Раздел 6. Практика применения языка CLIPS для построения экспертных систем

6.1.Введение

Если у вас есть проблема или задача, которую нельзя решить самостоятельно – вы обращаетесь к знающим людям, или к экспертам, т.е. к тем, кто обладает знаниями. Напомним, термин "системы, основанные на знаниях" (knowledge-based systems) появился в 1976 году одновременно с первыми системами, аккумулирующими опыт и знания экспертов. Это были экспертные системы MYCIN и DENDRAL для медицины и химии. Они ставили диагноз при инфекционных заболеваниях крови и расшифровывали данные масс-спектрографического анализа.

Экспертные системы появились в рамках исследований по искусственному интеллекту. Этот прорыв произошел, когда на смену поискам универсального алгоритма мышления и решения задач исследователям пришла идея моделировать конкретные знания специалистов-экспертов. Так в США появились первые коммерческие ***системы, основанные на знаниях, или экспертные системы (ЭС).*** Эти системы по праву стали первыми интеллектуальными системами, и до сих пор единственным критерием интеллектуальности является наличие механизмов работы со знаниями.

Так появился новый подход к решению задач искусственного интеллекта – ***представление знаний***.

При изучении интеллектуальных систем традиционно возникает вопрос – что же такое знания и чем они отличаются от обычных данных, десятилетиями обрабатываемых на компьютерах.

Данные – это информация, полученная в результате наблюдений или измерений отдельных свойств (атрибутов), характеризующих объекты, процессы и явления предметной области.

Знания – это связи и закономерности предметной области (принципы, модели, законы), полученные в результате практической деятельности и профессионального опыта, позволяющего специалистам ставить и решать задачи в данной области. Можно сказать, что знания – это хорошо структурированные данные, или данные о данных, или метаданные.

В настоящее время разработаны десятки моделей (или языков) представления знаний для различных предметных областей. Большинство из них может быть сведено к следующим классам:

- продукционные модели;
- семантические сети;
- фреймы;
- формальные логические модели.

Наиболее распространенной является ***продукционная модель.*** Продукционная модель или модель, основанная на правилах, позволяет представить знания в виде предложений типа "Если (условие), то (действие)". Под "условием" (антецедентом) понимается некоторое предложение-образец, по которому осуществляется поиск в базе знаний, а под "действием" (консеквентом) – действия, выполняемые при успешном исходе поиска. Они могут быть промежуточными, выступающими далее как условия, и терминальными или целевыми, завершающими работу системы. Внутри консеквента могут также генерироваться и добавляться в базу данных новые факты, которые были получены, например, результате вычислений или взаимодействия с пользователем.

Чаще всего вывод на такой базе знаний бывает ***прямой*** (от данных к поиску цели) или ***обратный*** (от цели для ее подтверждения – к данным). Данные – это исходные факты, хранящиеся в базе фактов, на основании которых запускается машина вывода или

интерпретатор правил, перебирающий правила из продукционной базы знаний.

Продукционная модель часто применяется в промышленных экспертных системах, поскольку привлекает разработчиков своей наглядностью, высокой модульностью, легкостью внесения дополнений и изменений и простотой механизма логического вывода.

***

В данной главе рассматривается практическое применение языка CLIPS, для построения экспертных систем основанных на продукционной модели представления знаний.

Прототип языка CLIPS был разработан в 1985 г. в космическом центре NASA. Изначально аббревиатура расшифровывалась как «C Language Integrated Production System» (язык С, интегрированный с продукционными системами). С момента разработки язык был неоднократно модернизирован, введены процедурные и объектно-ориентированные парадигмы, введена поддержка модульной структуры программ и многое другое. В настоящее время CLIPS является мощным инструментом для создания экспертных систем и распространяется бесплатно. Страница проекта в интернете находится по адресу **http://clipsrules.sourceforge.net/**, там же можно свободно загрузить текущую версию среды разработки для своей операционной системы.

В настоящей главе мы будем рассматривать работу с версией CLIPS для операционной системы Windows®. Примеры программного кода, а также скриншоты программной оболочки относятся к **CLIPS версии 6.241**.

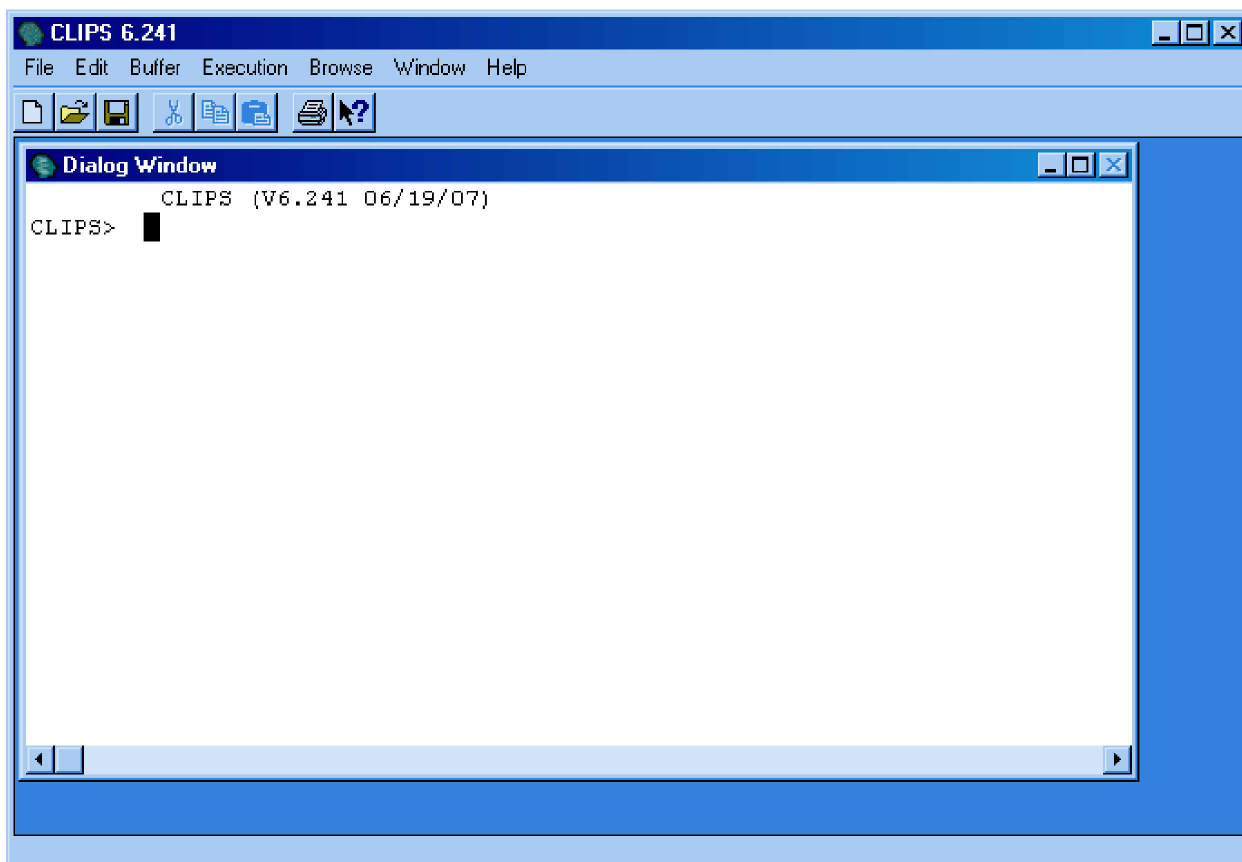

Рис. 6.1. Внешний вид оболочки языка CLIPS после запуска

Дочернее окно оболочки «Dialog Window», представленное на рис 6.1. используется для выполнения команд CLIPS и получения результата. Это своего рода командная строка оболочки, где можно выполнить команду CLIPS записав ее после приглашения и нажав клавишу <Enter>. Например, написав в строке после приглашения следующий код

(включая скобки):

```
(printout  t  “Hello world” crlf)
```

и нажав <Enter>, мы заставим CLIPS выполнить функцию вывода на экран и получим строку «Hello world», сразу после нашего вызова, после чего оболочка вернется в режим ожидания.

Для того чтобы начать писать программу на языке CLIPS нужно создать новый файл путем выбора пункта меню File – New.

