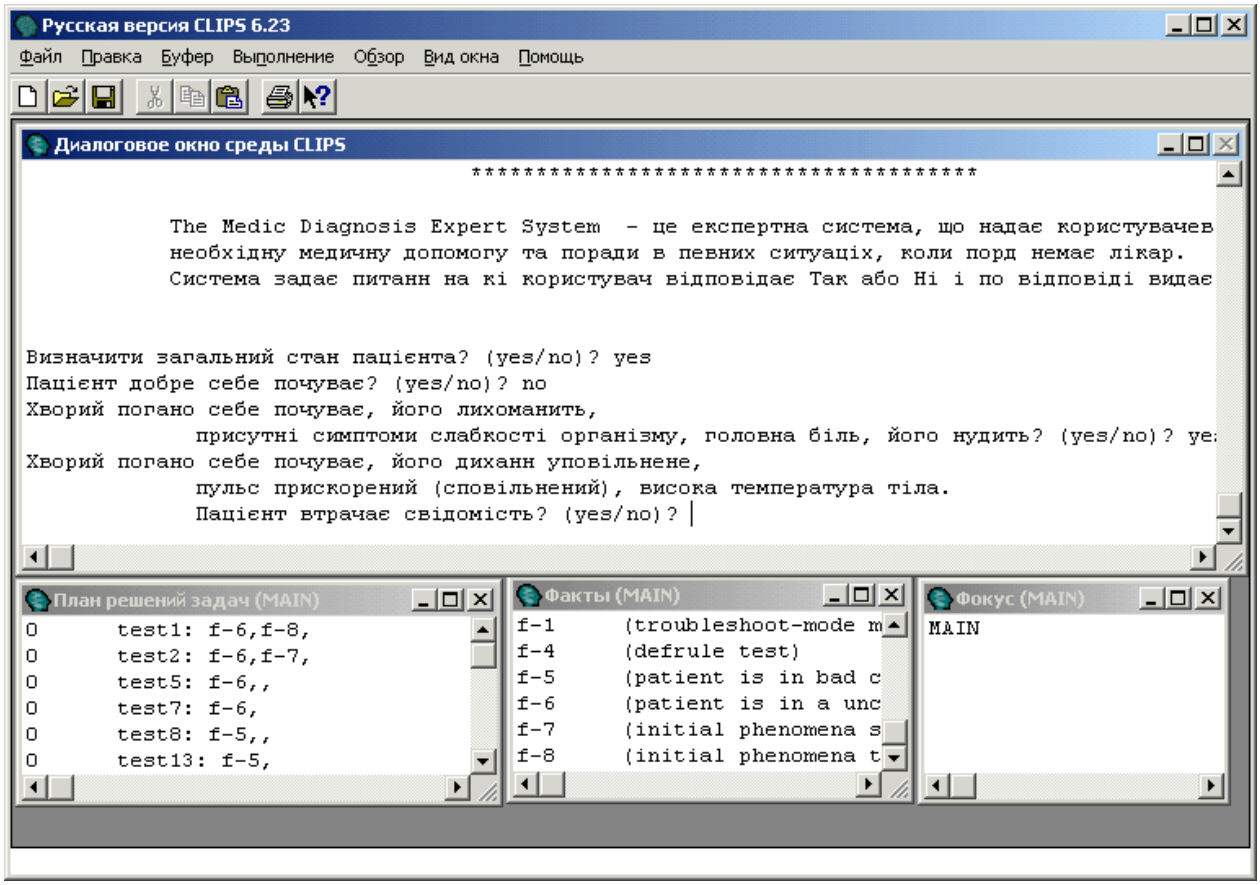
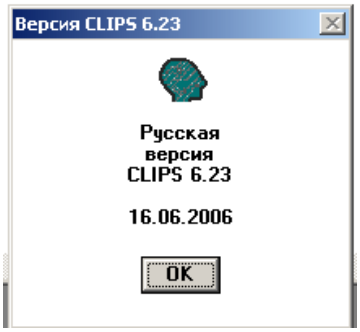
С.О. Семеріков, І.О. Теплицький

CLIPS:
локалізована оболонка експертної систем для вітчизняної системи освіти





Кривий Ріг

2006

Криворізький державний педагогічний університет
Кафедра інформатики та прикладної математики

С.О. Семеріков, І.О. Теплицький

CLIPS:
локалізована оболонка експертної систем для вітчизняної системи освіти

Кривий Ріг
Видавничий відділ КДПУ
2006

ЗМІСТ

ВСТУП

Для розробки експертних систем сьогодні найчастіше використовуються *оболонки експертних систем*, що дозволяють наповнювати базу знань та встановлювати правила виведення. Серед багатьох існуючих оболонок ЕС однією з найбільш популярних є оболонка CLIPS, що має понад 20 років історії розвитку та широку інсталяційну базу. Ця оболонка вигідно відрізняється від інших швидкістю та відкритістю, проте для не англомовного користувача її зручність обмежена неможливістю подання фактів та правил рідною мовою.

На подолання протиріччя між потенціалом оболонки ЕС CLIPS для розробки експертних систем (зокрема, в процесі навчання інтелектуальних систем) та відсутністю її локалізованого варіанту і спрямована дана робота.

Саме тому метою дослідження є створення локалізованого середовища CLIPS та розробка в ньому експертної системи рідною мовою.

Для реалізації поставленої мети необхідно було розв’язати наступні завдання:

1. Ознайомитись із синтаксисом CLIPS та вихідними текстами оболонки.
2. Локалізувати інтерфейс користувача, системні повідомлення та синтаксис.
3. Розробити експертні системи в локалізованому середовищі CLIPS.

Для вирішення поставленої задачі використовувалися так спеціальні методи, як аналіз вихідних текстів та програмний реінженерінг.

Практична значущість роботи полягає в локалізації інтерфейсу користувача оболонки CLIPS з метою поліпшення та покращення роботи з експертними системами, створення системи допомоги російською мовою.

I. CLIPS – інтерактивна продукційна оболонка експертної системи мовою Сі

1. Історія розвитку експертної системи CLIPS

Спочатку абревіатура CLIPS була назвою мови – C Language Integrated Production System (мова Сі, інтегрована із продукційними системами), зручної для розробки баз знань і макетів експертних систем. Мова CLIPS була розроблена в Центрі космічних досліджень NASA (NASA's Johnson Space Center) у 1984 році. Саме в цей час відділ штучного інтелекту (зараз Software Technology Branch) розробив багато прототипів експертних систем, які використовували сучасне програмне і технічне забезпечення. Але, не дивлячись на великий потенціал експертних систем, не багато із цих додатків дійшли до споживача. Це призвело до появи нових проблем: основні перешкоди створювала мова LISP, яка використовувалася як базова мова для розробки експертних систем. В зв'язку із цим відділ штучного інтелекту почав використовувати традиційні мови програмування, такі як С, для створення нових експертних систем.

Прототип оболонки CLIPS був створений у 1985 році. Синтаксис мови CLIPS був схожим на синтаксис експертної оболонки ART. Загальною метою прототипу CLIPS було створення мови, яка б могла вирішувати задачі, спираючись на концепцію знань (версія CLIPS 1.0). Після допоміжної розробки стало очевидно те, що CLIPS може стати дешевим інструментом для створення експертних систем, моделювання та навчання.

Подальше вдосконалення перетворило CLIPS в зручний інструмент для створення ефективних, функціональних експертних систем в різних сферах діяльності людини.

CLIPS версії 5.0 (1991 рік) включала в себе дві нові парадигми програмування: процедурне програмування (подібно мовам програмування С і Ada) і об'єктно-орієнтоване програмування (подібно мовам Common Lisp Object System – CLOS або Smalltalk). Об'єктно-орієнтоване програмування в CLIPS – CLIPS Object Oriented Language (COOL).

Систему CLIPS використовують різні організації, включаючи відділи NASA, військові відомства США, університети, компанії.

В систему CLIPS версії 6.1. (1998 рік) були добавлені декілька нових команд, а вихідний код CLIPS став сумісним із С++. Тепер для його компіляції можна використовувати будь-який ANSI C- або C++ компілятор.

Версія CLIPS 6.2 (31 березня 2002 р.) має вбудовані додатки, кращий інтерфейс для Windows-версії.

Розроблена нами російська версія CLIPS 6.23 може експлуатуватися на платформах UNIX, DOS, Windows і Macintosh. Сьогодні CLIPS являє собою сучасний інструмент, призначений для створення експертних систем. CLIPS складається з інтерактивного середовища – експертної оболонки зі своїм спо-

собом подання знань, гнучкої і могутньої мови і декількох допоміжних інструментів.

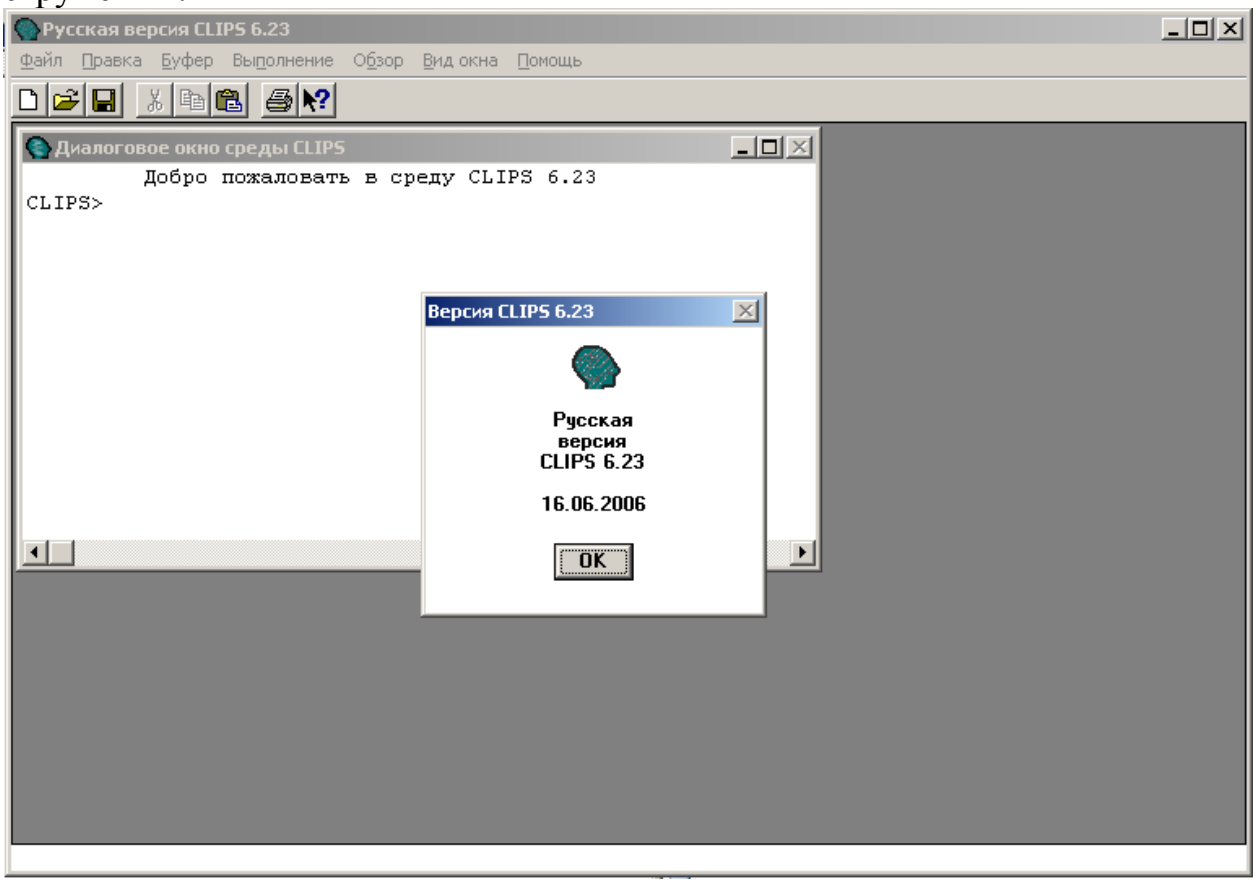


Рис. 2.1. Головне вікно CLIPS

2. Основи синтаксису CLIPS

Для демонстрації прикладів будемо використовувати російську Windows-версію CLIPS 6.23, що була розроблена і представлена в даній дипломній роботі. Зовнішній вид головного вікна CLIPS показано на рис. 2.1.

