ВВЕДЕНИЕ

Предметом изучения дисциплины "Интеллектуальные информационные системы" является исследование методик и инструментальных средств разработки информационных систем с элементами искусственного интеллекта. Наиболее распространенным типом таких систем являются так называемые экспертные системы.

Предлагается использовать для разработки экспертных систем язык представления знаний CLIPS. Первоначально аббревиатура CLIPS была названием языка – C Language Integrated Production System (язык С, интегрированный с продукционными системами), удобного для разработки баз знаний и макетов экспертных систем. CLIPS начал разрабатываться в космическом центре NASA в 1984 г. Теперь CLIPS представляет собой современный инструмент, предназначенный для создания экспертных систем (expert system tool). CLIPS состоит из интерактивной среды – экспертной оболочки со своим способом представления знаний, гибкого и мощного языка и нескольких вспомогательных инструментов. Сейчас, благодаря доброй воле своих создателей, CLIPS является абсолютно свободно распространяемым программным продуктом. Всем желающим доступен как сам CLIPS последней версии, так и его исходные коды. Официальный сайт CLIPS располагается по адресу: http://www.ghg.net/clips/CLIPS.html. Этот сайт поможет получить как сам CLIPS, так и всевозможный материал для его изучения и освоения (документацию, примеры, советы специалистов, исходные коды и многое другое).

Благодаря тому, что CLIPS является свободно распространяемым программным продуктом с доступными исходными кодами, в последнее время было выпущено множество программ и библиотек, усовершенствующих и дополняющих возможности CLIPS. Некоторые из этих продуктов являются собственностью выпустивших их компаний и предназначены для внутреннего использования или коммерческого распространения, другие, как и сам CLIPS, распространяются свободно. В качестве самых известных примеров подобных проектов можно привести DLL/OCX-библиотеку, позволяющую использовать механизм логического вывода CLIPS в ваших приложениях, Fuzzy CLIPS. CLIPS++, CLIPS code generator.

Таким образом, возникает возможность применять конструкции CLIPS в приложениях, разработанных в других программных средах, таких как Visual Studio.

Для получения практических навыков работы с CLIPS студентам предлагается выполнить две лабораторные работы. В первой должны быть использованы базовые конструкции исходного CLIPS для разработки прототипа экспертной системы. Во второй для реализации прототипа системы следует применять объектно-ориентированное расширение CLIPS под названием COOL.

ОПИСАНИЕ ОСНОВНЫХ КОНСТРУКЦИЙ ЯЗЫКА ПРЕДСТАВЛЕНИЯ ЗНАНИЙ CLIPS ВЕРСИИ 6.22

Экспертные системы, созданные с помощью CLIPS, могут быть запущены тремя основными способами:

– вводом соответствующих команд и конструкторов языка непосредственно в среду CLIPS;

– использованием интерактивного оконного интерфейса CLIPS (например для версий Windows или Macintosh);

– с помощью программ-оболочек, реализующих свой интерфейс общения с пользователем и использующих механизмы представления знаний и логического вывода CLIPS.

Windows-версия среды CLIPS полностью совместима с базовой спецификацией языка. Ввод команд осуществляется непосредственно в главное окно CLIPS. Однако по сравнению с базовой Windows-версия предоставляет множество дополнительных визуальных инструментов (например, менеджеры фактов или правил), значительно облегчающих жизнь разработчика экспертных систем.

Отличительной особенностью CLIPS являются конструкторы для создания баз знаний (БЗ):

defrule	– определение правил;
deffacts	– определение фактов;
deftemplate	– определение шаблона факта;
defglobal	– определение глобальных переменных;
deffunction	– определение функций;
defmodule	– определение модулей (совокупности правил);
defclass	– определение классов;
defintances	– определение объектов по шаблону, заданному defclass;
defmessagehandler	– определение сообщений для объектов;
defgeneric	– создание заголовка родовой функции;
defmethod	– определение метода родовой функции.

CLIPS поддерживает следующие типы данных: integer, float, string, symbol, external-address, fact-address, instance-name, instance-address.

Пример integer:	594,	23,	+51,	−17.
Пример float:	$594e^{2}$,	23.45,	+51.0,	$-17.5e^{-5}$.

String – это строка символов, заключенная в двойные кавычки.

Пример string: "expert", "Phil Blake", "состояние -0", "quote=\".

Факты – одна из основных форм представления данных в CLIPS (существует также возможность представления данных в виде объектов и глобальных переменных, но об этом речь пойдет позже). Каждый факт представляет собой определенный набор данных, сохраняемый в текущем списке фактов – рабочей памяти системы. Список фактов представляет собой универсальное хранилище фактов и является частью базы знаний. Объем списка фактов ограничен только памятью вашего компьютера. Список фактов хранится в оперативной памяти компьютера, но CLIPS предоставляет возможность сохранять текущий список в файл и загружать список из ранее сохраненного файла.

