE`, `AND-GATE`, `OR-GATE`, `NAND-GATE`, `XOR-GATE`, `SPLITTER`.

7. Який розроблений в ЕС конструктор призначено для встановлення зв'язку між логічними елементами та об'єднання всіх логічних елементів у логічну схему?

8. Яка процедура ініціалізації логічної схеми написана в розробленій ЕС?

9. Які глобальні знінні використані в ЕС?

10. Навести синтаксис правил у CLIPS. Яке розроблене правило ініціалізує завантаження логічної схеми?

11. Яке правило призначене для оптимізації таблиці істинності, яка обчислюється? Який пріоритет має це правило?

12. Яке правило призначене для виведення на екран заголовка таблиці істиності?

13. Яке правило призначене для виведення на екран оптимізованої таблиці істиності?

14. Описати алгоритм тестування роботи ЕС CIOS.

ПРАКТИЧНЕ ЗАНЯТТЯ 3
ОБ'ЄКТНО-ОРІЄНТОВАНИЙ ПІДХІД ДО РОЗРОБКИ ЕКСПЕРТНОЇ СИСТЕМИ

Мета заняття – використання парадігми об'єктно-орієнтованого програмування в CLIPS.

Завдання до виконання практичного заняття

1. Ознайомитися з об'єктно-орієнтованими можливостями мови COOL.

2. Оволодіти методами об'єктно-орієнтованого розширення CLIPS.

3. Розробити ЕС «Звіт про проведені заходи в студентському гуртожитку».

Загальні положення

CLIPS, як і будь-яка об'єктно-орієнтована система, має в своєму арсеналі такі п'ять характеристик: абстрактність, інкапсуляція, наслідування, поліморфізм і динамічне зв'язування [1, 5, 6, 8, 13–14].

Створення нового класу реалізує можливість абстрактного подання нового типу даних. Слоти й обробники повідомлень цього класу визначають властивості та поводження цілої групи об’єктів, що належать цьому класу.

Інкапсуляція реалізується в CLIPS вимогою обов’язково використовувати повідомлення під час роботи з об’єктами класів, які визначені користувачем. Обробники повідомлень класу становлять доступний користувачеві інтерфейс, що приховує реалізацію класу.

COOL підтримує множинне наслідування. Це означає, що певний клас може мати всі властивості зазначеного одного або декількох суперкласів.

Для встановлення лінійного порядку наслідування властивостей класів за множинного наслідування COOL використовує список передування класів (class precedence list), побудований з використанням ієрархії наслідування. Об’єкт, який становить екземпляр нового класу, наслідує всі властивості (слоти) і поводження (обробники повідомлень) кожного класу зі списку передування класів. Слово «передування» означає, що властивості та поводження класу, що знаходиться ближче до початку списку, перевизначають конфліктуючі визначення класів, що раніше зустрічалися.

Різні COOL–об’єкти можуть реагувати на одне й те саме повідомлення абсолютно по-різному. Це реалізує властивість поліморфізму. На практиці це виконується приєднанням до різних класів обробників одного і того саме повідомлення, але з різними послідовностями дій, що виконуються.

CLIPS також підтримує можливість динамічного зв’язування, яка реалізована за допомогою функції `send`, що призначена для надсилання повідомлень об’єкту. Виклик цієї функції здійснюється саме в процесі виконання програми, отже, визначення обробника також відбувається в процесі виконання програми. Наприклад, функція `send` може отримувати як параметр змінну, яка в різні моменти часу надсилається на різні об’єкти, разом із тим можуть викликатися абсолютно різні обробники.

На додаток до можливості використовувати об’єкти в процесі зіставлення зразків правил COOL підтримує гнучку систему запитів, що дозволяє використовувати задані користувачем критерії для вибірки деякого набору об’єктів і виконання над ним певних дій.

Запити дозволяють об’єднувати в набори об’єкти різних класів. Запити можна використовувати, наприклад, для перевірки існування того чи іншого набору об’єктів, виконання дій над набором або збереження посилання на набір для подальшого використання.

Порядок виконання практичного заняття

Приклад

Необхідно розробити ЕС «Звіт про проведені заходи в студентському гуртожитку».

1. Створити глобальні змінні, які відповідають номерам пунктів розробленого меню:

1) `(defglobal ?*DayAndMonthSearch* = 1)` – глобальна змінна «пошук за днем та місяцем»;

2) `(defglobal ?*MonthSearch* = 2)` – глобальна змінна «пошук тільки за місяцем»;

3) `(defglobal ?*ResponsibleSearch* = 3)` – глобальна змінна «пошук за відповідальною особою, яка повинна надати звіт після проведення заходу керівництву».