Російська Windows-версія середовища CLIPS цілком сумісна з базою специфікацією мови. Введення команд здійснюється безпосередньо в головне вікно CLIPS.

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

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

2) використанням інтерактивного віконного інтерфейсу CLIPS (наприклад, для версії Windows чи Macintosh);

3) за допомогою програм-оболонок, що реалізують свій інтерфейс спілкування з користувачем.

Крім того, CLIPS при запуску дозволяє виконувати командні файли власного формату (ця можливість так само доступна за допомогою команди *batch* чи меню *"Загрузка командного файла"*). Для реалізації цієї можливості необхідно запустити CLIPS (в нашому випадку CLIPSWin.exe) з одним із трьох наступних аргументів: *–f <ім'я файлу>*; *–f2 <ім'я файлу>*; *–l <ім'я*

файлу>.

Текстовий файл, заданий за допомогою команди *–f*, повинен містити команди CLIPS. Якщо заданий файл містить команду *exit*, то CLIPS завершить свою роботу і користувач повернеться в операційну систему. У випадку, якщо команда *exit* відсутня, то після виконання всіх команд із заданого файлу користувач побачить головне вікно CLIPS.

Опція *–f2* еквівалентна *–f*, але вона використовує команду *batch**. Файл, заданий цією опцією, також виконується після запуску CLIPS, але результати виконання команд не відображаються користувачеві на екран.

Опція *–l* задає текстовий файл, який містить конструктори CLIPS, які запускаються на виконання. Використання цієї опції еквівалентна команді *load*.

Основним методом спілкування з CLIPS є застосування командного рядка. Після появи в головному вікні CLIPS запрошення – *CLIPS*> – команди користувача можуть вводитися в середовище безпосередньо з клавіатури. Команди можуть бути викликами системних чи користувальницьких функцій, конструкторами різних даних CLIPS та т.п. У випадку виклику користувачем деякої функції вона негайно виконується, а результат її роботи відображається користувачу.

Для запуску програми наберіть свою програму у текстовому редакторі і збережіть файл з розширенням *.clp*, наприклад *ім'я файлу.clp*. Після цього запустіть CLIPS, якщо він уже був запущений, то визвіть команду *(clear)*. Завантажте створений файл за допомогою команди *load*, наприклад *(load "ім'я файлу.clp")*. Якщо файл був набраний без помилок, то ви повинні побачити повідомлення про успішну спробу завантаження файлу. Функція *load* повернула значення *TRUE*. Якщо це не так, то у синтаксисі визначень чи функцій правил була допущена помилка.

Для завантаження файлу також можна використовувати функцію *load**. У цьому випадку на екран не виводилася б інформація, що відбиває процес завантаження.

Для того щоб запустити експертну систему на виконання, необхідно виконати команду *reset* і команду *run*. Після цього система готова з вами співпрацювати. Для повторного запуску експертної системи необхідно ще раз виконати команди *reset* і *run*.

Існують певні правила для побудови синтаксису операцій:

1. Слово чи вираз, взяті в кутові дужки, називається *нетермінальним символом (наприклад, <string>)*. Нетермінальний символ вимагає подальшого визначення. Слово чи вираження, не включені в кутові дужки, називаються *термінальними символами*, і представляють синтаксис мови CLIPS.
2. Термінальні символи (особливо круглі дужки) повинні вводитися в командний рядок саме так, як показано у визначенні. Якщо за нетермінальними символами слідує символ *, то це означає, що в даному місці може знаходитися список, який включає нуль або більше елементів цього типу. Якщо ж за нетермінальним символом слідує +, то в даному

місці знаходитися список, який включає один або більше елементів цього типу.

3. Символи * і +, що зустрічаються самі по собі (не наступні після нетермінальних символів), є термінальними. Двокрапка, як горизонтальна, так і вертикальна, також використовується для відображення списку, який включає один або більше елементів.
4. Елементи, включені в квадратні дужки (*наприклад, [<коментарії>])*, є необов'язковими елементами, що можуть входити у визначення. Вертикальна риса, що розділяє два чи більш елементи визначення, вказує на те, що в конструкції необхідно використовувати один з перерахованих елементів.
5. Символ *::=* використовується для позначення необхідності заміни деякого нетермінального символу. Наприклад, визначення <*лексема*> *::=* <*символ*> | <*рядок*> позначає, що нетермінальний символ <*лексема*>, що зустрічається у визначенні, повинний бути замінений або на символ <*символ*>, або на символ <*рядок*>.
6. Пробіли, символи табуляції, переходи на інший рядок використовуються тільки для логічного поділу елементів визначення й ігноруються CLIPS (крім рядків, укладених у подвійні лапки).

2.1. Визначення фактів

Здатність експертної системи виконувати осмислені міркування, що спираються на закони логіки, є однією з найбільш дивовижних її особливостей. Такі міркування експертні системи можуть виконувати не гірше, ніж експерти. Проте для цього експерт повинен попередньо повідомити експертній системі факти та необхідні правила. Тільки після цього система формує базу знань, а це в свою чергу дає можливість користувачу спілкуватися з нею, при цьому отримувати певні рекомендації та висновки. Основою цього є зберігання в системі не тільки фактичної інформації, а й правил та функцій, яких у базі знань не було.

Відразу після запуску середовища CLIPS на виконання на екрані з'являється запрошення, яке сповіщає користувача, що він працює з інтерпретатором. У режимі інтерпретатора користувач може використовувати безліч команд: створювати нові факти, правила, описувати функції, використовувати конструктори, об'єкти CLIPS.

Факти – одна із основних форм подання інформації в CLIPS.

Кожен факт являє собою певний набір даних, що зберігається в поточному списку фактів – робочій пам'яті системи. Список фактів являє собою унікальне 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 файл і завантажувати список з раніше збереженого файлу.

У системі CLIPS фактом є список неподільних значень примітивних типів даних. CLIPS підтримує два типи фактів – упорядковані факти і неупорядковані факти чи шаблони. Посилатися на дані, які містяться у факті, мож-

на або використовуючи строго задану позицію значення в списку даних для упорядкованих фактів, або вказуючи ім'я значення для шаблонів.

Факти можна додавати, видаляти, змінювати і дублювати, уводячи відповідні команди з клавіатури, або з програми. Розглянемо опис деяких команд для роботи із фактами:

– за допомогою команди *assert* – (*assert <факт>+)*) можна добавляти факти у базу знань;

– видаляти із списку за допомогою команди *retract* – (*retract <визначення факту>+* | *);

– змінювати факти за допомогою команди *modify* – (*modify <визначення факту> <нове значення слоту>+*), або продублювати (команда *duplicate* – *(duplicate <визначення факту> <нове значення слоту>+)*);

– за допомогою конструктора *deftemplate* можна створити шаблон фактів: *(deftemplate <ім'я шаблону > [<необов'язковий коментар>][<визначення слоту>*])*;

– за допомогою конструктора *deffacts – (deffacts <ім'я списку фактів> [<необов'язковий коментар>][<факт>*])* – користувач визначає список фактів, які будуть автоматично добавлені після команди *reset*;

– факти можна зберігати та завантажувати. Для збереження фактів із поточного списку фактів в текстовий файл використовується команда *save-facts* (*save-facts <ім'я файлу>*). Для завантаження списку фактів в файл використовується команда *load-facts (load-facts <ім'я файлу>)*.

Однак після завантаження файлу факти не передаються відразу ж у базу фактів CLIPS. Власне завантаження виконується командою *reset:*

CLIPS> (reset)

Команда *reset* спочатку очищає базу фактів, а потім включає в неї факти з усіх раніше завантажених масивів. Вона також додає в базу єдиний системно визначений факт.

2.2. Визначення правил

Правила в CLIPS призначені для визначення евристик так званих "емпіричних правил", що визначають набір дій, які виконуються при виникненні деякої ситуації. Розробник експертної системи визначає набір правил, що разом працюють над розв'язанням деякої задачі. Правила складаються з передумов (Якщо) і наслідку (То). Передумови – це набір умов (чи умовних елементів), що повинні задовольнятися, для того щоб правило виконалося. А наслідки – це дії, які необхідно виконати.

Для додавання нових правил у базу знань CLIPS служить конструктор *defrule.* У загальному виді синтаксис даного конструктора можна представити так:

(defrule <ім'я правила> [<коментарі>]
[<визначення правила>]
<передумови> ; ліва частина правила
=>
<наслідок>) ; права частина правила

Ім’я правила повинне бути типу *symbol*. Коментарі є необов’язковими й описують призначення правила. Коментарі необхідно заключати в лапки.

Ліва частина правила задається набором умовних елементів, що звичайно складається з умов. Всі умови в лівій частині правила поєднуються за допомогою неявного логічного оператора *and*. Права частина правила містить список дій, виконуваних при активізації правила механізмом логічного висновку. Для поділу правої і лівої частини правил використовується символ =>. Дії правила виконуються послідовно, але тоді і тільки тоді, коли всі умовні елементи в лівій частині цього правила задовольняють іншим. Якщо в правій частині правила не визначена жодна дія, правило може бути активоване і виконане, але при цьому нічого не відбудеться.

CLIPS дозволяє завантажувати конструктори правил з текстового файлу. Для цього використовується команда: *(load < ім’я файлу>)*. Команда *load* відображає процес завантаження кожного конструктора. У випадку успішного завантаження команда повертає значення *TRUE*, у противному випадку – інформацію про помилку. CLIPS підтримує також команду *load*: (load* <ім’я файлу>)*. Ця команда цілком ідентична *load* за винятком того, що вона не відображає процес завантаження конструкторів.

CLIPS також дозволяє зберігати в текстовий файл усі конструктори, визначені в даний момент у системі. Для цього використовується команда *save*: *(save <ім’я-файлу>)*.

Для запуску CLIPS - програм використовується команда *run: (run)*.

2.3. Задання функцій

Функції можуть вводитися в діалогове вікно CLIPS з клавіатури або використовуватися в правилах, повідомленнях, визначених користувачем в функціях чи родових функціях.

CLIPS надає великий набір функцій, здатні задовольнити будь-які потреби користувача, серед яких логічні і математичні функції, функції роботи з рядками і складеними величинами, функції введення/виведення, процедурні функції, функції для роботи з методами родових функцій, функції для роботи з конструкторами deftemplate, deffacts, defrule, defglobal, deffunction, defgeneric, defmethod, defmodule.