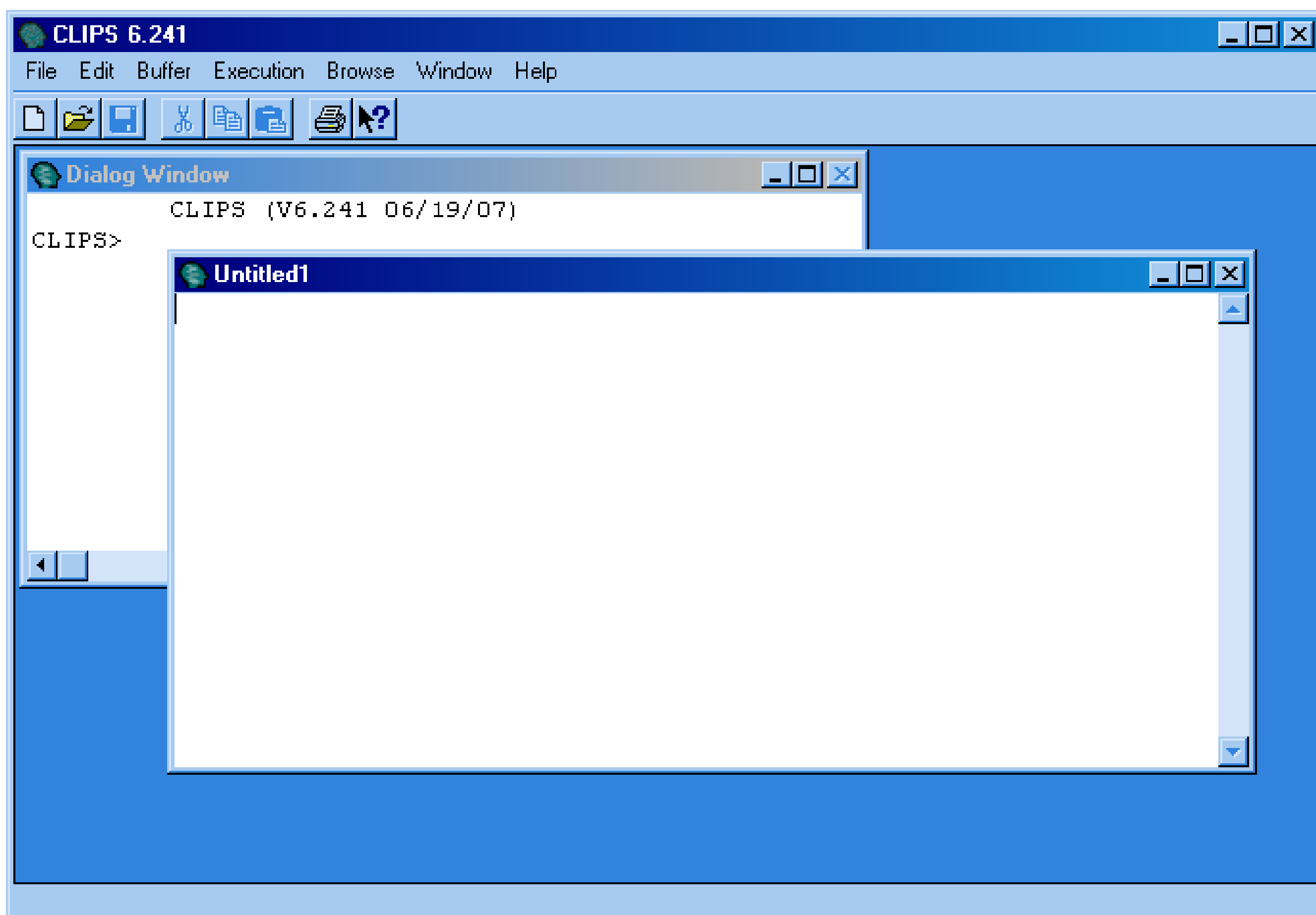

Рис 6.2. Окно для составления CLIPS-программ

Для того чтобы команды, написанные в окне, были исполнены необходимо выделить их и нажать комбинацию Ctrl-M или выбрать пункт меню Buffer – Batch Selection.

Сразу отметим, что программа на CLIPS принципиально отличается от программы на классическом языке программирования, таком как, например, C или Pascal. Запуск программы на CLIPS фактически означает запуск механизма логического вывода, основанного на имеющих в системе правилах, которые в свою очередь основаны на фактах. В общем случае простейшая программа на CLIPS состоит из набора правил и функций, манипулирующих фактами, о которых речь пойдет немного позже.

6.2.Примитивные типы данных языка CLIPS

Примитивными типами данных языка CLIPS являются: float, integer, symbol, string, external-address, fact-address, instance-name, instance-address. Для хранения числовой информации предназначаются типы float и integer, для символьной – symbol и string.

Число в CLIPS может состоять только из символов цифр (0–9), десятичной точки

(.), знака (+ или -) и экспоненциального символа (е) с соответствующим знаком, в случае представления числа в экспоненциальной форме. Ниже приведены примеры допустимых в CLIPS представлений целых и вещественных типов:

Целые: 237; 15; +12; -32

Вещественные: 237е3; 15.09; +12; -32.3е-7

Значением типа symbol может быть любая последовательность символов, начинающаяся с любого не управляющего ASCII-символа. Значение типа symbol заканчивается *ограничителем.* Ограничителями являются любые неотображаемые символы (например, пробел, символ табуляции или перехода на другую строку), двойные кавычки, открывающая или закрывающая круглая скобка, символы &, |, < и ~. Точка с запятой (;) является символом начала комментариев и также может ограничивать значение типа symbol. Ниже приведены образцы допустимых значений типа symbol:

good Hello B76-HI bad_value

127A 456-93-039 @+=-% 2each

В языке CLIPS существуют зарезервированные слова, которые не могут быть использованы в качестве значений типа symbol. Список зарезервированных слов достаточно велик, его можно найти в документации по языку.

Значение типа string представляет собой строку символов, заключенную в двойные кавычки. Примеры допустимых значений string приведены ниже

"Life is good" "A and B" "1 number" "Value"

Использование остальных типов данных выходит за рамки настоящегораздела, для их изучения обратитесь к документации и соответствующей литературе.

6.3.Особенности вызова функций в языке CLIPS

Функцией в CLIPS называется часть кода, имеющая имя и возвращающая полезный результат или выполняющая полезные действия (например, отображение информации на экране) Функции, не возвращающие результат и выполняющие некоторую полезную работу, обычно называются *командами.*

CLIPS оперирует несколькими типами функций – *внешние функции, системные функции, пользовательские функции, родовые функции.* Внешние функции могут создаваться на других языках программирования (например, С), и затем подключаются к CLIPS на этапе компилирования или функционирования среды, однако их использование выходит за рамки данного пособия. Системные функции созданы разработчиками среды CLIPS и описаны в руководстве. Пользовательские функции определяются в среде CLIPS при помощи специального конструктора `deffunction`, о котором речь пойдет позже.

Обратите внимание на следующие особенности вызова функций в CLIPS:

- имя функции вместе со всеми ее аргументами заключается в круглые скобки
- аргументы функции всегда следуют после имени функции.
- аргументы отделяются друг от друга, по крайней мере, одним пробелом.
- аргументами функций могут быть переменные примитивных типов, константы или вызовы других функций.

Ниже приведены примеры использования функций + (арифметическое сложение) и * (арифметическое умножение):

```
(+  3  4  5)
(*  5  6.0  2)
(+  3  (*  8  9)  4)
```

Вы можете увидеть результат выполнения функций, введя их в диалоговое окно оболочки CLIPS (см. рис. 1).

6.4. Факты в языке CLIPS

Факт – одна из основных форм представления данных в CLIPS. Каждый факт – это определенный набор данных, сохраняемый в текущем списке фактов – рабочей памяти системы. Список фактов представляет собой это универсальное хранилище фактов и является частью базы знаний. Объем списка фактов ограничен размером памяти памятью компьютера.

В системе CLIPS фактом является список неделимых (или атомарных) значений примитивных типов данных, заключенный в скобки. Например:

```
(life is good)
(power failed 22 minutes)
(temperature 10)
(model "OPEL Omega")
```

являются фактами языка CLIPS. Обычно факты используются для того, чтобы внести в систему заранее известные знания или добавить информацию, полученную в процессе диалога с пользователем или в результате вычислений.

Факты бывают простыми и составными. Составные факты являются аналогами структур в классических языках программирования и определяются с помощью специальных конструкторов. Составные факты будут рассмотрены позже, а сейчас рассмотрим функции для работы с простыми фактами:

6.4.1. Функция assert – добавление факта в список

Функция **assert** принимает в качестве параметров последовательность фактов, которые подлежат добавлению в список фактов. Синтаксическую структуру вызова функции **assert** можно представить следующим образом:

*(**assert** <факт> <факт> <факт>)*

Для того чтобы иметь возможность наблюдать процесс добавления удаления или изменения фактов необходимо вызвать пункт меню Execution – Watch и установить флажок напротив пункта «Facts» (см. рис. 6.3).

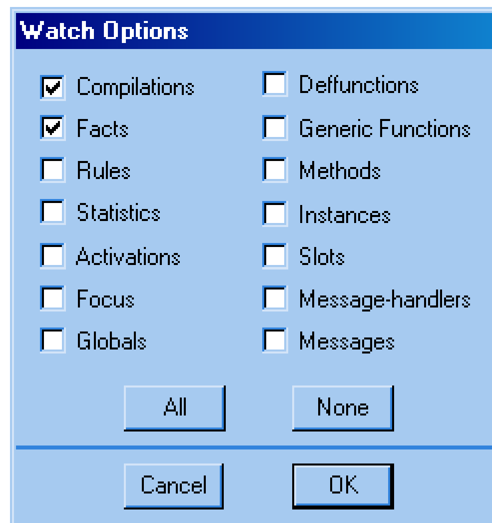

Рис 6.3. Установка параметров трассировки

Рассмотрим пример использования функции **assert**. Перейдите **в диалоговое окно** CLIPS, (см. рис. 1) и введите следующее выражение:

```
(assert (weather is fine))
```

Будьте внимательны со скобками. В том случае, если вы правильно ввели команду, в список фактов добавиться факт `(weather is fine)` и появится результат выполнения:

```
==> f-0     (weather is fine)
<Fact-0>
```

означающий, что факт был добавлен в систему и получил номер 0. При успешном выполнении функция **assert** возвращает адрес последнего добавленного факта. Если во время добавления некоторого факта произошла ошибка, команда прекращает свою работу и возвращает значение FALSE.

Проверить текущее состояние списка фактов можно либо посредством ввода в диалоговом окне команды `(facts)`, либо вызвав пункт меню Window – 1 Fact. (см. рис.6.4.) Во втором случае откроется окно со списком фактов.

При добавлении факта можно использовать вызовы функций, например выполнение:

```
(assert (totalcost (* 10 13))
```

вызовет функцию умножения и ее результат запишется в качестве поля факта. При этом в память добавится факт (totalcost 130).