2. Розробити факти про проведення заходів з указанням назви заходів, відповідальної особи за їхнє проведення, дня тижня, календарного місяця та керівника, якому відповідальна особа має надати звіт (табл. 5.5).

Таблиця 5.5 – Дані, на підставі яких розроблялись факти ЕС

Назва заходу	Відповідальний	День	Місяць	Особа, що вимагає звіт
Збори блоків	Старости блоків	3	Травень	Голова студради
Загальні збори факультету	Голова студради	5	Травень	Декан
Суботник	Голова студради	29	Квітень	Комендант
Суботник	Голова студради	6	Травень	Комендант
Перевірка стану блоків	Голова студради	4	Травень	Комендант
Перевірка кімнат	Комендант	10	Травень	Декан
Обхід боржників	Голова студради	11	Травень	Комендант
Збір проблем	Голова студради	4	Травень	Комендант
Збір проблем	Голова студради	26	Травень	Декан
Прибирання гуртожитку	Заступник голови студради	15	Травень	Комендант
Дезінфекція приміщень	Заступник голови студради	1	Червень	Комендант

3. Розробити клас `Event` (захід), який є конкретним класом та має такі слоти:

1) `(slot eventName (create-accessor read-write) (override-message name-put))` – слот «Назва заходу»;

2) `(slot responsible (create-accessor read-write) (override-message responsible-put))` – слот «Відповідальна особа»;

3) `(slot day (create-accessor read-write) (override-message day-put))` – слот «День проведення заходу»;

4) `(slot month (create-accessor read-write) (override-message month-put))` – слот «Місяць проведення заходу»;

5) `(slot accountability (create-accessor read-write) (override-message accountability-put))` – слот «Особа, яка вимагає звіт».

4. Розробити обробники повідомлень для редагування значень слотів:

1) `(defmessage-handler Event name-put primary(?value) (bind ?self:eventName ?value));`

2) `(defmessage-handler Event responsible-put (?value) (bind ?self:responsible ?value));`

3) `(defmessage-handler Event day-put (?value) (bind ?self:day ?value));`

4) `(defmessage-handler Event month-put (?value) (bind ?self:month ?value));`

5) `(defmessage-handler Event accountability-put (?value) (bind ?self:accountability ?value)).`

5. Розробити обробник повідомлень для друку значень слотів об'єкта класу `Event`:

```
(defmessage-handler Event print ()
(printout t "Захід '" ?self:eventName "' відбудеться " ?self:day ?self:month ". Відповідальним назначено " ?self:responsible ". Звіт необхідно надати " ?self:accountability "." crlf)).
```

6. Розробити правило, що відображає меню програми та зчитує обраний користувачем пункт меню. Якщо введене значення є цілим числом, то цей факт меню додається у базу даних:

```
(defrule mainrule

    (initial-fact)
=>
    (printout t "Menu" crlf)
(printout t "1. Find event by day and month" crlf)
    (printout t "2. Find event by month" crlf)
    (printout t "3. Find event by responsible" crlf)
```

```
        (printout t crlf "Input menu choice: ")
        (bind ?menu (read))
    (if (integerp ?menu) then
        (assert (menu ?menu)))).
```

7. Розробити правило вибору способу пошуку заходу, у якому залежно від вибраного користувачем пункту меню буде виводитися відповідне питання:

```
(defrule menurule
        (menu ?m)
=>
        (switch ?m
        (case ?*DayAndMonthSearch* then
            (printout t crlf " Ввести день заходу: ")
            (bind ?d (read))
            (printout t crlf " Ввести місяць заходу:")
            (bind ?mon (read))
            (printout t crlf " Список заходів:" crlf crlf)
            (assert (gdaymon ?d ?mon)))
        (case ?*MonthSearch* then
            (printout t crlf " Ввести місяць заходу:")
            (bind ?mon (read))
            (printout t crlf " Список заходів:" crlf crlf)
            (assert (gmon ?mon)))
        (case ?*ResponsibleSearch* then
            (printout t crlf " Ввести відповідальну особу заходу: (Comendant/Chairman/Deputy chairman/Starosts of blocks): ")
            (bind ?r (read))
            (printout t crlf " Список заходів:" crlf crlf)
            (assert (gresponsible ?r)))))).
```