Функцією в CLIPS називається частина коду, що має ім’я і повертає результат виконаної дії (наприклад, зображення інформації на екрані). Функції, що не повертають результат і при цьому виконують роботу, називаються командами. CLIPS оперує з декількома типами функцій – визначення користувачем *зовнішні функції, системні (внутрішні) функції,* функції, визначені в середовищі CLIPS за допомогою конструктора *deffunction* та *родові функції*.

Конструктор *deffunction* дозволяє визначати нові функції в CLIPS. Виклик функції здійснюється по імені, заданому користувачу. За ім’ям функції слідує список аргументів, відділений одним чи більш числом пробілів. Виклик функції разом зі списком аргументів повинен заключатися в дужки. Синтаксис конструктора *deffunction* містить у собі 5 елементів:

– ім’я функції;

– необов’язкові коментарі;

– список з нуля чи більше параметрів;

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

– послідовність дій чи виразів, що будуть виконані (обчислені) в момент виклику функції.

Такий синтаксис конструктора *deffunction* має вид:

(deffunction <ім’я-функції>
[<коментарі>]
<обов’язкові параметри>
[<групові-параметри>]
<дії>)
<обов’язкові-параметри> ::= <вираз просте поле>
<групові-параметри> ::= <виразскладене поле>

Родові функції визначаються за допомогою конструкторів *defgeneric* і *defmethod*. Родові функції дозволяють виконувати різні дії, в залежності від набору аргументів, заданих при виклику функції. Родові функції складаються з декількох компонентів, які називаються *методами*. Родова функція складається із заголовка і декількох методів (число яких може бути рівним нулю). Заголовок родової функції може бути або явно визначений користувачем, або не явно заданий.

Для створення заголовка родової функції служить конструктор *defgeneric*, а для створення кожного нового методу родової функції – конструктор *defmethod*.

Синтаксис конструктора *defgeneric* має вид:

(defgeneric <ім’я функції> [коментарі])

Синтаксис конструктора *defmethod*:

(defmethod < ім’я функції > [<індекс>] [<коментарі >]
(<обмеження параметру>* [<груповий параметр>]) <дія>*)
< обмеження-параметрів> ::= <проста-змінна> |
(<проста-змінна> <обмеженню-за-типом>*
[<обмеження-за-запитом>])
<груповий-параметр> ::= <складена-змінна> |
(<складена-змінна> <обмеження-за-типом>*
[<обмеження-за-запитом>])
<обмеження-за-типом> ::= <ім’я-класу>
<обмеження-за-запитом> ::= <глобальна-змінна> | <виклик-функції>

3. «Русская версия CLIPS»

3.1. Принципи локалізації системи

З метою локалізації CLIPS було необхідно:

– локалізувати інтерфейс користувача;

– локалізувати файли допомоги;

– локалізувати ядро системи.

З метою локалізації інтерфейсу користувача були змінені наступні програмні ресурси:

1) діалоги;
2) меню;
3) таблиці рядків;
4) “гарячі” клавіші;
5) шрифти.

Для локалізації файлів допомоги (рис. 2.2) було виконано їх переклад російською мовою з компіляцією у Help Workshop.

Рис. 2.2. Головний файл допомоги

Дійсною проблемою при локалізації завжди були операції з різними наборами символів. Наприклад, існують такі мови і системи писемності, наприклад, японські ієрогліфи, у яких стільки знаків, що одного байта, що дозволяє кодувати не більше 256 символів, просто недостатньо. Для підтримки подібних мов були створені двобайтові набори символів (символ представляється або одним, або двома байтами) (*double-byte character sets*, *DBCS*).

Компілятор ресурсів генерує двійове представлення всіх ресурсів, використовуваних програмою. Рядки в ресурсах (таблиці рядків, шаблони діалогових вікон, меню й ін.) завжди записуються в *Unicode*.

Unicode – стандарт, спочатку розроблений Apple і Xerox у 1988 р. У 1991 р. був створений консорціум для удосконалювання і впровадження Unicode. У нього ввійшли компанії Apple, Compaq, Hewlett-Packard, IBM, Microsoft, Oracle, Silicon Graphics, Sybase, Unisys і Xerox і ін. Ця група компаній спостерігає за дотриманням стандарту Unicode.

Unicode – це унікальний код для будь-якого символу, незалежно від платформи, незалежно від програми, незалежно від мови. Система Unicode присвоює унікальний код будь-якому символу, незалежно від платформи, незалежно від програми, незалежно від мови. Саме ця схема кодування використовується такими сучасними технологіями і стандартами, як наприклад XML, Java, ECMAScript (JavaScript), LDAP, CORBA 3.0, WML і так далі. Саме Unicode є офіційною схемою реалізації ISO/IEC 10646. Нарешті, це кодування підтримується в багатьох операційних системах, в усіх сучасних браузерах Інтернет і у великій кількості інших програм.

Для того, щоб почати роботу з кодуванням Unicode, необхідно:

– знати, що текстові рядки – це масиви символів, а не масиви байтів чи значень типу char;

– використовувати універсальні типи даних (*TCHAR* чи *PTSTR*) для текстових символів і рядків;

– використовувати явні типи даних (*BYTE* чи *PBYTE*) для байтів, покажчиків на байти і буферів даних;

– застосовувати макрос *_TEXT* для визначення символьних і строкових літералів;

– передбачати можливість глобальних замін (наприклад, *PSTR* на *PTSTR*);

– модифікувати логіку строкової арифметики. Наприклад, функції звичайно приймають розмір буфера в символах, а не в байтах. Це значить, що замість *sizeof (szBuffer)* необхідно передавати *(sizeof (szBuffer) / sizeof (TCHAR))*. Але блок пам'яті для рядка відомої довжини виділяється в байтах, а не в символах, тобто замість *malloc (nCharacters)* потрібно використовувати *malloc (nCbaracters * sizeof (TCHAR))*.

У Windows є набір функцій для роботи з Unicode-строками. Ці функції перераховані в табл. 2.1.

Таблиця 2.1. Набір функцій для роботи з Unicode-рядками

Функція	*Опис*
lstrcat	Виконує конкатенацію рядків

lstrcmp	Порівнює два рядки з обліком регістра букв
lstrcmpi	Порівнює два рядки без обліку регістра букв
lstrcpy	Копіює рядок в другу ділянку пам'яті
lstrlen	Повертає довжину рядка в символах

Вони реалізовані як макроси, що викликають або *Unicode*-, або *ANSI*-версію функції в залежності від того, чи визначений *UNICODE* при компіляції вихідного модуля. Наприклад, якщо *UNICODE* не визначений, *lstrcat* розкривається *в lstrcatA*, визначений – у *lstrcatW*.

Рядкові функції *lstrcmpi*, *lstrcmpi* поводяться не так, як їхні аналоги з бібліотеки С *(strcmp, strcmpi, wcscmp і wcscmpf)*, що просто порівнюють кодові позиції в символах рядків. Ігноруючи фактичні символи, вони порівнюють числове значення кожного символу першого рядка з числовим значенням символу другого рядка. Але *lstrcmp* і *lstrcmpi* реалізовані через виклики Windows-функції *CompareString:*

```
int CompareString(LCID lcid, DWORD fdwStyle, PCWSTR pString1, int cch1,
PCWSTR pString2, int cch2);
```

Вона порівнює два Unicode-рядки. Перший параметр задає так званий ідентифікатор локалізації *(locale ID, LCI)* – 32-бітне значення, що визначає конкретну мову. За допомогою цього ідентифікатора *CompareString* порівнює рядок з урахуванням значення конкретних символів у даній мові. Коли кожна з функцій сімейства *lstrcmp* викликає *CompareString*, у першому параметрі передається результат виклику Windows-функции *GetThreadLocale:LCIGetThreadLocale()*. Вона повертає вже згаданий ідентифікатор, що призначається потоку в момент його створення.

Другий параметр функції *CompareString* вказує флаги, що модифікують метод порівняння рядків. Припустимі флаги перераховані в наступній таблиці 2.2.

Таблиця 2.2.

Флаг	*Дія*
NORM_IGNORECASE	Різниця в регістрі букв ігноруються
NORM_IGNOREKANATYPE	Розходження між знаками хірагани та катакани ігноруються
NORM_IGNORENONSPACE	Знаки, відмінні від пробілів, ігноруються
NORM_IGNORESYMBOLS	Символи, відмінні від алфавітно-цифрових, ігноруються
NORM_IGNOREWIDTH	Різниця між одно- і двухбайтним представленням одного і того ж символу ігнорується
SORT_STRINGSORT	Розділові знаки обробляються так само, як і символи, відмінні від алфавітно-цифрових

Викликаючи *CompareString*, функція *lstrcmp* передає в параметрі *fdwStyle* нуль, а *lstrcmpi* – флаг *NORM_IGNORECASE*. Інші чотири параметри визначають два рядки і їхню довжину. Якщо cch1 дорівнює -1, функція вважає, що рядок pString2 завершується нульовим символом, і автоматично обчислює її довжину. Те ж відноситься і до параметрів *cch2 wpString2*.

Багато функцій С-бібліотеки з *Unicode*-рядками не працюють. Так,

tolower і toupper неправильно перетворюють регістр букв зі знаками наголосу. Тому для *Unicode*-рядків краще використовувати відповідні Windows-функції. До того ж вони коректно працюють із *ANSI*-рядками.

Перші дві функції:

PTSTRCharLower(PTSTRpszStnng);

PTSTRCharUpper(PTSTRpszString);

Перетворюють або окремий символ, або цілий рядок з нульовим символом в кінці. Щоб перетворити весь рядок, просто передайте йому адресу. Але, щоб перетворити окремий символ, його треба передати так:

TCHAR cLowerCaseCnr=CharLower((PTSTR)szString("0"));

Приведення типу окремого символу до *PTSTR* викликає обнуління старших 16 біт переданого значення, а в його молодші 16 біт міститься самий символ. Знайшовши, що старші 16 біт цього значення рівні 0, функція зрозуміє, що перетворити треба не рядок, а окремий символ. 32-бітне значення, що повертається, містить результат перетворення в молодших 16 бітах.

Наступні дві функції аналогічні двом попереднім за винятком того, що вони перетворюють символи, що містяться в буфері (який не потрібно завершувати нульовим символом):

DWORD CharLowerBuff (PTSTRpszString, DWORD cchString);

DWORD CharUppprRuff (PTSTRpszString, DWORD cchString);

Інші функції бібліотеки С (наприклад, *isalpha, islower, isupper*) повертають значення, що повідомляє, чи є даний символ буквою, а також рядкова вона чи прописна. У Windows API теж є подібні функції, але вони враховують і мову, яка вибрана користувачем у *ControlPanel*:

BOOL IsCharAlpha (TCHAR ch);

BOOL IsCharAlphaNumeric (TCHAR ch);

BOOL IsCharLower (TCHAR oh);

BOOL IsCharUpper (TCHAR ch);

І остання група функцій з бібліотеки С – *printf*. Якщо при компіляції *_UNICODE* визначений, вони очікують передачі всіх символьних і рядкових параметрів у *Unicode*; в іншому випадку – у *ANSI*. Microsoft ввела в сімейство функцій printf своєї С-бібліотеки додаткові типи полів, частина з яких не підтримується в *ANSI C*. Вони дозволяють легко порівнювати і змішувати символи і рядки з різним кодуванням. Також розширена функція *wsprintf* операційної системи. От кілька прикладів (зверніть увагу на використання букви *s* у верхньому і нижньому регістрі):

```
char szA [100]; //строковий буфер в ANSI
WCHAR szA [100]; //строковий буфер у Unicode
//звичайний виклик sprintf: усі рядки в ANSI
sprintf (szA,"%s","ANSI Str");
//перетворимо рядок з Unicode у ANSI
sprintf (szA,"%S","Unicode Str");
//звичайний виклик wsprintf всі рядки в Unicode
wsprintf (szW,L"%s",L"Unicode Str");
//перетворимо рядок з ANSI в Unicode
```

wsprintf (szW,L"%S","ANSI Str").