В системе CLIPS фактом является список неделимых (или атомарных) значений примитивных типов данных. CLIPS поддерживает два типа фактов – упорядоченные факты (ordered facts) и неупорядоченные факты или шаблоны (non-ordered facts или template facts). Ссылаться на данные, содержащиеся в факте, можно либо используя строго заданную позицию значения в списке данных для упорядоченных фактов, либо указывая имя значения для шаблонов. Упорядоченные факты состоят из поля, обязательно являющегося данным типа symbol и следующей за ним, возможно пустой, последовательности полей, разделенных пробелами. Ограничением факта служат круглые скобки:

(поле_типа_symbol [поле]*)

Так как упорядоченный факт для представления информации использует строго заданные позиции данных, то для доступа к ней пользователь должен знать, не только какие данные сохранены в факте, но и какое поле содержит эти данные. Неупорядоченные факты (или шаблоны) предоставляют пользователю возможность задавать абстрактную структуру факта путем назначения имени каждому полю. Для создания шаблонов, которые впоследствии будут применяться для доступа к полям факта по имени, используется конструктор deftemplate. Конструктор deftemplate аналогичен определениям записей или структур в таких языках программирования, как Pascal или C.

Конструктор deftemplate задает имя шаблона и определяет последовательность из нуля или более полей неупорядоченного факта, называемых также слотами. Слот состоит из имени, заданного значением типа symbol, и следующего за ним, возможно пустого, списка полей. Как и факт, слот с обеих сторон ограничивается круглыми скобками. В отличие от упорядоченных фактов слот неупорядоченного факта может жестко определять тип своих значений. Кроме того, слоту могут быть заданы значения по умолчанию. Синтаксис данного конструктора следующий:

(deftemplate <имя-шаблона> [<комментарии>] [<оперделение-слота>*])

Помимо фактов, CLIPS предоставляет еще один способ представления данных – глобальные переменные (globals). 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, таких как C или ADA. Однако, в отличие от переменных большинства процедурных языков программирования, глобальные переменные в CLIPS слабо типизированы. Фактически переменная может принимать значение любого примитивного типа CLIPS при каждом новом присваивании значения.

С помощью конструктора defglobal в среде CLIPS могут быть объявлены глобальные переменные и присвоены их начальные значения.

(defglobal [<имя-модуля>] <определение-переменной>*)
<определение-переменной> ::= <имя-переменной> = <выражение>
<имя-переменной> ::= ?*<значение-типа-symbol>*

CLIPS позволяет использовать произвольное количество конструкторов defglobal. Необязательный параметр <имя-модуля> указывает модуль, в котором должны быть определены конструируемые переменные. Если имя модуля не задано, то переменные будут помещены в текущий модуль.

Глобальные переменные применяются в любом месте, где могут быть использованы переменные, созданные в левой части правил с некоторыми исключениями. Во-первых, глобальные переменные не могут использоваться как параметры в конструкторах deffunction, defmethod или обработчиках сообщений. Во-вторых, глобальные переменные не могут использоваться для получения новых значений в левой части правил.

Неверно: (defrule example (fact ?*x*) =>).
Верно: (defrule example (fact ?y & :(> ?y ?*x*)) =>)
П р и м е р 1.

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

После выполнения данного конструктора в CLIPS появятся 4 глобальные переменные: x, y, z и q. Переменной x присваивается целое значение 3. Переменной y – значение, сохраненное в глобальной переменной x (т.е. 3). Переменной z – сумма значений x и y (т.е. 6).

Обратите внимание, что переменная y не является указателем на переменную x, просто их значения в данный момент совпадают. Если изменить значение x, значения переменных y и z, несмотря ни на что, останутся равными 3 и 6 соответственно.

Добавьте еще один конструктор defglobal, объявляющий переменные вещественного и текстового типа, а также переменную со значением типа symbol.

```
(defglobal
?*d* = 7.8
?*e* = "string"
```

?*f* = symbol)

При выполнении команды reset все глобальные переменные получают начальные значения, определенные в конструкторе.

Команда ppdefglobal выводит в диалоговое окно системы определение заданной глобальной переменной.

Имя глобальной переменной должно быть задано без вопросительного знака и символов *, т.е. name для переменной ?*name*.

Команда list-defglobals предназначена для отображения в диалоговом окне списка имен всех определенных в системе глобальных переменных.

(list-defglobals [<имя-модуля>])

Если необязательный параметр <имя-модуля> не указан, то данная команда выводит имена глобальных переменных, определенных в текущем модуле. Если параметр содержит имя конкретного модуля, команда list-defglobal выводит список переменных, определенных в заданном модуле. Допускается использование символа *. В этом случае команда выведет в диалоговое окно имена всех глобальных переменные, определенных во всех модулях системы.

Команда show-defglobals, в отличие от команды list-defglobals, выводит в диалоговое окно CLIPS не только имена глобальных переменных, но и их значения. В остальном эти две команды практически идентичны.

(show-defglobals [<имя-модуля>])

Команда undefglobal предназначена для удаления определенных пользователем глобальных переменных.

(undefglobal <имя-глобальной-переменной>)

В качестве параметра <имя-глобальной-переменной> допускается использование символа *. В этом случае команда попытается удалить все определенные пользователем глобальные переменные. Если глобальная переменная указана, например, в определении функции, удаление этой переменной закончится неудачей.