8. Розробити правила для пошуку заходу за різними комбінаціями даних:

```
(defrule findEventByDayAndMonth
            (gdaymon ?findday ?findmonth)
        (event ?evname ?res ?day ?month ?account)
=>
        (make-instance ev of Event (eventName ?evname) (responsible ?res) (day ?day) (month ?month) (accountability ?account))
        (if (and (eq ?findday ?day)(eq ?findmonth ?month)) then
        (send [ev] print))
```

```
    )

    (defrule findEventByMonth
        (gmon ?findmonth)
        (event ?evname ?res ?day ?month ?account)
    =>
        (if (eq ?findmonth ?month) then
        (make-instance ev of Event (eventName ?evname)
(responsible   ?res)   (day   ?day)   (month   ?month)
(accountability ?account))
        (send [ev] print))
    )

    (defrule findEventByResponsible
            (gresponsible ?findresponsible)
        (event ?evname ?res ?day ?month ?account)
    =>
        (if (eq ?findresponsible ?res) then
        (make-instance ev of Event (eventName ?evname)
(responsible   ?res)   (day   ?day)   (month   ?month)
(accountability ?account))
        (send [ev] print))
    ).
```

Повний лістинг програми наведено далі:

```
    (defglobal ?*DayAndMonthSearch* = 1)
    (defglobal ?*MonthSearch* = 2)
    (defglobal ?*ResponsibleSearch* = 3)

     (deffacts event
;Event      Name                    Responsible    Day  Month      Accountability
(event  "Block assembly"          Starosts of blocs  3     may       Chairman)
(event  " General Faculty Meeting "  Chairman     5     may       Decan)
(event  " Subbotnik "               Chairman     29    april     Comendant)
(event  " Subbotnik "               Chairman     6     may       Comendant)
(event " Checking the status of the block "  Chairman  4   may   Comendant)
(event " Checking the status of the block "  Comendan  10  may   Decan)
(event " Arrival of debtors "               Chairman  11  may   Comendant)
(event " Collecting problems "              Chairman  4   may   Comendant)
(event " Collecting problems "              Chairman  26  may   Decan)
(event " Cleaning of hostel "       Deputy chairman   15  may   Comendant)
(event " Disinfection of premises "  Deputy chairman   1   june  Comendant)
```

```
)

(defclass Event (is-a USER)
         (role concrete)
     (slot eventName   (create-accessor read-write)

; title of the event
     (override-message name-put))
     (slot responsible  (create-accessor read-write)
; responsible for holding the event
     (override-message responsible-put))
     (slot day  (create-accessor read-write)
; day of the event
     (override-message day-put))
     (slot month   (create-accessor read-write)
; month of the event
     (override-message month-put))
(slot accountability (create-accessor read-write)
; who to provide a report of the event
     (override-message accountability-put))
     (message-handler name-put)
     (message-handler responsible-put)
     (message-handler day-put)
     (message-handler month-put)
     (message-handler accountability-put)
     (message-handler print))

(defmessage-handler Event name-put primary   (?value)
; handler for editing the name slot value
     (bind ?self:eventName ?value)
)

(defmessage-handler Event responsible-put    (?value)
;  handler for editing the responsible slot value
         (bind ?self:responsible ?value)
)
(defmessage-handler Event day-put            (?value)
; handler for editing the slot value of the day
         (bind ?self:day ?value)
)

(defmessage-handler Event month-put          (?value)
; handler for editing the month slot value
         (bind ?self:month ?value)
)
```

```
(defmessage-handler Event accountability-put (?value)
; handler for editing the reporting slot value
        (bind ?self:accountability ?value))

; handler to print class object slots values Event
(defmessage-handler Event print ()

(printout t " Event '" ?self:eventName "' will take place "
?self:day ?self:month ". Responsible is appointed "
?self:responsible ". Report must be provided "
?self:accountability"."crlf)))

(defrule mainrule
    ; The rule displaying the program menu
    (initial-fact)
=>
    (printout t "Menu" crlf)
(printout t "1. Find event by day and month" crlf)
    (printout t "2. Find event by month" crlf)
    (printout t "3. Find event by responsible" crlf)
    (printout t crlf "Input menu choice: ")
    (bind ?menu (read))
    ; Enter a menu item
    (if (integerp ?menu) then
    ; If the input value is an integer, then
    (assert (menu ?menu))))
    ; add menu fact in the database;

(defrule menurule
; Rule for managing the menu
    (menu ?m)
=>
    (switch ?m

; If user choose day and month search
    (case ?*DayAndMonthSearch* then
    (printout t crlf " Enter the day of the event:")
        (bind ?d (read))
(printout t crlf " Enter the month of the event:")
        (bind ?mon (read))
    (printout t crlf " List of events:" crlf crlf)
        (assert (gdaymon ?d ?mon)))

; If user choosed to search only month of activity
```

```
        (case ?*MonthSearch* then
    (printout t crlf " Enter the month of the event:")
            (bind ?mon (read))
        (printout t crlf " List of events:" crlf crlf)
            (assert (gmon ?mon))
    )
```