Для того, щоб встановити мову програми, яка впливає на використання функцій класифікації символів, на початку програми необхідно додати *setlocale(LC_CTYPE, "Russian_Russia.1251");*

3.2. Локалізований інтерфейс користувача

При запуску експертної системи CLIPS появляється основне діалогове вікно програми (рис. 2.3).

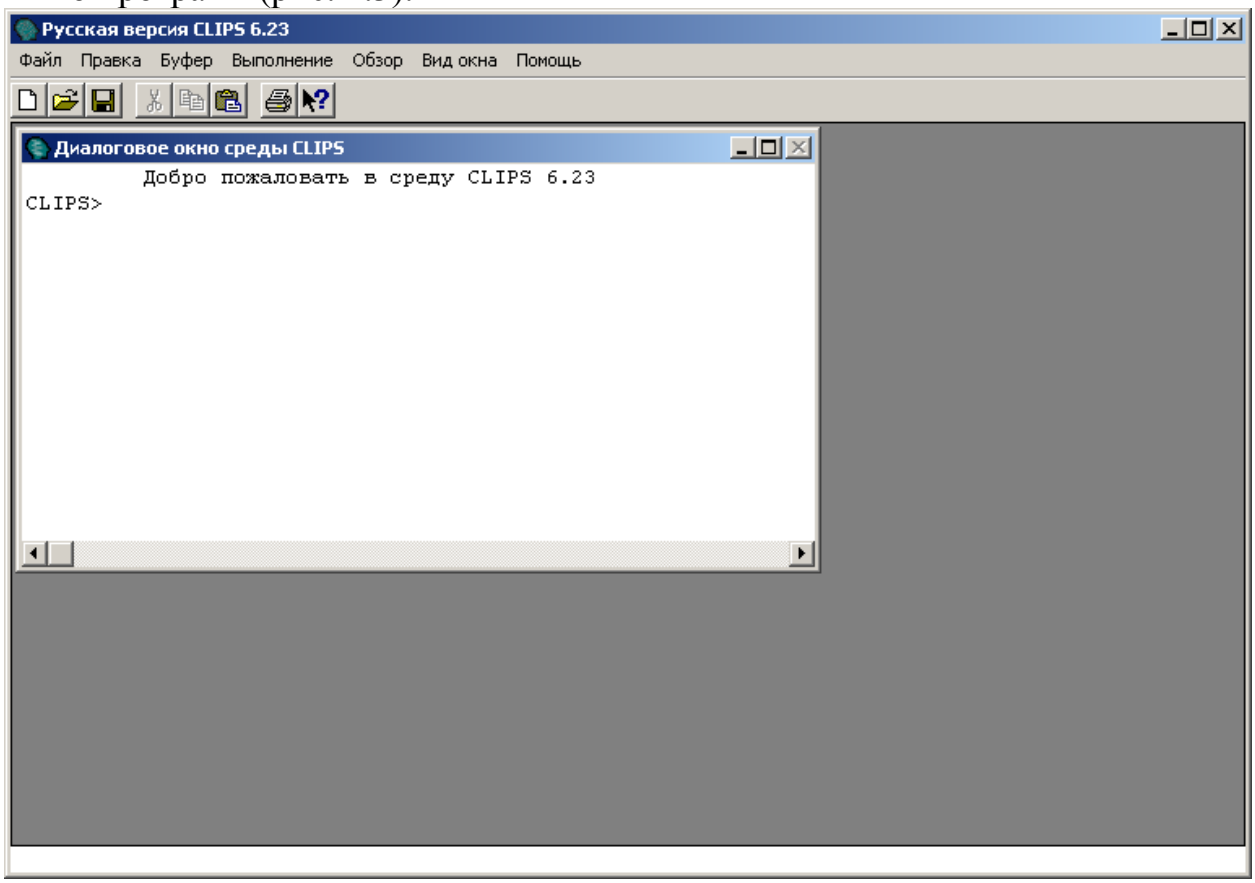


Рис. 2.3. Діалогове вікно CLIPS

Меню “Файл” (рис. 2.4) використовується для роботи з файлами даних:
– команда *Создать* – служить для створення нового документу;
– команда *Открыть* – команда відкриває існуючий файл;
– команда *Загрузка* – команда завантажує конструктори із текстового файлу;
– команда *Загрузка командного файла* – завантаження командного файлу;
– команда *Загрузка бинарного файла* – завантаження бінарного файлу;
– команда *Загрузка текстового файла* – виконує завантаження текстового файлу в діалогове вікно CLIPS;
– команда *Закрыть* – служить для зачинення документу;
– команда *Сохранить* – служить для поточного запису конструкторів у текстовий файл;
– команда *Сохранить как* – служить для запису текстового файлу під новим іменем;
– команда *Сохранить бинарный файл* – служить для запису бінарного файлу;
– команда *Возврат* – вертає останню збережену версію файлу;

– команда *Параметры страницы* – змінює опції друку;
– команда *Печать* – друкує документ;
– команда *Выход* – вихід із програми.

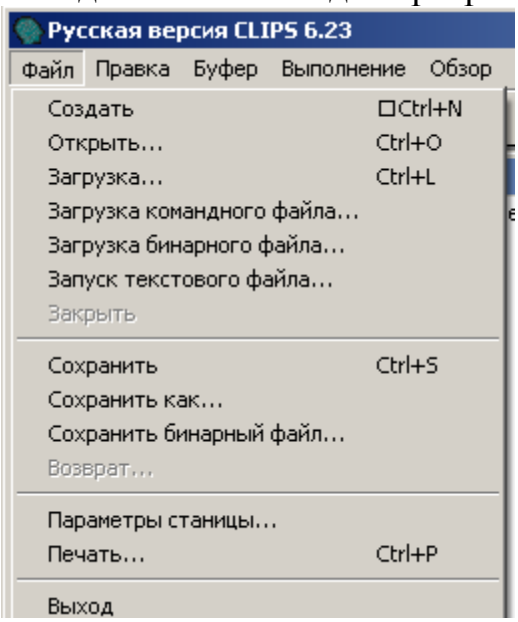


Рис. 2.4. Меню “Файл”

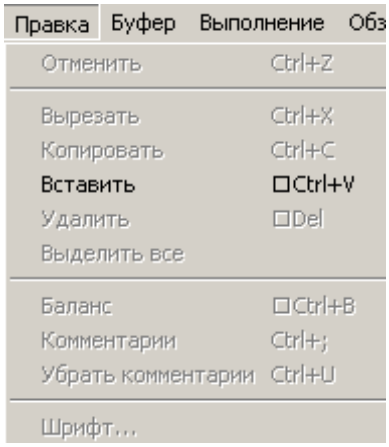


Рис. 2.5. Меню “Правка”

Меню “Правка” (рис. 2.5) використовується для редагування текстів програми:
– команда *Отменить* – служить для відміни дій;
– команда *Вырезать* – вирізає виділені області тексту і вміщує їх в буфер обміну;
– команда *Копировать* – слугує для копіювання тексту;
– команда *Вставить* – вставка із буферу текстів програми;
– команда *Удалить* – видаляє виділений текст;
– команда *Выделить все* – служить для виділення всього тексту;
– команда *Баланс* – вибір тексту між скобками;
– команда *Комментарии* – служить для встановлення коментарів після кожного символу рядка;
– команда *Убрать комментарии* – слугує для видалення коментарів;
– команда *Шрифт* – служить для вибору шрифту.

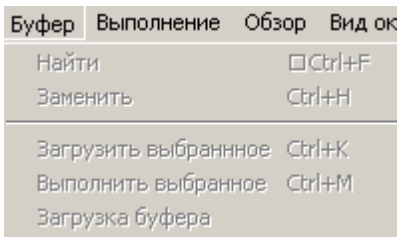


Рис. 2.6. Меню “Буфер”

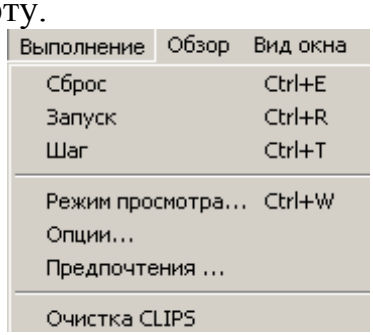


Рис. 2.7. Меню “Выполнение”

Меню “Буфер” (рис. 2.6) використовується для заповнення буфера, пошуку інформації:

– команда *Найти* – служить для пошуку тексту;
– команда *Заменить* – замінює старий текст новим;
– команда *Загрузить выбранное* – служить для завантаження установок;
– команда *Выполнить выбранное* – здійснює установлені завантаження;
– команда *Загрузка буфера* – служить для завантаження буферу.

Меню “Выполнение” (рис. 2.7) використовується для компіляції даних, дозволяє встановлювати опції, очищає середовище:
– команда *Сброс* – служить для оновлення середовища;
– команда *Запуск* – служить для запуску програми на виконання;
– команда *Шаг* – покрокове виконання програми;
– команда *Режим просмотра* – дозволяє переглядати списки елементів середовища;
– команда *Опции* – дозволяє змінювати опції;
– команда *Предпочтения* – змінює всі дані;
– команда *Очистка CLIPS* – служить для очистки робочої пам’яті системи.

Меню “Обзор” (рис. 2.8) дозволяє переглядати результати виконання дій правил, фактів, шаблонів, об’єктів, глобальних змінних, родових функцій та ін.:
– команда *Модуль* – включає в себе функцію main;
– команда *Менеджер правил* – служить для виконання правил;
– команда *Менеджер фактов* – служить для виконання фактів;
– команда *Менеджер шаблонов* – служить для виконання шаблонів;
– команда *Менеджер функций* – служить для виконання функцій;
– команда *Менеджер глобальных переменных* – служить для виконання глобальних змінних;
– команда *Менеджер классов* – служить для виконання певних класів;
– команда *Менеджер предопределенных объектов* – служить для виконання об’єктів;
– команда *Менеджер плана решений задачи* – служить для виконання певних дій над активними правилами.