Для очистки памяти среды используйте команду `(clear)`

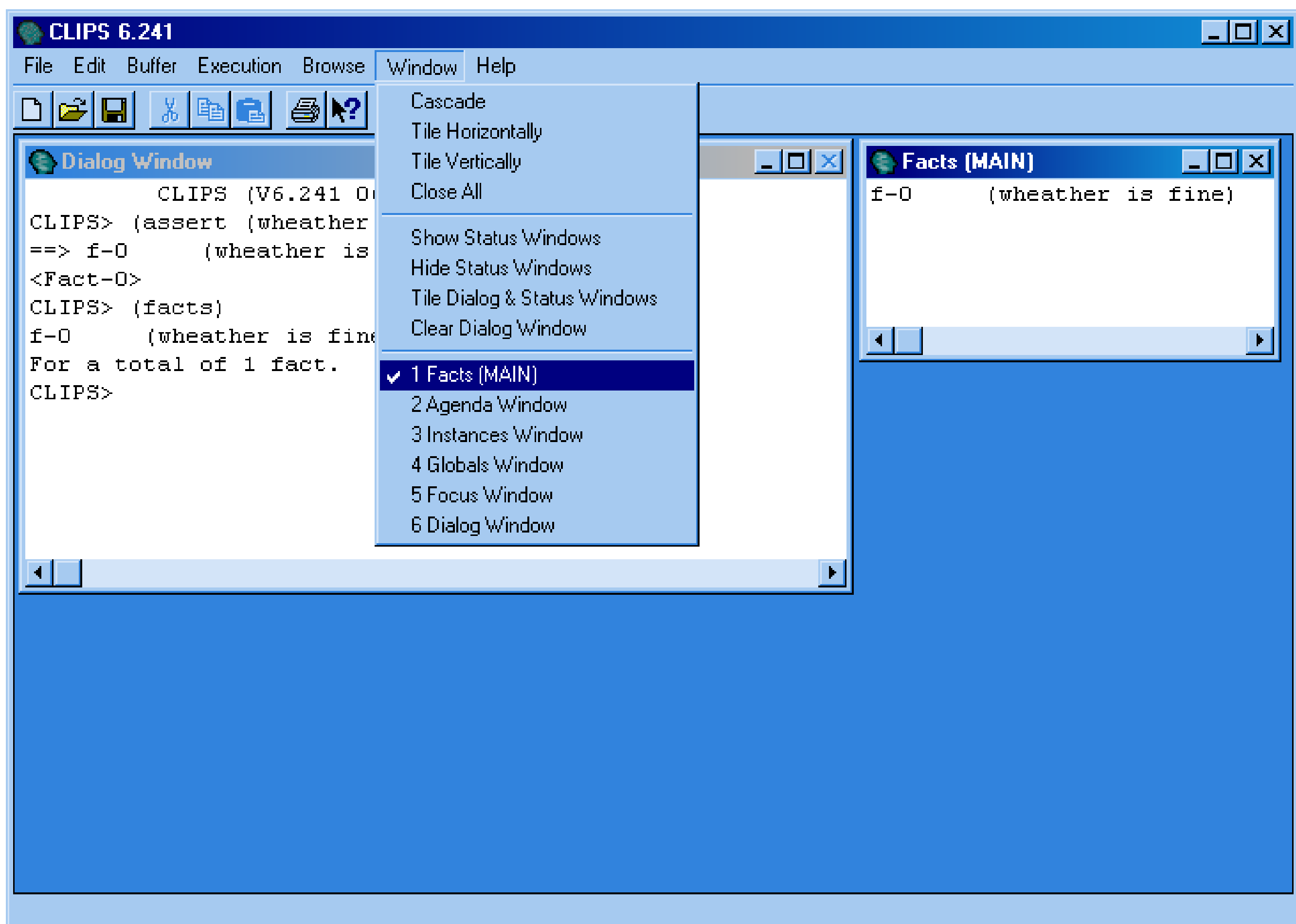

Рис 6.4. Вызов окна со списком фактов

➢ *Задания для самостоятельной работы:*

1. Выполните самостоятельно следующий набор команд:

```
(clear)
(assert (color red))
```

```
(assert (color blue)(value (+ 3 4)))
(assert (suggest "Turn on the left"))
```

2. Выведите список фактов в диалоговое окно, убедитесь, что все факты добавлены верно. Обратите внимание на факт Value.

3. Самостоятельно придумайте и добавьте факт означающий, что лампа в коридоре включена.

6.4.2. Функция retract – удаление фактов

Для удаления фактов из текущего списка фактов в системе CLIPS предусмотрена функция **retract**. Каждым вызовом этой функции можно удалить произвольное число фактов. В случае если был включен режим просмотра изменения списка фактов, то соответствующее информационное сообщение будет отображаться в окне CLIPS при удалении каждого факта.

Синтаксическую структуру вызова функции **retract** можно представить следующим образом:

*(**retract** <определение-факта> <определение-факта> ...)*

или

*(**retract** *)*

Аргумент *<определение-факта>* может являться либо переменной, связанной с адресом факта (адрес факта возвращается командой **assert**), либо индексом факта без префикса (например, 3 для факта с индексом f-3), либо выражением, вычисляющим этот индекс (например, (+ 1 2) для факта с индексом f-3). Если в качестве аргумента функции **retract** использовался символ * (звездочка), то из текущего списка будут удалены все факты. Функция **retract** не имеет возвращаемого значения.

Рассмотрим работу функции **retract** на примере. Перейдите **в диалоговое окно** CLIPS, (см. рис. 1) и введите следующие команды:

```
(clear)
(assert (a) (b) (c) (d) (e) (f))
(retract 0 (+ 0 2) (+ 0 2 2))
```

Результатом выполнения приведенной выше последовательности команд будет:

1. Очистка списка фактов.
2. Добавление в систему шести фактов.
3. Удаление из системы фактов с номерами 0, 2 и 4.

Обратите внимание, как меняется список фактов в окне Facts с каждой выполненной командой. При успешном выполнении указанных команд в списке останутся факты (b), (d) и (f).

6.4.3. Конструктор deffacts

Конструктор `deffacts` позволяет определить набор фактов, которые автоматически будет добавлять в список фактов при выполнении сброса среды CLIPS (команда `reset`). Синтаксическая схема конструктора `deffacts`:

*(**deffacts** <имя_списка_фактов>*
<факт>
...
<факт>
)

Обратите внимание, что в памяти системы может быть несколько списков автоматически добавляемых фактов, и все они будут внесены в список. Например:

```
(deffacts AutoFactListNumber1    ; имя первого списка фактов
    (code  1)
```

```
        (temp  10)
        (state   working)
)
(deffacts InitFactList      ;имя  второго списка фактов
        (usemargin  TRUE)
        (defaultcolor  black)
)
```

Обратите также внимание на то, что имя списка фактов может быть любым допустимым значением типа symbol и служит более в качестве информативного атрибута, нежели как идентификатор.

При выполнении сброса среды (команда reset) в память системы будут добавлены 5 фактов:

1. (code 1)
2. (temp 10)
3. (state   working)
4. (usemargin TRUE)
5. (defaultcolor black)

Кроме явно заданных фактов, CLIPS также автоматически добавляет предопределенный факт **(initial-fact)** каждый раз при выполнении команды `(reset)`. Факт (initial-fact) может использоваться для определения момента запуска механизма логического вывода, а также неявно присутствует в правилах, для которых не задана предпосылка (см. ниже).

6.4.4.Неупорядоченные факты (шаблоны)

Как было оговорено выше, неупорядоченные факты в CLIPS похожи на структуры (struct) в языке C или записи (record) в языке Pascal. Шаблон факта состоит из имени факта и определения полей для хранения данных, которые называются **слотами.** Шаблоны полезно использовать для описания какой-либо сущности, имеющей набор атрибутов. Например, если перед нами стоит цель внести систему информацию об автомобиле, то его слотами могут в простом случае являться цвет и марка. Для определения абстрактной структуры шаблона служит специальный конструктор `deftemplate`. Синтаксическая схема использования конструктора в простом случае может быть представлена в следующем виде:

*(**deftemplate** <Имя_шаблона> [“Необязательный комментарий”]*

<Определение_слота>

…

<Определение_слота>

)

Слот может быть простым или составным. В простом слоте может быть сохранено одно значение примитивного типа CLIPS, в составном слоте может храниться список из нескольких примитивных типов. Ключевыми словами для определения назначения слота являются `slot` для простого слота и `multilslot –` для составного. Итак, *<определение слота>* в простом случае состоит из:

- ключевого слова slot или multislot, определяющего тип слота;
- имени слота, которое является значением типа symbol.
- необязательного ограничения на тип значения, хранимого в слоте.

<Определение слота> =

(`Slot / multislot` <имя_слота>)

Рассмотрим ***пример шаблона*** для фактов, описывающих автомобиль. Перейдите в

диалоговое окно CLIPS (см. рис. 1). Введите команду `(clear)` для очистки среды. Затем введите конструктор:

```
(deftemplate  car  "This is template for describing a car"
      (slot color)
      (slot model)
      (multislot  owner)
)
```

В этом примере мы создали абстрактный шаблон с именем car, в котором имеется 2 простых слота для хранения цвета и модели автомобиля и один составной слот для хранения данных о владельце.

В случае успешного создания шаблона, оболочка вернется в режим ожидания ввода без каких-либо сообщений. В противном случае мы получим сообщение об ошибке. В случае ошибки внимательно проверьте расстановку скобок в теле конструктора.