; If user choosed by a responsible chief who should provide a report on the activity

```
        (case ?*ResponsibleSearch* then
            (printout t crlf " Enter the responsible
person of the event (Comendant/Chairman/Deputy
chairman/Starosts of blocs): ")
            (bind ?r (read))
        (printout t crlf " List of events:" crlf crlf)
            (assert (gresponsible ?r)))))

    (defrule findEventByDayAndMonth
    ; The rule for finding an event by day and month
        (gdaymon ?findday ?findmonth)
        (event ?evname ?res ?day ?month ?account)
    =>
        (make-instance ev of Event (eventName ?evname)
(responsible ?res) (day ?day) (month ?month)
(accountability ?account))
    (if (and (eq ?findday ?day)(eq ?findmonth ?month))
then
        (send [ev] print)))

    ; Use the print handler to display information
    (defrule findEventByMonth
    ; The rule of the search for the event by month
        (gmon ?findmonth)
        (event ?evname ?res ?day ?month ?account)
    =>
        (if (eq ?findmonth ?month) then
        (make-instance ev of Event (eventName ?evname)
(responsible ?res) (day ?day) (month ?month)
(accountability ?account))
        (send [ev] print))
    )
    ; Use the print handler to display information

    (defrule findEventByResponsible
    ; The rule of the search for the event on the responsible person
```

```
        (gresponsible ?findresponsible)
        (event ?evname ?res ?day ?month ?account)
    =>
        (if (eq ?findresponsible ?res) then
        (make-instance ev of Event (eventName ?evname)
(responsible ?res) (day ?day) (month ?month)
(accountability ?account))
        (send [ev] print))
    )
```

; *Use the print handler to display information.*

Результати роботи ЕС у вигляді звіту про проведені заходи в студентському гуртожитку наведено на рисунку 5.10.

```
CLIPS> (reset)
CLIPS> (run)
Menu
1. Find event by day and month
2. Find event by month
3. Find event by responsible

Input menu choice: 2

 Enter the month of the event:may

 List of events:

 Event ' Collecting problems ' will take place 26may. Responsible is appointed Chairman. Report must be provided Decan.
 Event ' Collecting problems ' will take place 4may. Responsible is appointed Chairman. Report must be provided Comendant.
 Event ' Arrival of debtors ' will take place 11may. Responsible is appointed Chairman. Report must be provided Comendant.
 Event ' Checking the status of the block ' will take place 10may. Responsible is appointed Comendant. Report must be prov
 Event ' Checking the status of the block ' will take place 4may. Responsible is appointed Chairman. Report must be provic
 Event ' Subbotnik ' will take place 6may. Responsible is appointed Chairman. Report must be provided Comendant.
 Event ' General Faculty Meeting ' will take place 5may. Responsible is appointed Chairman. Report must be provided Decan.
CLIPS>
```

Рисунок 5.10 – Рекомендації розробленої ЕС, які оформлені у вигляді звіту про проведені заходи в студентському гуртожитку

Контрольні питання

1. Дати визначення поняття «Експертна система».
2. З яких розділів складається розроблена ЕС «Звіт про проведені заходи в студентському гуртожитку»?
3. Охарактеризувати шаблони питань розробленої ЕС.
4. За допомогою якої функції задається користувачеві питання та отримується відповідь із заданого набору коректних відповідей? Описати алгоритм роботи цієї функції.
5. Описати необхідність створення модулів у CLIPS–програмах.
6. Які модулі запрограмовані в ЕС? Охарактеризувати їхнє призначення.

7. У чому виявляються об’єктно-орієнтовані можливості мови CLIPS під час побудови експертної системи?

8. Які глобальні змінні використані в експертній системі «Звіт про проведені заходи в студентському гуртожитку»?

9. Які факти використовуються в розробленій експертній системі, у чому їхнє призначення?

10. Які упорядковані чи неупорядковані факти використовуються в розробленій ЕС?

11. Які функції використано в ЕС для додавання нових фактів в ЕС?