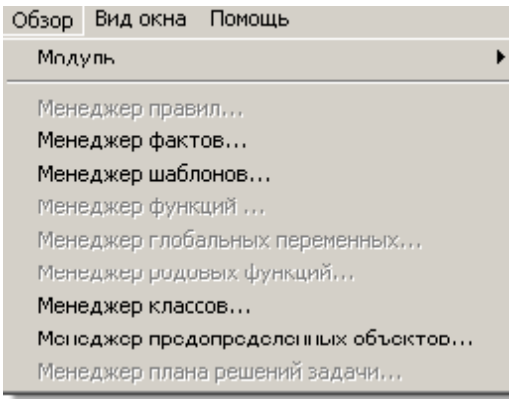


Рис. 2.8. Меню “Обзор”

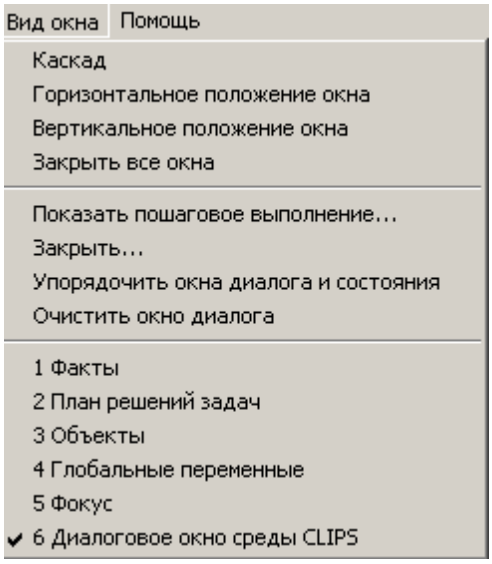


Рис. 2.9. Меню “Вид окна”

Меню “Вид окна” (рис. 2.9) використовується для зміни положення вікна середовища:
– команда *Каскад* – розміщує вікна каскадом;
– команда *Горизонтальное положение окна* – розміщує вікна у горизонтальному положенні;
– команда *Вертикальное положение окна* – розміщує вікна у вертикальному положенні;
– команда *Закрыть все окна* – зачиняє всі активні вікна середовища;
– команда *Пошаговое выполнение* – служить для показу покрокового виконання програми;
– команда *Закрыть* – зачиняє всі вікна покрокового виконання;
– команда *Упорядочить окна диалога и состояния* – впорядковує всі вікна середовища;
– команда *Очистить окна диалога* – очищає діалогове вікно від записів.

Меню “Помощь” (рис. 2.10) використовується для отримання довідкової інформацію про роботу програми:
– команда *Версия CLIPS* – виводить користувачеві інформацію про версію CLIPS;
– команда *Справка по CLIPS* – виводить користувачеві необхідну та додаткову інформацію про CLIPS;
– команда *Пример экспертной системы* – виводить приклад експертної системи з коментарями;
– команда *Пример экспертной системы 2* – виводить приклад експертної системи з коментарями;
– команда *Строка состояния* – показує додаткову інформацію.

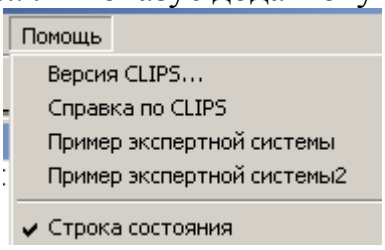


Рис. 2.10. Меню “Помощь”

4. Приклади експертних систем в локалізованому середовищі CLIPS

4.1. Експертна система “The Medic Diagnosis Expert System”

В даному розділі приведена програма експертної системи з детальними коментарями. Приведена нижче експертна система здатна діагностувати загальний стан хворого, визначати типи захворювання і надавати користувачу рекомендації з надання першої медичної допомоги.

Функція *ask-question* задає користувачу питання, отримане у *?question*, і отримує від користувача відповідь, що належить списку припустимих відповідей, заданому в *$?allowed-values*:

```
(deffunction ask-question (?question $?allowed-values)
(printout t ?question) (bind ?answer (read))
(if (lexemep ?answer)
then
(bind ?answer (lowcase ?answer)))
(while (not (member ?answer ?allowed-values)) do
(printout t ?question) (bind ?answer (read))
(if (lexemep ?answer)
then
(bind ?answer (lowcase ?answer)))) ?answer)
```

Функція yes-or-no-p задає користувачу питання і чекає на відповідь. У випадку позитивної відповіді функція повертає значення TRUE, інакше – FELSE:

```
(deffunction yes-or-no-p (?question)
(bind ?response (ask-question ?question yes  no  y n))
(if (or (eq ?response yes) (eq ?response y))
Then TRUE else FALSE))
```

Правило “begin” визначає поточний загальний стан хворого по відповідях від користувача. (not (help ?)) – гарантує, що діагноз ще не поставлений, а умовний елемент (patient is in?)) гарантує, що загальний стан хворого ще не визначений. Якщо це так, то користувачу задаються запитання й в систему додаються факти, що описують загальний стан хворого. Це такі факти як (patient is in a normal condition) – у хворого немає скарг, тобто він здоровий; (patient is in bad condition) хворий погано себе почуває і йому потрібна допомога; (patient is in a uncertain condition) – стан хворого невизначено:

```
(defrule begin ""
            (not (patient is in?)) (not (help ?))
             =>
          (if (yes-or-no-p "Визначити загальний стан пацієнта? (yes/no)? ")
           then
             (if (yes-or-no-p "Пацієнт добре себе почуває? (yes/no)? ")
             then
             (assert (patient is in consciousness))
             else
             (assert (patient is in bad condition)) (assert (patient is in
a uncertain condition)))
             else
             (assert (patient is in a normal condition))))
```

Правило test виконується у випадку, якщо загальний стан хворого визначений і відомо, що він погано себе почуває. Крім того, умовний елемент (not (patient is in?)) гарантує, що це правило ще не викликалося. В залежності від тієї чи іншої відповіді користувача правило додає відповідний набір фактів – це (initial phenomena sunstroke) сонячний удар; (initial phenomena thermal impact) тепловий удар; (initial phenomena electrotrauma) електротравма; (initial phenomena poisoning) різні отруєння:

```
(defrule  test ""
             (patient is in a uncertain condition)
             (not (initial phenomena ?))
             (not (help ?))
             =>
             (if ( yes-or-no-p "Хворий погано себе почуває, його лихома-
нить, присутні симптоми слабкості організму, головна біль, його нудить?
(yes/no)? ")
```

```
            then
            (assert (initial phenomena sunstroke))
            (assert (initial phenomena thermal impact))
            else
            (assert (initial phenomena electrotrauma))
            (assert (initial phenomena poisoning))))
```

Правило test1 за відповіддю користувача визначає тип захворювання. У випадку, якщо користувач підтверджує такий діагноз хворого, то система видає коротку і головну медичну допомогу. З появою повідомлення виконання діагностичних правил припиняється:

```
(defrule test1 ""
            (patient is in a uncertain condition)
            (initial phenomena thermal impact)
            (not (help ?))
            =>
            (if ( yes-or-no-p "Хворий погано себе почуває, його дихання
уповільнене, пульс прискорений (сповільнений), висока температура тіла. Па-
цієнт втрачає свідомість? (yes/no)? ")
            then
            (assert (help "Напевно це тепловий удар. Перша допомога: як
найшвидше збити  температуру тіла. Пацієнта необхідно перемістити у примі-
щення, де немає прямих  сонячних променів, звільнити від верхнього одягу.
По необхідності дати прохолодної  питної води, або прикласти до потилиці
лід."))))
```

Правила test2 – test4 системи задають користувачу певне питання, і за відповіддю визначають захворювання, чи це може бути сонячний удар, чи тепловий удар, а можливо поразка електричним струмом, або пацієнт просто отруївся. Після цього система повідомляє користувачу, яку медичну допомогу потрібно надати:

```
(defrule test2 ""
           (patient is in a uncertain condition)
           (initial phenomena sunstroke)
           (not (help ?))
            =>
            (if ( yes-or-no-p "У хворого слабкий зір, запаморочення, його
нудить, почервоніння шкіри,
висока температура тіла. У хворого невиразні рухи, нечітка вимова?
(yes/no)? ")
             then
            (assert (help "Можливо у пацієнта сонячний удар. Перша допомо-
га: Знизити температуру тіла, пацієнта необхідно перемістити у приміщення,
де немає прямих  сонячних променів, звільнити від верхнього одягу. По необ-
хідності дати прохолодної  питної води, або прикласти до потилиці лід"))))

(defrule test3 ""
            (patient is in a uncertain condition)
            (initial phenomena poisoning)
            (not (help ?))
            =>
            (if ( yes-or-no-p "Хворий не досить добре себе почуває, тобто
його нудить, болить голова. Колір шкіри  і обличчя зеленкуватого кольору,
пульс прискорений, температура тіла висока,  тиск поза нормою? (yes/no)? ")
               then
            (assert (help "Більш всього - це отруєння. Перша допомога:
дати кисневу маску, необхідно промити шлунок, дати випити теплої води."))))
```

```
(defrule test4 ""
             (patient is in a uncertain condition)
             (initial phenomena electrotrauma)
             (not (help ?))
             =>
            (if ( yes-or-no-p "У хворого судоми кінцівок чи м'язів, ушко-
джена центральна нервова система? (yes/no)? ")
             then
            (assert (help "Перша допомога: необхідно швидко доставити хворо-
го до лікарні."))))
```

Правило test5, крім визначення стану хворого, також застосовується для додавання в систему фактів, що описують загальний стан хворого, що можуть бути використані іншими правилами. Це такі факти, як (patient is in faint) – хворий знепритомнів і (patient is in not faint) – хворий усвідомлює свої вчинки та дії, а умовний елемент (not (patient is in?) гарантує, що це правило ще не викликалося:

```
(defrule test5 ""
           (patient is in a uncertain condition)
           (not (patient is in ?))
           (not (help ?))
           =>
          (if (yes-or-no-p "Пацієнту важко дихати або дихання не прослухо-
вується, пульс сповільнений, тахікардія? (yes/no)? ")
           then
           (assert (patient is in faint))
           else
           (assert (patient is in not faint))))
```

Правило test6 за відповіддю користувача визначає поточний стан хворого:

```
(defrule test6 ""
             (patient is in a uncertain condition)
             (patient is in faint)
             (not (help ?))
              =>
             (if ( yes-or-no-p "Скарги на слабкість, важко дихати. Пацієнт
блідий,
 часто втрачає свідомість, рухи обмежені, пульс нерівномірний, зіниці роз-
ширені? (yes/no)? ")
              then
              (assert (help "Можливо у пацієнта відкрита рана. Перша медична
допомога: зупинити кровотечу і направити хворого до лікарні "))
              else
              (if ( yes-or-no-p "Пацієнт знаходиться в стані клінічної смер-
ті? (yes/no)? ")
              then
              (assert (help " Перша допомога: зробити штучне дихання, масаж
серця, надіти кисневу маску.
Негайно доставити потерпілого до лікарні.")))
              (assert (patient is in not faint))))
```

Правило test7 намагається визначити подальший стан хворого за відповідями користувача, і видає першу медичну допомогу. На цьому подальше виконання правил припиняється:

```
(defrule test7 ""
             (patient is in a uncertain condition)
             (not (help ?))
```

```
            =>
            (if (yes-or-no-p "Пацієнт скаржиться на гострий біль при русі у
нозі (руці), не може зігнути руки чи ноги, виявлені набряки та синці. Функ-
ції кінцівок порушені? (yes/no)? ")
            then
            (assert (help "Можливо - це вивих. Перша допомога: вправити кі-
стку,
накласти пов'язку, звернутися до лікаря."))
            else
            (assert (help "Можливо - це забите місце. Перша допомога: прик-
ласти до забитого місця лід,
накласти пов'язку. ")))
```