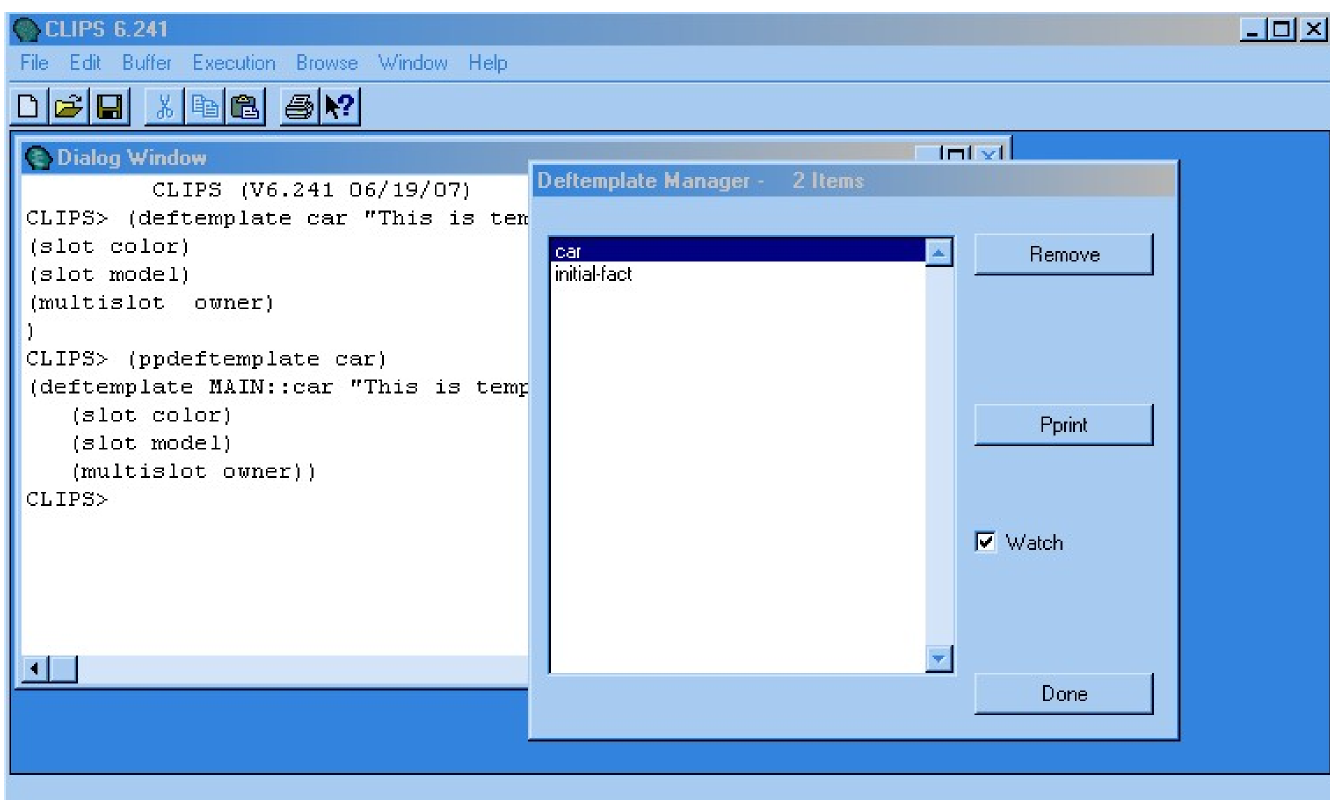

Рис. 6.5. Результат создания шаблона car и окно менеджера шаблонов.

Для просмотра списка шаблонов в системе на текущий момент воспользуйтесь визуальным инструментом "Deftemplate Manager", который находится в меню "Browse". Открыв "Deftemlate Manager", убедитесь, что в списке шаблонов имеется только что определенный шаблон `Car`. Существует возможность вывести содержимое шаблона в диалоговое окно, нажав кнопку Pprint в окне "Deftemlate Manager" (рис. 5).

После определения шаблона, мы можем вносить информацию об автомобилях в виде фактов. Неупорядоченный факт определяется путем указания имени его шаблона с последующим перечислением имен и значений слотов. Например, неупорядоченным фактом является следующая конструкция:

(car (color red) (model Ford) (owner Petrov Sergey).

Перейдите в диалоговое окно и введите следующий вызов функции assert (обратите внимание, что CLIPS – регистрозависимый язык):

```
(assert  (car
              (color red)
              (model Ford)
              (owner Petrov Sergey)
         ) ;скобка закрывает факт
);  скобка, закрывающая вызов функции assert
(assert  (car
              (color white)
              (model Opel)
              (owner Ivanov Dima)
         )
);  скобка, закрывающая вызов функции assert
```

Проверьте добавление фактов в систему. Обратите внимание, что для всех фактов мы используем одно и то же имя шаблона. Для факта не существует имени в его классическом понимании. Идентификатором факта служит его индекс или адрес.

- *Задания для самостоятельной работы*

1. Добавьте в систему еще несколько фактов об автомобилях ваших друзей.

2. Определите и внесите в систему шаблон, описывающий мобильный телефон (модель, цвет, тип корпуса).

3. Добавьте в систему несколько моделей мобильных телефонов.

- ***Совет:*** *Сохраняйте определение разработанных шаблонов и команд в отдельном текстовом файле – они понадобятся вам в дальнейшем при изучении языка.*

6.4.5. Команды modify, reset, clear

Команда modify – изменение шаблонного факта. Для изменения шаблонного факта служит команда modify. В качестве параметра команда принимает определение факта (переменная-адрес или индекс факта), а также новые значения определенных слотов. Например, выполним последовательно команды:

```
(clear)
(deftemplate  weather
     (slot temperature)
     (slot windspeed)
)
(assert
   (weather (temperature 10) (windspeed 0))
)
(modify 0 (temperature 20) )
```

Приведенные выше команды определяют шаблон `weather`, добавляют в систему факт о температуре и скорости ветра, а затем командой `modify` производятся изменения значения слота `temperature`. Проследите за выполнением команды modify, включив трассировку фактов, как показано на рис. 3. По сути, команда удаляет существующий факт и добавляет новый.

- ***Важно!*** *При выполнении команды* `modify` *индекс факта изменяется!*

Существует множество дополнительных функций для работы с фактами. Их описание и примеры использования можно найти в on-line документации, а также в рекомендованной литературе.

Команда reset. Команда `(reset)` очищает список фактов, а затем добавляет в него факты, объявленные конструкторами `deffacts`, включая факт (initial-fact). Обычно используется в сочетании с командой `(run)` для перезапуска написанной программы.

Команда clear. В отличие от команды `(reset)`, команда `(clear)` выполняет глубокую очистку выполняемой среды. Очищаются не только факты, но также все определенные списки, правила переменные, шаблоны. Команда (clear) не добавляет в память системы никаких фактов.

6.5.Правила

Правила в CLIPS служат для представления эвристик или так называемых "эмпирических правил", которые определяют набор действий, выполняемых при возникновении некоторой ситуации. Правила состоят из предпосылок и действия.

Предпосылки правила представляют собой набор условий (или условных элементов), которые должны удовлетвориться, для того чтобы правило выполнилось. Предпосылки правил удовлетворяются в зависимости от наличия или отсутствия некоторых заданных фактов в списке фактов (о котором было рассказано в предыдущей главе) или некоторых созданных объектов, являющихся экземплярами классов, определенных пользователем (о которых будет показано ниже). Один из наиболее распространенных типов условных выражений в CLIPS – образцы (patterns). Образцы состоят из набора ограничений, которые используются для определения того, удовлетворяет ли некоторый факт или объект условному элементу. Другими словами, образец задает некоторую маску для фактов или объектов. Процесс сопоставления образцов фактам или объектам называется *процессом сопоставления образцов* (pattern-matching). CLIPS предоставляет механизм, называемый механизмом логического вывода (inference engine), который автоматически сопоставляет образцы с текущим списком фактов и определенными объектами в поисках правил, которые применимы в данный момент.

6.5.1.Использование конструктора defrule

Для объявления и добавления новых правил в базу используется конструктор `defrule`.

Синтаксическая схема конструктора может быть представлена следующим образом:

(defrule *<имя_правила> [<необязательный_комментарии>]*
[<необязательное_определение_свойства_правила>]
<предпосылки > *; левая часть правила*
=> *; спец. символ*
<следствие> *; правая часть правила*
)

Имя правила должно являться значением типа symbol и не может быть зарезервированным словом языка CLIPS. В случае повторного объявления правила с одинаковым именем, старое правило будет удалено, даже если новое правило не возможно будет добавить вследствие синтаксической ошибки или по любым другим причинам

Комментарий является необязательным – обычно в нем описывают назначения правила. Комментарий должен являться значением типа string, значения комментария

сохраняется вместе с правилом.

Определение свойства правила состоит из ключевого слова `declare`, и последующего за ним указания свойства. У правила может быть два свойства – `salience` и `auto-focus`.

<определение-свойства-правила> =

*(**declare** <свойство-правила>)*

<свойство-правила> =

*(**salience** <целочисленное выражение>)*

или

*(**auto-focus** <логическое выражение>)*

Свойство `salience` позволяет пользователю назначать определенный приоритет для своих правил. Объявляемый приоритет должен быть выражением, имеющим целочисленное значение из диапазона от -10 000 до +10 000. Выражение, представляющее приоритет правила, может использовать глобальные переменные и функции. Однако лучше не использовать в этом выражении функций, имеющих побочное действие. В случае если приоритет правила явно не задан, ему присваивается значение по умолчанию – 0. Чем больше число, определяющее приоритет, тем выше приоритет у правила.

Значение приоритета может быть вычислено в одном из трех случаев:

1. при добавлении нового правила;
2. при активации правила;
3. на каждом шаге основного цикла выполнения правил.

Два последних варианта называются *динамическим приоритетом* (dynamic salience). По умолчанию значение приоритета вычисляется только во время добавления правила. Для изменения этой установки можно использовать диалоговое окно пункта меню Execution – Options. В появившемся диалоговом окне укажите необходимый режим вычисления приоритета с помощью раскрывающегося списка Salience Evaluation, как показано на рис. 6.6.