12. Які правила використовуються в експертній системі «Звіт про проведені заходи в студентському гуртожитку», в чому їхнє призначення?

13. Як в ЕС задається пріоритет виконання правил?

14. Охарактеризувати класи в експертній системі «Звіт про проведені заходи в студентському гуртожитку», у чому їхнє призначення?

15. Які атрибути класів Вам відомі?

16. Охарактеризувати атрибути класів `import`, `export` та пояснити їхнє призначення в модулях.

17. Які типи обробників повідомлень є в експертній системі «Звіт про проведені заходи в студентському гуртожитку»?

18. Охарактеризувати основні частини експертної системи «Звіт про проведені заходи в студентському гуртожитку».

19. Як оргінізовано меню із вибору способу пошуку потрібної інформації в розробленій ЕС?

20. Пояснити призначення та декларування питань, якими оперує розроблена програма.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Джарратано Дж. Экспертные системы : принципы разработки и программирование : [пер. с англ.] / Дж. Джарратано, Г. Райли. – 4-е изд. – М. : ООО «И. Д. Вильмс», 2007. – 1152 с.
2. Articles on KurzweilAI.net written by Edward Feigenbaum [Електронний ресурс]. – Режим доступу : https://web.archive.org/web/20090311033516/http://www.kurzweilai.net:80/bios/frame.html?main=/bios/bio0019.html.
3. Официальный веб-узел CLIPS [Електронний ресурс]. – Режим доступу : http://www.ghg.net/clips/CLIPS.html.
4. Справочное руководство CLIPS Reference Manual [Електронний ресурс]. – Режим доступу : http://www.ghgcorp.com/clips/CLIPS.html.
5. CLIPS Reference Manual, Volume II, Advanced Programming Guide [Електронний ресурс]. – Режим доступу : http://clipsrules.sourceforge.net/documentation/v630/apg.pdf.
6. Частиков А. П. Разработка экспертных систем. Среда CLIPS / А. П. Частиков, Т. А. Гаврилова, Д. Л. Белов. – СПб. : БХВ – Петербург, 2003. – 608 с.
7. Forgy C. L. Rete : A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem / C. L. Forgy. // Artificial Intelligence. – 1985. – Vol. 19. – P. 17–37.
8. Уотермен Д. Руководство по экспертным системам : [пер. с англ.] / Д. Уотермен. – М. : Мир, 1989. – 537 с.
9. Гаврилова Т. А. Базы знаний интеллектуальных систем / Т. А. Гаврилова, В. Ф. Хорошевский. – СПб. : Питер, 2000. – 384 с.
10. Программирование на языке CLIPS [Електронний ресурс]. – Режим доступу : http://ryk-kypc2.narod.ru/.
11. CLIPS. Примеры программ [Електронний ресурс]. – Режим доступу : http://fkn.ktu10.com/?q=node/2183.
12. Татевосян А. С. Практикум по извлечению и структурированию знаний в среде CLIPS по дисциплине «Интеллектуальные информационные системы». / А. С. Татевосян. – Омск : СибАДИ, 2006. – 100 с. [Електронний ресурс]. – Режим доступу : http://5fan.ru/wievjob.php?id=23665.
13. Джексон П. Введение в экспертные системы / П. Джексон – М. : Вильямс, 2001. – 624 с.
14. Основные возможности ООП [Електронний ресурс]. – Режим доступу: https://studopedia.ru/6_130594_osnovnie-vozmozhnosti-oop.html.

Навчальне видання

НОВОЖИЛОВА Марина Володимирівна
ПЕТРОВА Олена Олександрівна

РОЗРОБКА ЕКСПЕРТНИХ СИСТЕМ В СЕРЕДОВИЩІ CLIPS

НАВЧАЛЬНИЙ ПОСІБНИК

Відповідальний за випуск *М. В. Новожилова*

Редактор *О. В. Михаленко*

Комп’ютерне верстання *О. О. Петрова*

Дизайн обкладинки *Т. А. Лазуренко*

Підп. до друку 27.12.2018. Формат 60×84/16.
Друк на ризографі. Ум. друк. арк. 5,5.
Тираж 50 пр. Зам. № .

Видавець і виготовлювач:
Харківський національний університет
міського господарства імені О. М. Бекетова,
вул. Маршала Бажанова, 17, Харків, 61002.
Електронна адреса: rectorat@kname.edu.ua
Свідоцтво суб’єкта видавничої справи:
ДК № 5328 від 11.04.2017.