Якщо в правилі test7 користувач на питання відповів “НІ”, то система приступає до виконання правила test8:

```
(defrule test8 ""
            (patient is in bad condition)
            (not (traumas of a stomach | traumas of a skull))
            (not (help ?))
            =>
            (if (yes-or-no-p "Чи є травми живота?  (yes/no)? ")
            then
            (assert (traumas of a stomach))
            else
            (assert (traumas of a skull))))
```

Правила test9, test10 визначають за відповіддю користувача, які травми має хворий. Травми можуть бути трьох типів (у нашому випадку): закриті травми живота, поранення живота, травми грудної клітки. У двох останніх випадках користувачу видаються відповідні рекомендації. Якщо користувач вибрав перший випадок, система переходить до наступного правила і видає повідомлення про надання медичної допомоги:

```
(defrule test9 ""
            (or (and (patient is in bad condition) (patient is in not
faint)) (traumas of a stomach))
            (not (help ?))
             =>
            (bind ?response (ask-question "Які травми має пацієнт
( 1 закриті травми живота /2 поранення в живіт /3 травми грудної клітки )
?" 1 2 3 ))
            (if (eq ?response 2)
            then
            (assert (help "Перша медична допомога: Захист рани від інфек-
цій. Дати знеболювальне. Якщо є температура, намагатися її знизити. Зупини-
ти кровотечу. Хворого негайно госпіталізувати. "))
            else
            (if (eq ?response 3)
            then
            (assert (help "Хворого негайно госпіталізувати."))))))
(defrule test10 ""
            (patient is in bad condition)
            (patient is in not faint)
            (traumas of a stomach)
            (not (help ?))
              =>
             (if (yes-or-no-p "У хворого присутні такі симптоми як біль в
животі, нудота, запаморочення, слабкість, висока температура, хворий часто
втрачає свідомість? (yes/no)? ")
```

```
            then
            (assert (help "Захист рани від інфекцій, перев'язати рану, зупинити кровотечу. Негайно госпіталізувати хворого. "))))
```

Правила test11 і test12 по відповідях користувача визначає, чи є в хворого переломи чи травми черепа. Якщо ні, то система переходить до реалізації правил test13 і test14. Якщо в пацієнта присутні травми (переломи) черепа чи кінцівок, то по відповідях користувача система визначає типи переломів і видає повідомлення про те, як правильно надати першу медичну допомогу:

```
(defrule test11 ""
            (patient is in bad condition)
            (traumas of a skull)
            (not (help ?))
              =>
            (if (yes-or-no-p "У пацієнта є ще травми чи переломи черепа? (yes/no)? ")
            then
            (assert (help "Перша медична допомога: Захист рани від інфекцій, перев'язати рану, зупинити кровотечу, дати знеболювальне. Негайно госпіталізувати хворого."))
            else
            (assert (crises of the patient))))
(defrule test12 ""
            (or (and (patient is in bad condition) (patient is in not faint)) (crises of the patient))
            (not (help ?))
            =>
           (bind ?response (ask-question "Вкажіть вид перелому? (1 переломів у пацієнта немає / 2 переломи кінцівок / 3 переломи хребта) ?" 1 2 3))
           (if (eq ?response 2)
           then
           (assert (help "Перша допомога: накласти шину, дати хворому знеболювальне. Госпіталізувати хворого."))
            else
           (if (eq ?response 3)
            then
           (assert (help "Негайна госпіталізація хворого")))))
```

Правила test13 і test14 завершують процес визначення загального стану хворого, і на цьому робота діагностуючих правил завершена. Як показано, ці правила визначають подальший стан хворого: у нього може бути травматичний шок чи синдром здавлювання. У цих випадках система теж видасть повідомлення про те, як правильно надати першу медичну допомогу:

```
(defrule test13 ""
            (patient is in bad condition)
            (not (help ?))
             =>
            (if (yes-or-no-p "Пацієнт реагує на подразники, пульс та тиск пацієнта в нормі? (yes/no)? ")
             then
            (assert (help "Це, напевно, травматичний шок. Перша допомога: виявнити причину винекнення шоку, дати пацієнтові відпочити, можливе спілкування з психологом."))))

(defrule test14 ""
             (patient is in bad condition)
             (not (help ?))
```

```
             =>
             (if (yes-or-no-p "Чи є у хвоого інші скарги: в нього щось бо-
лить, він відчуває слабкість, йому все байдуже, рухи повільні, а мова не
зрозуміла? (yes/no)? ")
              then
              (assert (help "Пацієнту потрібен стаціонарний, додатковий курс
лікування.")))))
```

Правила test15, test16 реалізують правило “begin”. Якщо людина нормально себе почуває, то надавати їй першу медичну допомогу не потрібно:

```
(defrule test15 ""
            (declare (salience 10))
            (patient is in a normal condition)
             =>
            (assert (pomosh "Пацієнт повністю здоровий і дієздатний."))
            (assert (patient is in faint)))
(defrule test16 ""
            (declare (salience 10))
            (patient is in consciousness)
             =>
            (assert (patient is in faint))
            (assert (help "Потрібен додатковий медичний огляд пацієн-
та!t!")))
```

Правило print-help виводить на екран діагностичне повідомлення по наданню першої медичної допомоги:

```
(defrule print-help ""
(declare (salience 10))
(help ?item)
=>
(printout t crlf crlf)
(printout t "Перша медична допомога:")
(printout t crlf crlf)
(format t " %s%n%n%n" ?item))
```

Можливі ситуації, коли стан хворого не можна описати приведеними вище фактами і хворому може знадобитися більш детальний аналіз стану, то CLIPS запустить правило no-helps, що просто порекомендує користувачу звернутися до досвідченого експерта, наприклад, до лікаря:

```
(defrule no-helps ""
(declare (salience -10))
(not (help ?))
=>
(assert (help "Пацієнту потрібен додатковий медичний огляд і нове медичне
заключення лікаря!")))
```

За допомогою факту init ми вибираємо первісні умови системи, тобто система, перед тим як запустити програму, пропонує нам вибрати відповідні дії – чи бажаємо ми почати роботу, чи просто вийти із системи:

```
(deffacts init
(troubleshoot-mode medic)
(menu-level medic main))
```

Правило main-menu дозволяє користувачу вибирати одну з дій головного меню:

```
(defrule main-menu
(declare (salience 10))
(troubleshoot-mode medic) ?ml <- (menu-level medic main) =>
(retract ?ml)
```

```
(printout t crlf crlf )
(printout t "                    Виберіть необхідний пункт меню" crlf)
(printout t "                                                  " crlf)
(printout t "                 1) Запустити програму на виконання" crlf)
(printout t "                 2) Вихід із програми" crlf)
(printout t "                                                   " crlf)
(printout t "Вибір меню: " crlf) (bind ?response (read))
(assert (problem-response medic ?response))
(printout t crlf crlf ))
```

Правило user-exit призначене для виходу з програми, при цьому користувач отримує відповідне повідомлення:

```
(defrule user-exit
(troubleshoot-mode medic)
(problem-response medic 2)
=>
(printout t "До побачення!" crlf)
(halt))
```

Правило begin-start виконується тоді, коли користувач вибрав з головного меню перший пункт. Система приступає до виконання правила test, а користувач може відповідати на питання:

```
(defrule begin-start
(troubleshoot-mode medic) ?pr <- (problem-response medic 1)
=>
(retract ?pr)
(assert (defrule test)))
```

Правило system-banner виводить на екран назву експертної системи, і інформацію про те, що вона робить, при кожному новому запуску:

```
(defrule system-banner ""
(declare (salience 10))
=>
(printout t crlf crlf)
(printout t "             ************************************" crlf)
(printout t "             *   The Medic Diagnosis Expert System *" crlf)
(printout t "             ************************************" crlf)
(printout t "                                                 " crlf)
(printout t "          The Medic Diagnosis Expert System  - це експертна
система, що надає користувачеві  " crlf)
(printout t "          необхідну медичну допомогу та поради в певних ситу-
аціях, коли поряд немає лікаря." crlf)
(printout t "          Система задає питання на які користувач відповідає
Так або Ні і по відповіді видає певні рекомендації " crlf)
(printout t crlf crlf))
```

Щоб продемонструвати роботу програми, необхідно зберегти її в текстовому файлі з розширенням *.clp*. За допомогою команди *load* загрузіть програму в середовище на виконання. Далі треба виконати такі команди, як (reset) і (run). Після цього система повідомляє про наявність помилок у програмі, якщо вони відсутні, то програма готова для демонстрації. Так відбувається спілкування між системою і користувачем, при цьому вона видає певні рекомендації. Наприклад, на рис. 2.11 продемонстровано запуск програми.

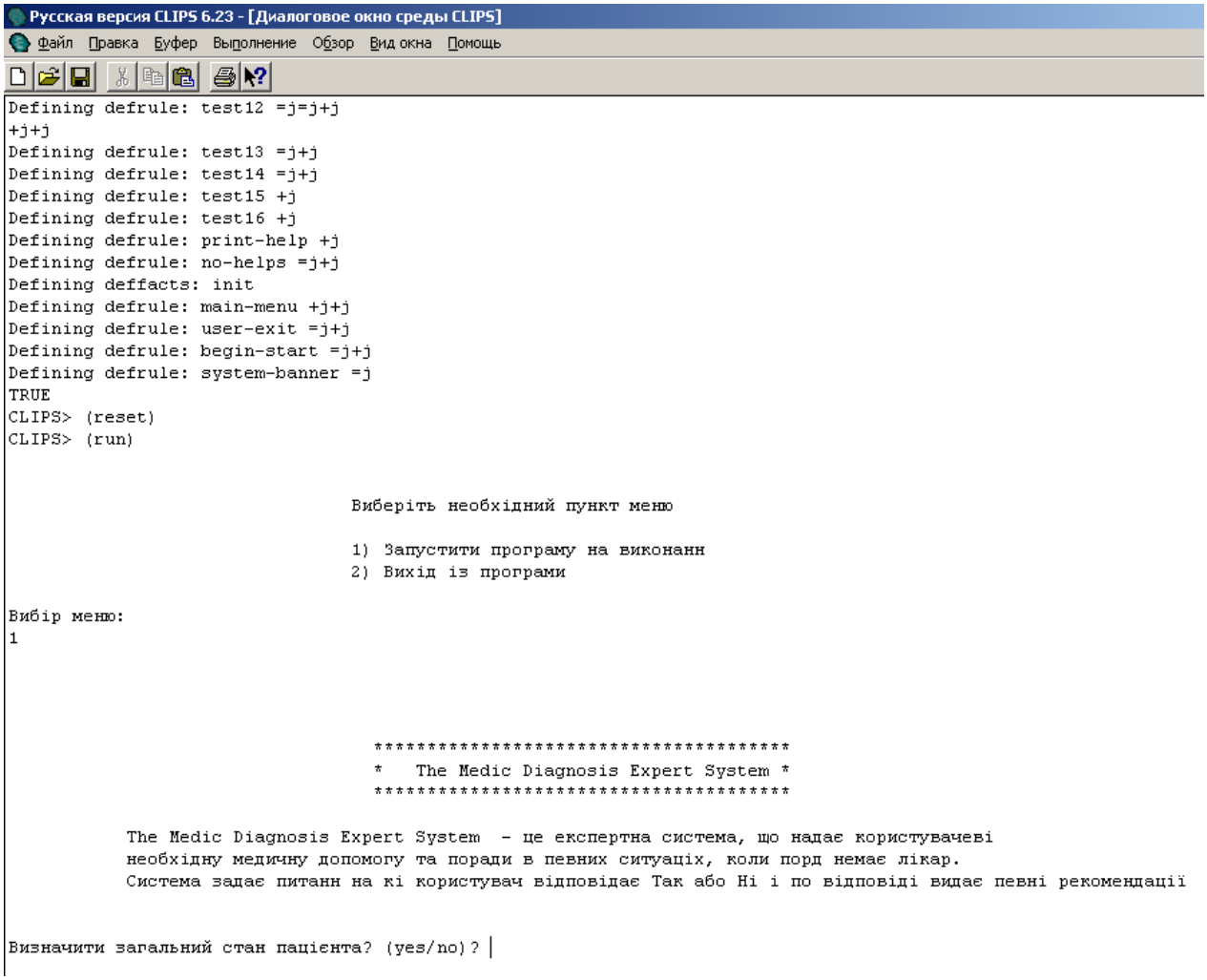


Рис. 2.11. Демонстрація експертної системи

4.2. Експертна система “АвтоЕксперт”

Наведена нижче експертна система здатна проводити діагностику несправності автомобіля і надавати користувачу рекомендації з усунення несправності.