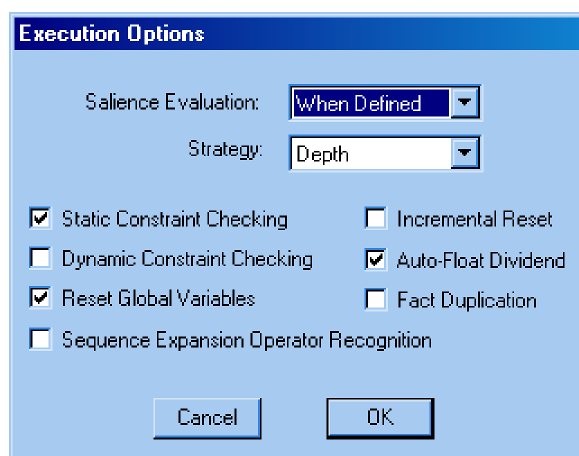

Рис.6.6. Окно настройки параметров механизма логического вывода

Свойство **auto-focus** используется при многомодульной архитектуре построения программ и его рассмотрение выходит за рамки данного пособия.

Предпосылка **или** ***левая часть правила*** задается набором условных элементов, который обычно состоит из условий, примененных к некоторым *образцам*. Заданный набор образцов используется системой для сопоставления с имеющимися фактами и объектами. Все условия в левой части правила объединяются с помощью неявного логического оператора and. Если в левой части правила не указан ни один условный элемент, CLIPS

автоматически подставляет условие-образец initial-fact. Таким образом, правило активизируется всякий раз при появлении в базе знаний факта initial-fact.

Следствие* или *правая часть правила содержит список действий, выполняемых при активизации правила механизмом логического вывода. Действия правила выполняются последовательно, но *тогда и только тогда, когда все условные элементы в левой части этого правила удовлетворены.*

Для разделения правой и левой части правил используется символ =>.

➢ *Совет: для рассмотрения примеров и выполнения практических заданий данного раздела удобнее пользоваться отдельным окном для написания команд (см. рис. 2) и затем передавать их в диалоговое окно посредством выделения текста и нажатия комбинации Ctrl-M.*

Рассмотрим пример программы:

```
(clear)
(defrule HelloWorldRule  "This is a comment"
     (initial-fact)    ; предпосылка
     =>
     (printout  t  crlf)
     (printout  t  "HELLO WORLD!"  crlf)
     (printout  t  crlf)
)
```

Наберите вышеприведенный листинг в окне ввода, которое показано на рис. 6.2, выделите текст и нажмите Ctrl-M. (рис. 6.7.) Выделенные команды будут выполнены в диалоговом окне. Там же будут выведены сообщения об ошибках. Если добавление правила произошло с ошибкой, внимательно проверьте синтаксис в блоке конструктора.

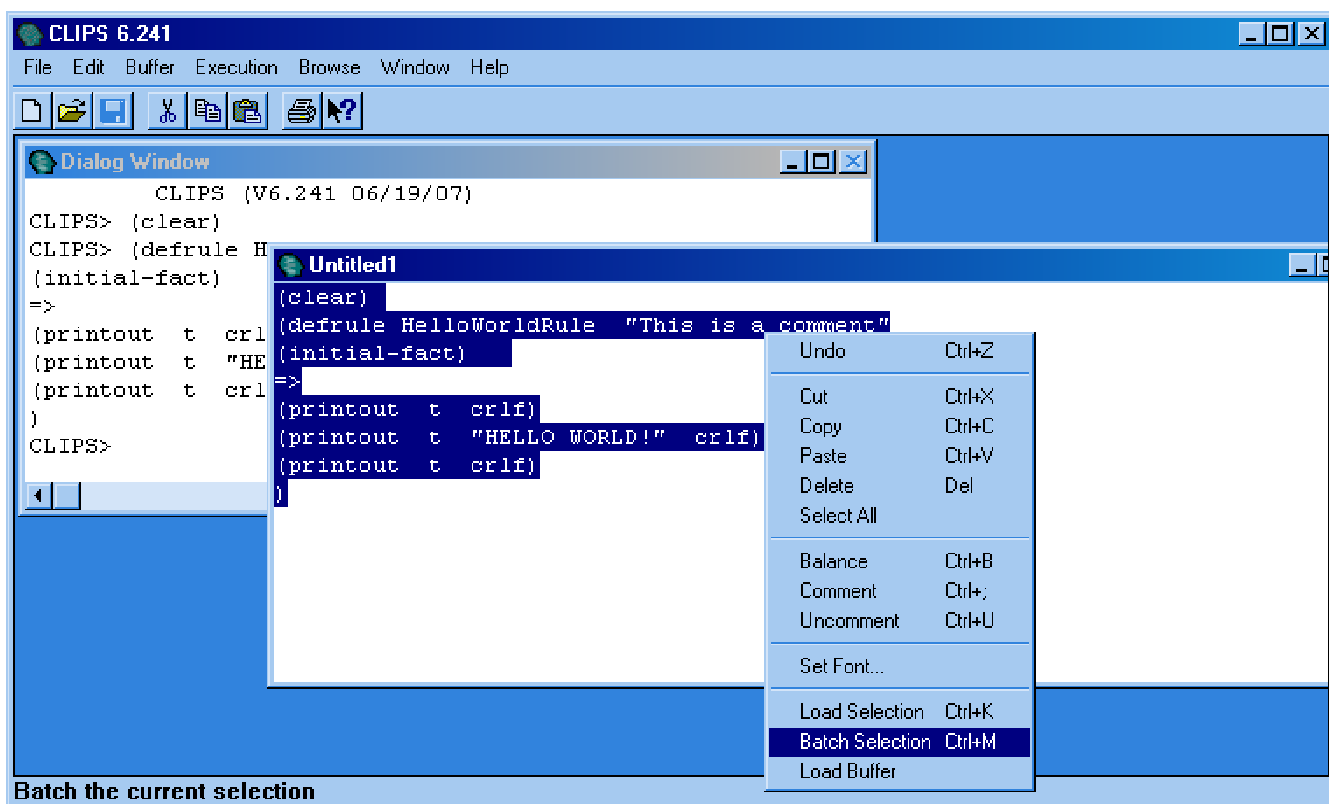

Рис. 6.7. Отправка на выполнение выделенной части программы.

Наличие правила в системе можно проконтролировать путем вызова пункта меню Browse – Defrule Manager.

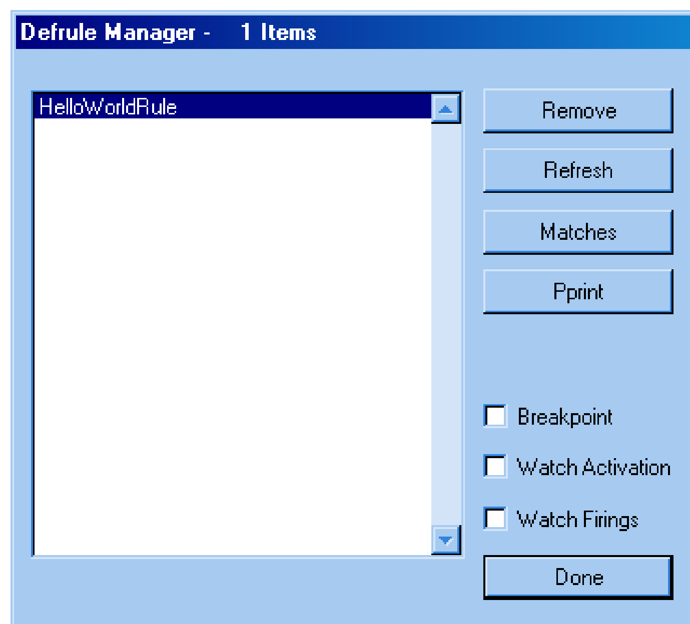

Рис. 6.8. Менеджер правил, внесенных в систему.

Если добавление правила прошло удачно и в менеджере фактов отображается строка HelloWorldRule (рис. 8), можно запустить механизм логического вывода. Для этого выполните в командном диалоговом окне последовательно две команды:

```
(reset)
(run)
```

Напомним, что команда (reset) очищает список фактов, а затем добавляет в него факты, объявленные конструкторами `deffacts`, включая предопределенный факт (initial-fact).

Команда `(run)` запускает процесс логического вывода. По сути, начинается проверка условий для правил, которые были введены в оболочку CLIPS ранее. Проверка предпосылки для нашего правила HelloWorldRule будет удачной – предпосылка будет удовлетворена, поскольку в системе существует факт (initial-fact), соответствующий образцу в нашем правиле. Подробнее о способе задания образцов речь пойдет ниже, сейчас же отметим, что для создания правила, активация которого произойдет при появлении в системе определенных фактов, достаточно просто перечислить эти факты в предпосылке правила.

Так как предпосылка удовлетворена, будет выполнено следствие правила, т.е. выполнена последовательность команд, записанная после оператора =>, и мы увидим в диалоговом окне приветственное сообщение «HELLO WORLD!». Наша программа познакомилась с миром.

Рассмотрим более сложный **пример**. Наберите в диалоговом окне, показанном на рис. 2. следующий код.

```
(clear)
(defrule FinalRule   ;   последнее правило
      (All Rules Activated)  ;требуем наличия факта для активации
      =>
      (printout t "All rules has been activated. THE END" crlf)
)
```

```
(defrule SecondRule   ; второе правило
     (First Rule Activated) ;   требуем наличия факта для активации
     =>
     (printout t "II rule activated. Adding fact..."  crlf)
     (assert                  ;    добавляем факт, означающий активацию правила №2
          (Second Rule Activated)
     )
)
(defrule ThirdRule   ;  третье правило
     (First Rule Activated)
     (Second Rule Activated) ; требуем наличия сразу двух фактов
     =>
     (printout t "III rule activated."  crlf)
     (assert               ;   добавляем факт, означающий активацию всех правил
          (All Rules Activated)
     )
)
(defrule FirstRule    ; первое правило
     =>                    ;предпосылки нет – активация при наличии inintal-fact
     (printout t "I rule activated. Adding fact..." crlf)
     (assert
          (First Rule Activated)
     )
)
```

Обратите внимания на комментарии, приведенные после символа точки с запятой (;). Выделите набранный код и передайте его на выполнение среде, нажав комбинацию Ctrl-M. (для лучшего контроля за процессом создания программы рекомендуется добавлять правила по мере их написания). В менеджере правил проверьте наличие четырех правил с именами FirstRule, SecondRule, ThirdRule и FinalRule.

Правила, заданные в данной программе, должны активироваться по следующей схеме: Первое правило – при запуске программы, второе – при активации первого, третье – при успешной активации первого и второго правила, финальное – при успешной активации всех трех правил.

Выполните запуск программы, дав команды `(reset)` и `(run)` и убедитесь, что правила выполняются в требуемой последовательности. Обратите внимание, что в коде программы определение правил записано в разброс – первым определено финальное правило, затем второе и т.д. Последовательность определения правил может быть важна лишь в случае наступления одинаковых условий выполнения для правил с одинаковым приоритетом. В этом случае применяется стратегия разрешения конфликтов. В нашем же случае последовательность определения правил абсолютно не важна, так как все правила имеют такие условия, которые обеспечивают отсутствие конфликтов и гарантируют желаемую последовательность выполнения. При старте логического вывода единственным правилом, условие которое удовлетворяется, является правило FirstRule, т.к. для активации других правил нет подходящих фактов. Внутри своего тела правило FirstRule добавляет в систему факт (First Rule Activated), говорящий о его активации.

Напомним, что этот факт – упорядоченный и состоит из трех значений типа symbol. При следующей проверке, в системе обнаруживается факт (First Rule Activated), который подходит для активации правила SecondRule, которое в свою очередь добавляет в систему свой факт.

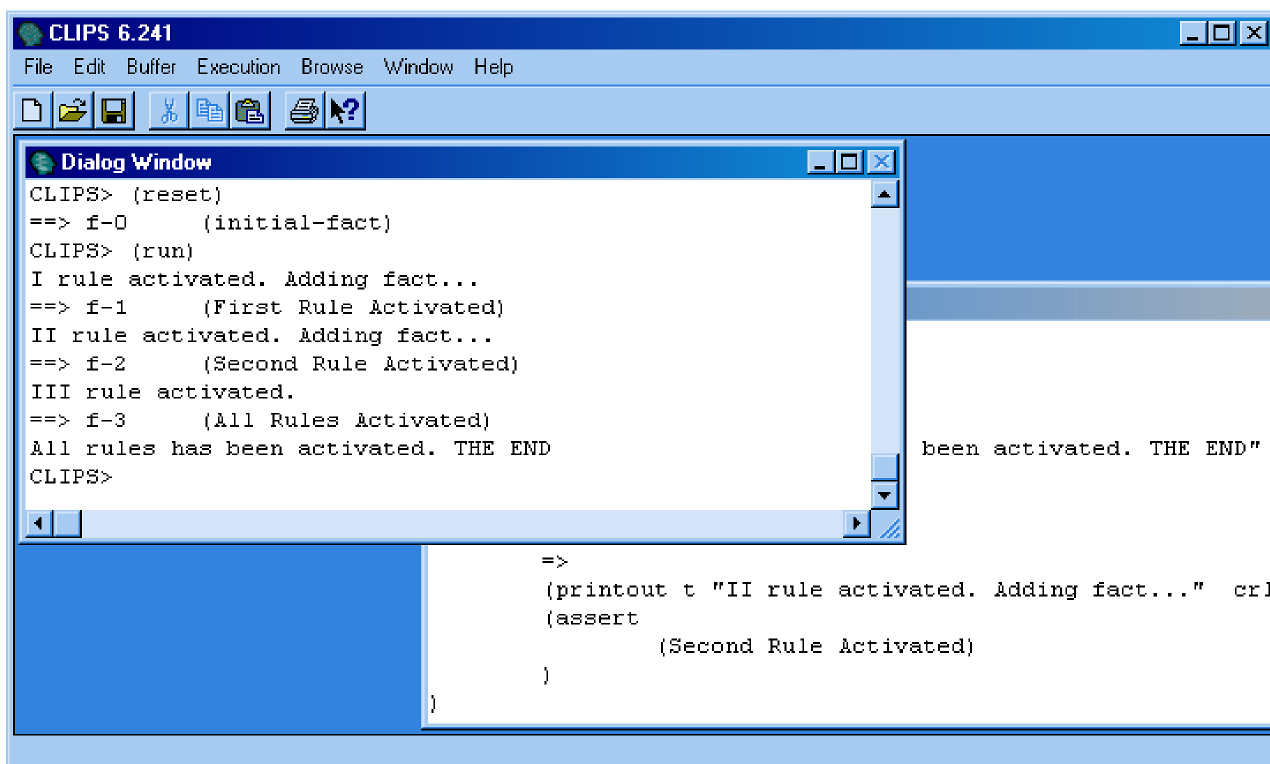

Рис. 6.9. Результат выполнения «программы о трех правилах»

Стратегия разрешения конфликтов

План решения задачи – это список всех правил, имеющих удовлетворенные условия при некотором, текущем состоянием списка фактов и объектов (и которые еще не были выполнены). Каждый модуль имеет свой собственный план решения задачи. Выполнение плана подобно стеку (верхнее правило плана всегда будет выполнено первым). Когда активируется новое правило, оно размещается в плане решения задачи, руководствуясь следующими факторами:

1. Только что активированное правило помещается выше всех правил с меньшим приоритетом и ниже всех правил с большим приоритетом.
2. Среди правил с одинаковым приоритетом используется текущая стратегия разрешения конфликтов для определения размещения среди других правил с одинаковым приоритетом.
3. Если правило активировано вместе с несколькими другими правилами, добавлением или исключением некоторого факта и с помощью шагов 1 и 2 нельзя определить порядок правила в плане решения задачи, то правило произвольным образом упорядочиваются вместе с другими правилами, которые были активированы. Заметьте, что в этом случае порядок, в котором правила были добавлены в систему, оказывает произвольный эффект на разрешения конфликта (который в высшей степени зависит от текущей реализации правил). Старайтесь не использовать произвольное упорядочивание правил при решении задач, в которых требуются точные результаты или объяснения полученных решений.

CLIPS поддерживает семь различных стратегий разрешения конфликтов: *стратегия глубины* (depth strategy), *стратегия ширины* (breadth strategy), *стратегия упрощения* (simplicity strategy), *стратегия усложнения* (complexity strategy), *LEX* (LEX strategy), *MEA* (MEA strategy) и *случайная стратегия* (random strategy). По умолчанию в CLIPS установлена стратегия глубины. Необходимая стратегия может быть указана в пункте

Options меню Execution, в появившемся диалоговом окне выберите необходимую стратегию с помощью раскрывающегося списка Strategy.

6.5.2.Использование образцов в правилах

В общем случае левая часть правила (предпосылка) содержит список условных элементов (conditional elements или CEs), которые должны удовлетворяться для активации правила. Существует восемь типов условных элементов, используемых в левой части правил: *CEs-образцы, test CEs, and CEs, or CEs, not CEs, exists CEs, forall CEs* и *logical CEs.*

- Образцы – это наиболее часто используемый условный элемент. Он содержит ограничения, которые служат для определения, удовлетворяет ли какой-либо факт из списка фактов заданному образцу.
- Условие test используется для оценки выражения, как части процесса сопоставления образов.
- Условие and применяется для определения группы условий, каждое из которой должно быть удовлетворено.
- Условие or – для определения одного условия из некоторой группы, которое должно быть удовлетворено.
- Условие not – для определения условия, которое не должно быть удовлетворено.
- Условие exists – для проверки наличия, по крайней мере, одного совпадения факта (или объекта) с некоторым заданным образцом.
- Условие logical позволяет выполнить добавление фактов и создание объектов в правой части правила, связанных с фактами и объектами, совпавшими с заданным образцом в левой часта правила (поддержка достоверности фактов в базе знаний)

Образцы

Образец состоит из списка *ограничений полей, групповых символов* (wildcards) и *переменных,* которые используются для поиска множества фактов, соответствующих желаемому образцу. Таким образом, образец как бы определяет маску, которой должны соответствовать факты. Если в списке фактов найден факт, соответствующий ограничениям, то условный элемент считается удовлетворенным.