Функція *ask-question* задає користувачу питання і чекає на відповідь:

```
(deffunction ask-question (?question $?allowed-values)
                    (printout t ?question)
                    (bind ?answer (read))
                    (if (lexemep ?answer)
                     then
                     (bind ?answer (lowcase ?answer)))
                     (while (not (member ?answer ?allowed-values)) do
                     (printout t ?question)
                     (bind ?answer (read))
                     (if (lexemep ?answer)
                      then
                      (bind ?answer (lowcase ?answer)))) ?answer)
```

Функція *yes-or-no-p* задає користувачу питання, а користувач відповідає на них *yes (y)* чи *no (n)*. У випадку позитивної відповіді функція повертає значення TRUE, інакше – FALSE:

```
(deffunction yes-or-no-p (?question)
                    (bind ?response (ask-question ?question yes no y n))
                    (if (or (eq ?response yes) (eq ?response y))
```

```
                    Then TRUE else FALSE))
```
Правило determine-engine-state визначає поточний стан двигуна машини по відповідях, одержуваним від користувача. Двигун може знаходитися в одному з трьох станів: працювати нормально (working-state engine normal); працювати незадовільно (working-state engine unsatisfactory); і не заводитися (working-state engine does-not start):
```
(defrule determine-engine-state ""
                (not (working-state engine ?))
                (not (repair ?))
                =>
    (if (yes-or-no-p "Запустить программу на выполнение (yes/no)? ")
    then
     (if (yes-or-no-p "Двигатель запускается нормально (yes/no)? ")
     then
        (assert (working-state engine normal))
     else
        (assert (working-state engine unsatisfactory)))
    else
     (assert (working-state engine does-not start))))
```
Правило *determine-rotation-state* визначає стан обертання двигуна по відповіді, одержуваній від користувача. Двигун може обертатися *(rolation-state engine rotates)* чи не обертатися *(rotation-state engine does-not-rotate)*. Крім того, правило робить припущення про наявність поганої іскри чи її відсутність у системі запалювання:
```
(defrule determine-rotation-state ""
               (working-state engine does-not-start)
               (not (rolation-state engine ?))
               (not (repair ?))
               =>
              (if (yes-or-no-p "Двигатель вращается нормально (yes/no)? ")
                   then
      ; двигун обертається
                   (assert (rolation-state engine rotates))
                   ;погана іскра
                   (assert (spark-state engine irregular-spark))
                   Else
                  ; двигун не обертається
                  (assert (rotation-state engine does-not-rotate))
     ; немає іскри
                  (assert (spark-state engine does-not-spark))))
```
Правило *determine-gas-level* по відповіді користувача визначає наявність палива в баці. У випадку якщо палива в баці немає, користувачу видається рекомендація з ремонту – *машину необходимо заправить*. З появою відповідної рекомендації виконання діагностичних правил припиняється:
```
(defrule determine-gas-level ""
               (working-state engine does-not-start)
               (rotation-state engine rotates)
               (not (repair ?))
              =>
       (if (not (yes-or-no-p "Есть ли в баке машины топливо? (yes/no)? "))
              then
        ; машину необхідно заправити
                  (assert (repair "Машину необходимо заправить."))))
```
Правило *determine-battery-state* по відповіді користувача визначає, чи

заряджений акумулятор. У випадку якщо це не так, користувачу видається рекомендація з ремонту – *Зарядите аккумулятор (repair "Зарядите аккумулятор.")*. Крім того, правило додає факт, що описує стан акумулятора. Виконання діагностичних правил припиняється:

```
(defrule determine-battery-state ""
                (rotation-state engine does-not-rotate)
                (not (charge-state battery ?))
                (not (repair ?))
                =>
          (if (yes-or-no-p "Проверьте, заряжен ли аккумулятор (yes/no)? ")
                 then
    ; акумулятор заряджений
                (assert (charge-state battery charged))
                else
    ; зарядіть акумулятор
               (assert (repair "Зарядите аккумулятор."))
              ; акумулятор не заряджений
    (assert (charge-state battery dead))))
```

Правило *determine-low-output* визначає, чи розвиває двигун нормальну вихідну потужність чи ні, і додає в систему факт, що описує цю характеристику:

```
(defrule determine-low-output ""
                (working-state engine unsatisfactory)
                ; потужність роботи двигуна ще не визначена
      (not (symptom engine low-output | not-low-output))
                (not (repair ?))
                =>
                 (if (yes-or-no-p "Ваш двигатель развивает нормально выход-
ную мощность (yes/no)? ")
                then
    ; низька вихідна потужність двигуна
               (assert (symptom engine low-output))
                else
   ; нормальна вихідна потужність двигуна
               (assert (symptom engine not-low-output))))
```

Правило *determine-point-surface-state* визначає по відповіді користувача стан контактів. Контакти можуть знаходитися в одному з трьох станів: чисті, обпалені і забруднені. У двох останніх випадках користувачу видаються відповідні рекомендації. Виконання діагностичних правил припиняється:

```
(defrule determine-point-surface-state ""
                (or (and (working-state engine does-not-start) ; не заво-
диться
               (spark-state engine irregular-spark)) ; і погана іскра
                (symptom engine low-output)) ; чи низька потужність
                (not (repair ?))
                =>
                 (bind ?response (ask-question "Проверте состояние контак-
тов и укажите какие они (нормальные / опаленные / загрязненные)?" нормаль-
ные опаленные ))
                 (if (eq ?response опаленны)
                    then
        ; контакти обпалені - замініть контакти
                  (assert (repair "Контакты опалены - замените их."))
                 else
                 (if (eq ?response загрязненные)
```

```
                 then
                ; контакти забруднені - прочистіть їх
                     (assert (repair "Контакты загрязненные - прочистите
их.")))))
```

Правило *determine-conductivity-test* по відповіді користувача визначає, чи пропускає струм котушка запалювання. Якщо ні, то її варто замінити. Якщо пропускає, то причина не справності – розподільні проводи. Для нормальної роботи правила необхідно переконатися, що акумулятор заряджений і іскри немає:

```
(defrule determine-conductivity-test ""
               (working-state engine does-not-start)
               (spark-state engine does-not-spark) ; немає іскри
               (charge-state battery charged) ; акумулятор заряджений
               (not (repair ?))
               =>
                (if (yes-or-no-p "Пропускает ли ток катушка зажигания
(yes/no)? ")
                 then
          ; замініть провід
                   (assert (repair "Замените распределительные провода."))
                 else
          ; замініть котушку запалювання
                (assert (repair "Замените катушку зажигания."))))
```

Правило *determine-sluggishness* запитує користувача, чи не веде себе машина інертно (не відразу реагує на подачу палива). Якщо такий факт виявлений, то необхідно прочистити паливну систему, виконання діагностичних правил припиняється:

```
(defrule determine-sluggishness ""
               (working-state engine unsatisfactory)
               (not (repair ?))
               =>
                (if (yes-or-no-p "Машина не ведет себя инертно
                 (не сразу реагирует на подачу топлива (yes/no)? ")
                   then
      (assert (repair "Нужно прочистить систему подачи топлива."))))
```

Правило *determine-misfiring* визначає, чи немає перебоїв із запалюванням. Якщо це так, то необхідно відрегулювати зазори між контактами. Виконання діагностичних правил припиняється:

```
(defrule determine-misfiring ""
               (working-state engine unsatisfactory) (not (repair ?))
               =>
               (if (yes-or-no-p "Нет ли перебоев с зажиганием (yes/no)? ")
                   then
        (assert (repair "Отрегулируйте зазоры между контактами."))
        (assert (spark-state engine irregular-spark)))) ; погана іскра
```

Правило *determine-knocking* визначає, чи не стукає двигун. Якщо це так, то необхідно відрегулювати запалювання. Виконання діагностичних правил припиняється:

```
(defrule determine-knocking ""
               (working-state engine unsatisfactory)
               (not (repair ?))
               =>
          (if (yes-or-no-p "Случайно, Ваш двигатель не стучит (yes/no)? ")
                    then
```

```
        ; відрегулюйте положення запалювання
                    (assert (repair "Отрегулируйте положение зажигания."))))
```

Наступні правила визначають стан деяких підсистем автомобіля по характерних станах двигуна:

```
(defrule normal-engine-state-conclusions ""
                (declare (salience 10))
      ; якщо двигун працює нормально
      (working-state engine normal)
            =>
            (assert (repair "Ремонт машины не нужен"))
            (assert (spark-state engine normal)) ; запалювання в нормі
            (assert (charge-state battery charged)) ; акумулятор заряджений
            (assert (rolation-state engine rotates)) ) ; двигун обертається
(defrule unsatisfactory-engine-state-conclusions ""
                (declare (salience 10))
     ; якщо двигун працює незадовільно
                (working-state engine unsatisfactory)
                =>
           (assert (charge-state battery charged)) ; акумулятор заряджений
           (assert (rotation-state engine rotates)) ) ;двигун обертається
```

Правило *no-repairs* запускається у випадку, якщо жодне з діагностичних правил не здатне визначити несправність. Правило перериває виконання експертної системи і пропонує пройти більш ретельну перевірку:

```
(defrule no-repairs ""
                (declare (salience -10))
                (not (repair ?))
                =>
                 (if (yes-or-no-p "Выдать другие полезные рекомендации
(yes/no)? ")
                     then
  (assert (repair "1. Перед каждым выездом проверяйте и доводите до нормы
давление воздуха в шинах.
  2. Своевременно, в соответствии с дорожными условиями на низшую передачу,
избегая перегрузки двигателя.
  3. Не меняйте масло, залитое в двигатель на заводе.
  4. Не включайте стартер при работающем двигателе. Это может привести к
поломке зубьев приводной шестерни или маховика.
  5. Регулярно проверяйте  состояние защитных резиновых чехлов
     шаровых опор и защитных колпачков рулетовых тяг.
  6. Никогда не допускайте работы двигателя с частотой вращения коленчатого
вала, при котором стрелка тахометра находится в красной зоне шкалы.
  7. Постоянно следите за чистотой клемм и зажимов аккумуляторной батареи и
за их соединение.
  8. При установке аккумуляторной батареи на автомобиль следите за тем,
чтобы провода были соединены.
  9. При появлении во время движения вибраций проверьте балансировку колес
на предприятиях технического обслуживания.
 10. В промежутках между обслуживаниями для надежной работы двигателя
     проводите слив отстоя из топливного фильтра с последующей прокачкой
топливной системы." ))))
```