Ограничения полей – это набор ограничений, которые используются для проверки простых полей или слотов объектов. Ограничения полей могут состоять только из одного символьного ограничения, однако, несколько ограничений можно соединять вместе. В дополнение к символьным ограничениям, CLIPS поддерживает три других типа ограничений: *объединяющие ограничения, предикатные ограничения* и *ограничения, возвращающие значения.*

Символьные ограничения – это ограничения, определяющие точное соответствие между полями факта и образцом. Символьное ограничение полностью состоит из констант, таких как вещественные и целые числа, значения типа symbol, строки или имена объектов

Например, в рассмотренном выше правиле:

```
(defrule FinalRule
      (All Rules Activated) ; образец с символьным ограничением.
      =>
      (printout t "All rules has been activated. THE END"
crlf)
)
```

строка (All Rules Activated) является условным элементом с символьным ограничением, накладывающим на необходимый для активации правила факт жесткие рамки

соответствия заданному шаблону. Другими словами, факт, который может активировать правило, должен быть точно таким же как указано в условном элементе. Символьные ограничения можно накладывать и на неупорядоченные факты. Например, для шаблона car, определенного в разделе 6.4.4, возможна такая запись правила для реакции на покупку Сергеем Петровым машины Opel черного цвета (предполагаем, что это событие будет ознаменовано добавлением факта в нашу систему):

```
(defrule PetrovHasBlackOpel
      ( car
            (color black)
            (model Opel)
            (owner Sergey Petrov)
      )
      =>
      (printout t “Sergey Petrov has a black Opel”)
)
```

Групповые символы для простых и составных полей

В CLIPS имеется два различных групповых символа, которые используются для сопоставления полей в образцах. CLIPS интерпретирует эти групповые символы как место для подстановки некоторых частей фактов. Групповой символ для простого поля записывается с помощью знака ?, который соответствует *одному любому значению*, сохраненному в заданном поле. Групповой символ составного поля записывается с помощью знака $? и соответствует последовательности значений, сохраненной в составном поле. Последовательность может быть пустая. Групповые символы для простых и составных полей могут комбинироваться в любой последовательности. Нельзя использовать групповой символ составного поля для простых полей. Рассмотрим **пример**:

```
(defrule FirstOrSecondRule
      (? Rule Activated) ; образец с групповым символом.
      =>
      (printout t "1st or 2nd rule has been activated" crlf)
)
```

Образец `(? Rule Activated)` активирует свое правило при появлении в системе любого факта, в котором на первом месте будет стоять любое значение, а на втором и третьем соответственно значения Rule и Activated. Таким образом, в рассмотренной выше программе «о трех правилах» факты (First Rule Activated) и (Second Rule Activated) будут удовлетворять образцу. Причем правило активируется при появлении каждого из фактов, т.е. в нашем примере – дважды. Обратите внимание, что факт (All Rules Activated) не сможет активировать правило FirstOrSecondRule, т.к. его второй элемент «Rule**s**» не совпадает с ограничением «Rule».

Рассмотрим образец с составным полем `(data 1 $?)`. В качестве примера приведем лишь несколько фактов, которые подходят под ограничение:

(data 1 blue red)
(data 1 blue 3 red)
(data 1 yellow red green 10)

и т.д. Но факт (data 2 red blue), уже не будет подходить под заданные ограничения, в силу различности второго элемента.

Если рассмотреть образец `($? Activated)` в примере про три правила, то можно убедиться, что он подойдет под все факты, которые добавляются по ходу выполнения программы. Мы можем записать правило

```
(defrule AnyOfThreeRule
      ($? Activated) ; образец с групповым символом.
```

```
        =>
        (printout t "Some rule has been activated" crlf)
)
```

которое будет выполняться каждый раз после активации первого, второго или третьего правила.

С составными фактами ситуация аналогичная. Так, правило:

```
(defrule SomebodyHasBlackCar
        (car
                (color black)
                (model ?)
                (owner $?)
        )
        =>
        (printout t "Somebody has black car" crlf)
)
```

активируется, если в системе появится факт, описывающий любой черный автомобиль любого владельца, причем количество элементов в составном слоте owner значения не имеет и может быть любым.

Переменные, связанные с простыми и составными полями

Групповые символы заменяют любые поля образца, и могут принимать какие угодно значения этих полей. Значение поля может быть связаны с переменными путем записи имени переменной непосредственно после знака группового символа. Имя переменной должно быть значением типа symbol и обязательно начинаться с буквы. В имени переменной не разрешается использовать кавычки, т. е. строка не может использоваться как имя переменной или ее часть. Переменные, которым были присвоены значения в результате выполнения сопоставления, можно использовать в правой части правила. Область видимости такой переменной ограничивается телом правила.

Рассмотрим пример с уже известным нам шаблоном для автомобилей. Введите в систему шаблон car из пункта 6.4.4, и добавьте список предопределенных фактов:

```
(deffacts  Cars
 (car (color black)(model Opel) (owner Sergey Petrov))
 (car (color red)(model Ferrari ) (owner Harrison Ford))
)
```

Добавте правило:

```
(defrule ListOfAutoOwners
        (car
                (color ?x)
                (model ?y)
                (owner $?z)
        )
        =>
        (printout t ?z " has " ?x ?y crlf)
)
```

При запуске программы, содержащей вышеприведенное правило, на экран будет выведена информация обо всех фактах car, которые присутствуют в списке, причем с указанием значений полей этих фактов

Пример ответа системы:

```
(Sergey Petrov) has black Opel
```

```
(Harrison Ford) has red Ferrari
```

При проверке фактов на соответствие условиям правила было выполнено присвоение подходящих значений переменным **?x**, **?y**, **?z**, после эти значения были использованы в правой части правила в функции вывода на экран. Ситуация с простыми фактами аналогичная.

Возможно и более сложное использование переменных в правилах. Например, правило

```
(defrule ListOfOwnersWithIdenticalCar
     (car (color ?a)(model ?b)(owner $?x))
     (car (color ?a)(model ?b)(owner $?y&~$?x))
     =>
     (printout t ?x " and " ?z " has " ?x ?y crlf)
)
```

активируется при обнаружении в списке фактов сведения о двух владельцах одинаковых машин. В качестве результата будет выдано соответствующее сообщение. Обратите внимание, что для корректного построение правила, мы использовали *связанное ограничение* `$?y&~$?x`. Рассмотрим его подробно: оно состоит из двух переменных `$?y` и `$?x`, символа отрицания `~` и символа объединения `&`. На обычном языке данное ограничение может быть прочитано так: «в составном поле owner должна находиться некоторая последовательность полей, которая будет присвоена переменной $?y, причем она не должна равняться последовательности, присвоенной переменной $?x»
Подробнее о связанных ограничениях и других типах образцов вы можете узнать в соответствующей литературе.

6.6.Глобальные переменные

Помимо фактов, CLIPS предоставляет еще один способ представления данных – глобальные переменные. В отличие от переменных, связанных со своим значением в левой части правила, глобальная переменная доступна везде после своего создания (а не только в правиле, в котором она получила свое значение). Глобальные переменные CLIPS подобны глобальным переменным в процедурных языках программирования. Переменные CLIPS слабо типизированы – фактически переменная может принимать значение любого примитивного типа CLIPS при каждом новом присваивании. Глобальные переменные могут использоваться для определения образцов, но лишь в том случае если они не подлежат сопоставлению, например, для логических ограничений.

Определить глобальную переменную позволяет конструктор defglobal, синтаксическая структура которого упрощенно может быть представлена в виде:

(defglobal
*?*имя_переменной* = <выражение>*
…
*[?*имя_переменной* = <выражение>]*
)

Пример использования:

```
(defglobal
     ?*x*  = 3
     ?*y*  =  ?*x*
     ?*z*  =  (+  ?*x*  ?*y*)
     ?*d*  =  7.8
```

```
        ?*e*   =  "string"
        ?*f*   =   symbol
)
```

Для присвоения значения переменной в процессе выполнения используется функция bind. Синтаксис вызова следующий:

(bind <имя-переменной> <выражение>)

Пример использования:

```
(bind ?*x* symbolvalue)
(bind ?*y*  (+ 6  8  1) )
(bind ?*z*  "string value" )
```

Обратите внимание, что типы значений, присваиваемых одной и той же переменной, могут меняться в процессе выполнения.

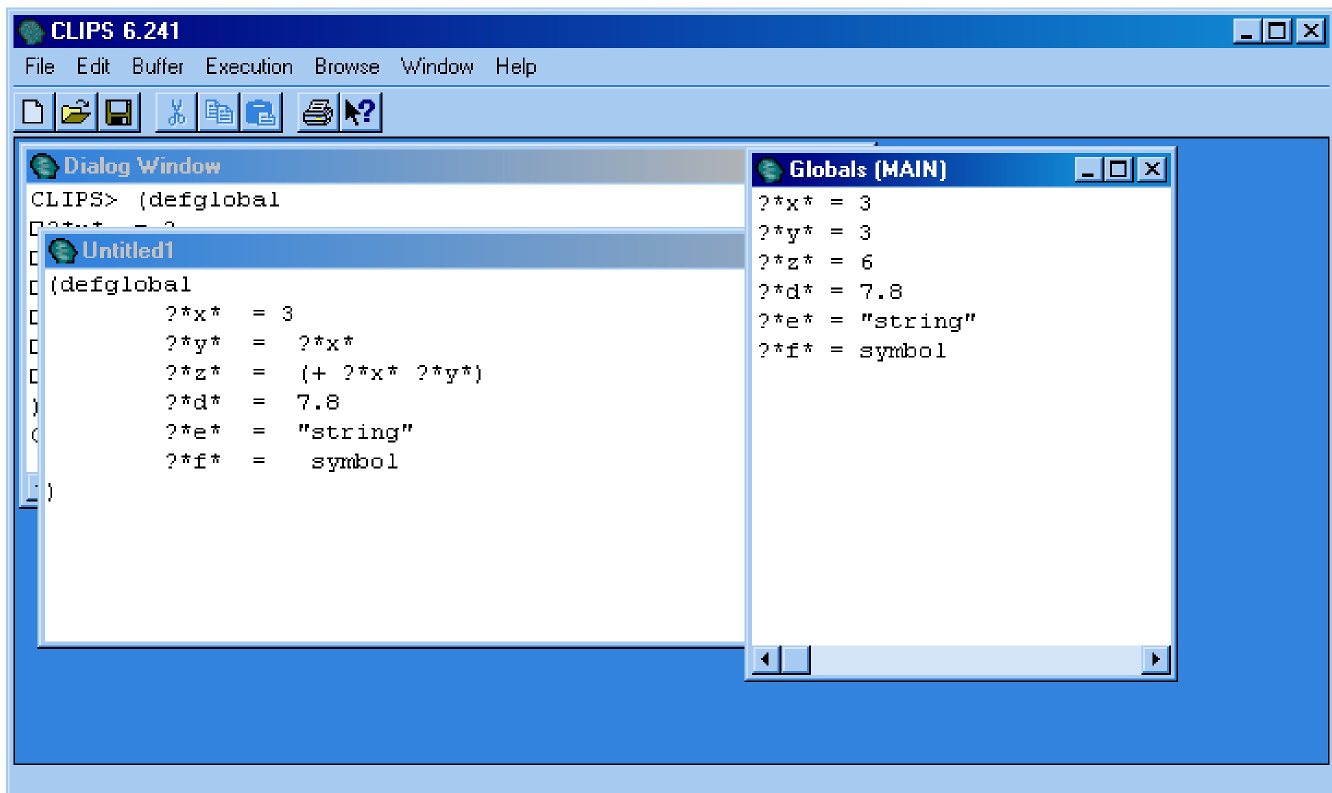

Рис. 6.10. Объявление глобальных переменных и окно для контроля их значений.

Для контроля над глобальными переменными может использоваться два визуальных инструмента: окно из пункта меню Window – Globals и пункт меню Browse – Defglobal Manager (рис. 6. 10.) Принцип их функционирования аналогичен подобным инструментам для фактов и шаблонов.

6.7.Пользовательские функции

Для определения пользовательских функций служит конструктор deffunction, синтаксис включает в себя 5 элементов:

- имя функции;
- необязательные комментарии;
- список из нуля или более параметров;
- необязательный символ групповых параметров для указания того, что функция может иметь переменное число аргументов;

- последовательность действий или выражений, которые будут выполнены (вычислены) по порядку во время исполнения тела функции.

(deffunction *<имя-функции> [<необяз-комментарий>]*
<обязательные-параметры> [<групповой-параметр>]
<действия>
)

В качестве примера рассмотрим простую функцию, выводящую на экран свой единственный аргумент:

```
(deffunction  PrintOneArgument (?a)
     (printout t ?a crlf)
)
```

Здесь PrintOneArgument – имя функции, (?a) – один принимаемый аргумент, (printout t ?a crlf) – тело функции, состоящее из одного вызова функции printout.

Вызов данной функции из тела программы может выглядеть так:

```
(PrintOneArgument  "Hello world")
```

Функция может принимать также и групповой параметр – набор некоторых значений. Например:

```
(deffunction  CountElementsInGroup ($?x)
     (printout  t  "Argument  x  consist  of"  (length  ?x)
"elements"          crlf)
)
```

Вызов такой функции может выглядеть так:

```
(CountElementsInGroup    (a,b,c,d))
```

Ответ системы:

```
Argument x consist of 4 elements
```

Функция возвращает значение, равное значению, которое вернуло последнее действие или вычисленное выражение во время выполнения функции. Если последнее действие не вернуло никакого результата, то выполняемая функция также не вернет результата (как в приведенном выше примере). Если функция не выполняет никаких действий, то возвращенное значение равно FALSE. В случае возникновения ошибки при выполнении очередного действия выполнение функции будет прервано и возвращенным значением также будет FALSE.

Рассмотрим пример функции, задающей пользователю вопрос и возвращающий ответ, введенный с клавиатуры:

```
(deffunction  ask   (?question)
     (printout  t   ?question)
     (bind  ?answer   (read))
?answer
)
```

Здесь имя функции – ask, принимаемый параметр ?question. При вызове функции значение переменной ?question выводится в качестве вопроса пользователю, затем выполняется функция bind связывающая переменную ?answer с информацией введенной с клавиатуры. Запрос ввода с клавиатуры выполняется функцией (read).

Пример вызова:

```
(ask  "How old are you?")
```

Для более удобного анализа пользовательских ответов может потребоваться, чтобы функция разрешала пользователю вводить не произвольный ответ, а один из заранее заданных вариантов. Задачу можно решить с помощью группового символа. Рассмотрим

следующую функцию, которая использует в своем коде функцию ask, рассмотренную выше.

```
(deffunction  ask-allowed   (?question   $?allowed)
      (bind   ?answer (ask ?question))
      (while (not (member  ?answer  $?allowed) )
       do
            (printout t "Reenter, please" crlf)
            (bind ?answer (ask ?question))
      )
?answer
)
```

Здесь групповой параметр $?allowed служит для задания набора допустимых ответов. Первой строкой тела выполняется связывание значения, возвращаемого функцией ask, с переменной ?answer. После чего используется циклическая структура

(while <условие> do

<оператор>

...

<оператор>

)

в которой вопрос задается до тех пор, пока пользователь не введет разрешенный ответ. Проверка соответствия выполняется функцией (member ?x $?y), которая возвращает TRUE в том случае, если внутри составного поля $?y найдет элемент идентичный значению переменной ?x.

➢ *Задания для самостоятельной работы:*

1. *Напишите функцию под именем aks-yes-no-question, которая принимает один параметр – текст вопроса, а возвращает два варианта ответа – «yes» или «no». Для выполнения задания используйте уже известную функцию ask-allowed.*
2. *Модифицируйте получившийся код так, чтобы при вводе положительного ответа на вопрос, функция возвращала TRUE, иначе – FALSE. Для этого используйте оператор проверки эквивалентности (eq <выражение> <выражение>), который возвращает TRUE в случае эквивалентность своих аргументов и FALSE в противном случае.*
3. *Модифицируйте функцию **ask** так, чтобы после вопроса на экран выводились возможные варианты ответа. Для этого определите в функции дополнительный групповой параметр. Не забудьте модифицировать функцию **ask-allowed** для корректной работы с новой модификацией функции **ask.***

6.8.Пример простейшей экспертной системы

Выше был рассмотрен минимальный набор средств необходимый для построения несложной экспертной системы. Для примера рассмотрим простую экспертную систему, которая в самых простых случаях поможет локализовать проблему с двигателем автомобиля. Допустим, что в результате беседы с экспертом мы получили следующие знания.

В простейшем случае проблемы с двигателем могут делиться на две категории «Работает нестабильно» и «Не заводится». Если двигатель не заводится, то он может либо совсем не раскручиваться, либо раскручиваться от аккумулятора но не заводиться. Если двигатель не раскручивается, то проблема может быть в аккумуляторной батарее или в стартере двигателя. Если же двигатель не заводится, то проблема может заключаться в отсутствии топлива или неисправности системы зажигания. Если двигатель работает

нестабильно, виной может слабая искра или неисправность свечей, а также неисправность в системе подачи топлива.

Не вдаваясь в технические подробности работы автомобильного двигателя (оставим это экспертам), запрограммируем наши знания используя продукционную модель, которую реализуем на языке CLIPS:

```
(clear)
(deffunction  ask   (?question  $?allowed)
     (printout  t  ?question  ?allowed)
     (bind  ?answer  (read))
?answer
)

(deffunction  ask-allowed   (?question $?allowed)
         (bind   ?answer (ask ?question))
     (while (not (member  ?answer $?allowed) )
      do
          (printout t "Reenter, please" crlf)
          (bind ?answer (ask ?question))
     )
?answer
)
(deffunction ask-yes-no   (?question)
     (bind  ?response   (ask-allowed  ?question yes no))
     (eq ?response yes)
)
(defrule EngineState
     (not (work ?))
          =>
     (if (eq (ask-allowed "What is the problem: 1-engine does not
work, 2-engine works unstable" 1 2) 1)
     then
          (assert(work doesnot))
     else
          (assert(work unstable))
     )
)
(defrule KindOfFail
      (work doesnot)
               =>
     (if (eq (ask-allowed "1 - Engine does not rotate, 2 - engine
rotates, but not start" 1 2) 1)
     then
          (assert(engine does-not-rotate))
     else
          (assert(engine rotates))
     )
)
(defrule CheckBatt
     (engine does-not-rotate)
               =>
               (assert(suggest  "Check  your  battery  or  engine
starter"))
```

```
)
(defrule EnoughFuel
     (engine rotates)
               =>
          (if  (ask-yes-no "Is it enough fuel?")
          then
               (assert(need to check ignition))
          else
               (assert(suggest "Add Fuel"))
          )
)
(defrule IgnitionCheck
     (not (ignition ?))
     (need to check ignition)
               =>
          (if  (ask-yes-no  "Check  ignition  system.  Is  there
strong spark?")
          then
               (assert(ignition ok))
          else
               (assert(ignition failed))
          )
)
(defrule SuggestOfIgnition
     (ignition failed)
     (engine rotates)
     =>
     (assert(suggest "Your engine does not start because of the
faulty of ignition system"))
)
(defrule UnstableIgnition
     (work unstable)
     (not (ignition ?))
     =>
     (assert(need to check ignition))
)

(defrule SuggestFuel
     (work unstable)
     (ignition ok)
     =>
     (assert (suggest "Your engine work unstable due to unstable
fuel supply"))
)
(defrule PrintSuggest
     (suggest ?x)
     =>
     (printout t ?x crlf)
)
(defrule NoSuggest
     (declare (salience -10))
     (not (suggest ?))
     =>
```

```
    (printout t "Sorry, there is no suggest." crlf)
)
```

Если задать набор вышеприведенных конструкторов в CLIPS и выполнить команды (reset) и (run), можно протестировать нашу простейшую экспертную систему-помощника для локализации проблем с двигателем автомобиля.

Литература к разделу 6

1. А. П. Частиков Разработка экспертных систем. Среда CLIPS. / Т. А. Гаврилова Д. Л.Белов – С.-П. «БХВ-Петербург» 2003. 393 с.
2. http://clipsrules.sourceforge.net/OnlineDocs.html
3. http://www.gsi.dit.upm.es/docs/clipsdocs/clipshtml/vol1.html
4. http://www.google.com.ua/search?q=CLIPS