Правило *print-repair* виводить на екран діагностичне повідомлення по усуненню знайденої несправності:

```
(defrule print-repair ""
                (declare (salience 10))
                (repair ?item)
                =>
```

```
                (printout t crlf crlf)
                (printout t "Устранение неисправностей:")
                (printout t crlf crlf)
                (format t " %s%n%n%n" ?item))
```

Факт *init* дозволяє створити меню вибору:

```
(deffacts init
             (troubleshoot-mode engine)
             (menu-level engine main))
```

Правило *main-menu* дає користувачу вибрати деякі дії – запустити програму чи вийти з неї:

```
(defrule main-menu ""
            (declare (salience 500))
            (troubleshoot-mode engine)
            ?ml <- (menu-level engine main)
            =>
           (retract ?ml)
           (printout t crlf crlf )
            (printout t "          Выберете подходящий пункт меню" crlf)
            (printout t "                                        " crlf)
            (printout t "                  1) Запустить программу" crlf)
            (printout t "                   2) Выход из программы" crlf)
            (printout t "                                        " crlf)
            (printout t "Пункт меню: " crlf)
            (bind ?response (read))
            (assert (problem-response engine ?response))
            (printout t crlf crlf ))
```

Правило *user-exit* надає можливість вийти з програми:

```
(defrule user-exit ""
            (troubleshoot-mode engine) (problem-response engine 2)
            =>
               (printout t "До встречи!" crlf)
               (halt))
```

Правило *begin-start* дозволяє користувачу почати роботу і зробити діагностику роботи машини:

```
(defrule begin-start ""
            (troubleshoot-mode engine)
            ?pr <- (problem-response engine 1)
            =>
                (retract ?pr)
                (assert (defrule determine-engine-state)))
```

Правило *system-banner* виводить на екран назву експертної системи при кожному новому запуску:

```
(defrule system-banner ""
                (declare (salience 10))
                =>
 (printout t crlf crlf)
 (printout t "                ************************************" crlf)
 (printout t "                *   Экспертная система АвтоМеханик  *" crlf)
 (printout t "                ************************************" crlf)
 (printout t "                                                    " crlf)
 (printout t "    Экспертная система "АвтоМеханик" способна диагностировать
некоторые неисправности" crlf)
 (printout t "  автомобиля и предоставлять пользователю рекомендации по
устранению неисправности." crlf)
 (printout t "  На вопросы системы отвечайте YES или NO." crlf)
```

Для запуску програми наберіть свою програму у текстовому редакторі і збережіть файл з розширенням *.clp*, наприклад *авто.clp*. Після цього запустіть CLIPS, якщо він уже був запущений, то визвіть команду *(clear)*. Завантажте створений Вами файл за допомогою команди *load*, наприклад *(load "авто.clp")*. Якщо файл був набраний без помилок, то Ви повинні побачити повідомлення про успішну спробу завантаження файлу. Функція *load* повернула значення *TRUE*. Якщо це не так, то у синтаксисі визначень чи функцій правил була допущена помилка. На рис. 2.12 зображено запуск експертної системи “АвтоЕксперт”.

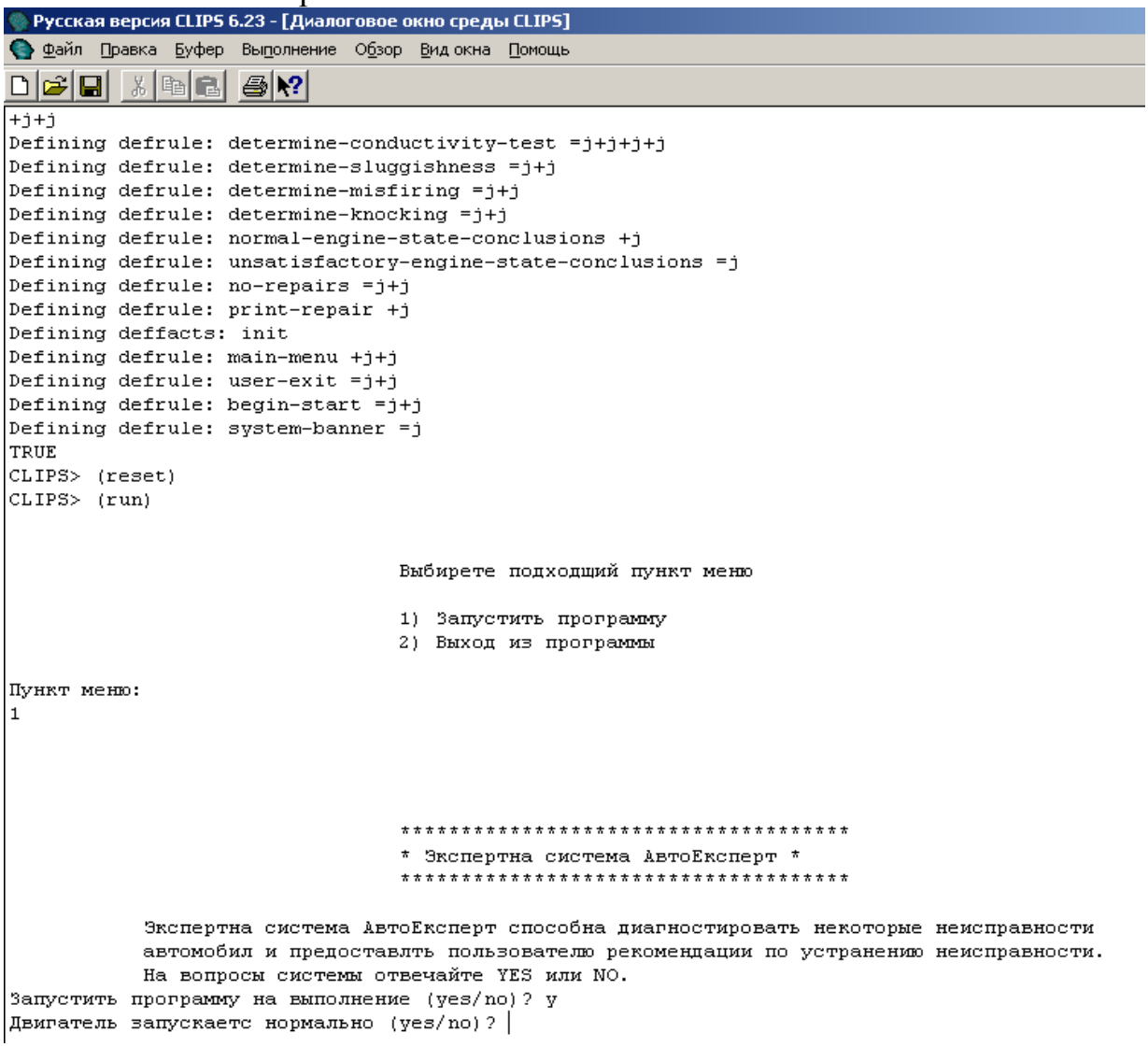


Рис. 2.12. Запуск експертної системи “АвтоЕксперт”

Висновки

Основними перевагами CLIPS як оболонки експертної системи є наступні:

1. Суть технології CLIPS полягає в тому, що мова і середовище CLIPS дають змогу користувачеві швидко створювати ефективні, компактні і легко керовані експертні системи. При цьому користувач застосовує безліч вже готових інструментів (вбудований механізм управління базою знань, механізм логічного виводу, менеджери різних об'єктів CLIPS і т.п.) та конструкції (факти, шаблони, правила, функції, родові функції, класи, модулі, обмеження, мову COOL).

2. Система, крім виконання обчислювальних операцій, формує певні висновки, ґрунтуючись на тих знаннях, якими вона володіє. Знання в системі представлені на спеціальній мові і зберігаються окремо від власне програмного коду, що і формує висновки і розуміння. Цей компонент програми прийнятий називати базою знань.

3. CLIPS має змогу пояснити, чому запропоновано саме таке рішення, і довести його обґрунтованість.

4. На даний час CLIPS – це вільно поширюваний продукт, який продовжує успішно розвиватися і вдосконалюватися.

5. Локалізація CLIPS передбачає локалізацію інтерфейсу користувача, ядра та синтаксису CLIPS. В результаті локалізації була створена «Русская версия CLIPS», застосування якої показало наступні переваги:

а) зручність у використанні та при написанні експертних систем;

б) якщо виникає помилка, система видає повідомлення російською мовою, що полегшує розуміння змісту помилки;

в) при написанні програми користувач має можливість застосовувати російські та українські позначення фактів, змінних, правил, процедур тощо.

Список використаних джерел

1. Вивчення експертних систем у курсі основи інформатики і обчислювальної техніки: Методичні рекомендації / Укл. Ю.С. Рамський, Н.Р. Балик. – К.: УДПУ, 1995.
2. Іваськів І.С., Рамський Ю.С., Балик Н.Р. До питання про розробку інструментальної експертної системи // Матеріали Всеук. конф. молодих науковців "Інформаційні технології в науці та освіті". – Черкаси, 1997. – С. 9–14.
3. Левин Р., Дранг Д., Эдельсон Б. Практическое введение в технологию искусственного интеллекта и экспертных систем с иллюстрациями на Бейсике. – Москва, 2000.
4. Марселлус Д. Программирование экспертных систем на Турбо-Прологе: Пер. с англ. / Предисл. С.В. Трубицына. – М.: Финансы и статистика, 1994.
5. Осуга С. Обработка знаний. – М.: Мир, 1990. – 304 с.
6. Петрушин В.А. Экспертно-обучающие системы. – К.: Наук. думка, 1992. – 196 с.
7. Представление и использование знаний: Пер. с япон. – М.: Мир, 1989. – 220 с.
8. Пустынникова И.Н. Построение баз знаний экспертных систем как вид учебной деятельности. // Теорія та методика навчання математики, фізики, інформатики: Збір. наук. праць: В 3-х т. – Кривий Ріг: Вид. відділ НМетАУ, 2002. – Т. 3. – С. 197–20.
9. Рамський Ю.С., Балик Н.Р. Методичні основи вивчення експертних систем у школі. – К.: Логос, 1997. – 128 с.
10. Рихтер Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows. Пер. с англ. – 4 изд. – СПб.: Питер; М.: Русская редакция, 2001. – 752 с., ил.
11. Фролов А.В., Фролов Г.В. Библиотека системного программиста. Т. 11-17. Операционная система Microsoft Windows 3.1 для программиста. – М.: ДИАЛОГ-МИФИ, 1994.
12. Хейес-Рот Ф. и др. Построение экспертных систем // Под ред. Ф. Хейес-Рота, Д. Уотермена, Д. Лената. – М.: Мир, 2001.
13. Частиков А.П., Гаврилова Т.А., Белов Д.Л. Разработка экспертных систем. Среда CLIPS. – СПб.: БХВ-Петербург, 2003. – 608 с.: ил.
14. Частиков А.П., Белов Д.Л. Регенеративные экспертные системы. // Искусственный интеллект. – 2002. – №3.
15. Яшин А.М. Разработка экспертных систем. – Л.: Изд-во ЛПИ, 1990.