Существуют похожие функции для стандартных операций с конструкторами различных типов (не только для defglobal).

(bind <имя-переменной> <выражение>*)

Параметр выражения является необязательным. Если он не задан, то переменной будет установлено начальное значение, заданное в конструкторе defglobal. В случае, если выражение было задано, то его значение будет вычислено и результат присвоен переменной. Если было задано несколько выражений, все они будут вычислены, из их результатов будет составлено составное поле, которое будет присвоено глобальной переменной.

Функция bind возвращает значение false в случае, если переменной по какой-то причине не было присвоено никакого значения. В противном случае функция возвращает значение, присвоенное переменной.

Поскольку переменные в CLIPS слабо типизированы, типы значений, присваиваемые одной и той же переменной, в разные моменты времени могут не совпадать.

CLIPS поддерживает эвристическую и процедурную парадигму представления знаний. Для представления знаний в процедурной парадигме CLIPS предоставляет такие механизмы, как глобальные переменные, функции и родовые функции. Кроме того, существует такой способ представления знаний, как правила. Правила в CLIPS служат для представления эвристик или так называемых "эмпирических правил" действий при возникновении некоторой ситуации. Разработчик экспертной системы определяет набор правил, которые вместе работают над решением некоторой задачи. Правила состоят из предпосылок и следствия. Предпосылки называются также ЕСЛИ-частью правила, левой частью правила или LHS правила (left-hand side of rule). Следствие называется ТО-частью правила, правой частью правила или RHS правила (right-hand side of rule).

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 или некоторых созданных объектов, являющихся экземплярами классов, определенных пользователем. 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.

Следствие правила представляется набором некоторых действий, которые необходимо выполнить в случае, если правило применимо к текущей ситуации. Таким образом, действия, заданные вследствие правила, выполняются по команде механизма логического вывода, если все предпосылки правила удовлетворены. В случае, если в данный момент применимо более одного правила, механизм логического вывода использует так называемую стратегию разрешения конфликтов (conflict resolution strategy), которая определяет, какое именно правило будет выполнено. После этого CLIPS выполняет действия, описанные вследствие выбранного правила (которые могут оказать влияние на список применимых правил), и приступает к выбору следующего правила. Этот процесс продолжается до тех пор, пока список применимых правил не опустеет.

Чтобы лучше понять сущность правил в CLIPS, их можно представить в виде оператора IF-THEN, используемого в процедурных языках программирования, например, таких как Ada или C. Однако условия выражения IF-THEN в процедурных языках вычисляются тогда, когда поток управления программой непосредственно попадает на данное выражение путем последовательного перебора выражений и операторов, составляющих программу. В CLIPS, в отличии от этого, механизм логического вывода создает и постоянно модифицирует список правил, условия которых в данный момент удовлетворены. Эти правила запускаются на выполнение механизмом логического вывода. С этой стороны правила похожи на обработчики сообщений, присутствующие в таких языках программирования, как, например, Ada или SmallTalk.

Для добавления новых правил в базу знаний CLIPS предоставляет специальный конструктор defrule. В общем виде синтаксис данного конструктора можно представить следующим образом:

(defrule
<имя-правила>
[<комментарии>]
[<определение-свойства-правила>]
<предпосылки> ; левая часть правила
=>
<следствие> ; правая часть правила
)

Имя правила должно быть значением типа symbol. В качестве имени правила нельзя использовать зарезервированные слова CLIPS, которые были перечислены ранее. Определение правила может содержать объявление свойств правила, которое следует непосредственно после имени правила и комментариев.

В справочной системе и документации по CLIPS для обозначения предпосылок правила чаще всего используется термин "LHS of rule", а для обозначения следствия – "RHS of rule", поэтому в дальнейшем мы будем использовать аналогичную терминологию – левая и правая часть правила.

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. Правая часть правила содержит список действий, выполняемых при активизации правила механизмом логического вывода. Для разделения правой и левой части правил используется символ →. Правило не имеет ограничений на количество условных элементов или действий. Единственным ограничением является свободная память вашего компьютера. Действия правила выполняются последовательно, но тогда и только тогда, когда все условные элементы в левой части этого правила удовлетворены.

Если в левой части правила не указан ни один условный элемент, CLIPS автоматически подставляет условие образец initial-fact или initial-object.

После того как в систему добавлены все необходимые правила и приготовлены начальные списки фактов и объектов, CLIPS готов выполнять правила. В традиционных языках программирования точка входа, точка остановки и последовательность вычислений явно определяются программистом. В CLIPS поток исполнения программы совершенно не требует ясного определения. Знания (правила) и данные (факты и объекты) разделены, и механизм логического вывода, предоставляемый CLIPS, применяет данные к знаниям, формируя список применимых правил, после чего последовательно выполняет их. Этот процесс называется основным циклом выполнения правил (basic cycle of rule execution). Рассмотрим последовательность действий (шагов), выполняемых системой CLIPS в этом цикле в момент выполнения нашей программы:

1. Если был достигнут предел выполнения правил или не был установлен текущий фокус, выполнение прерывается. В противном случае для выполнения выбирается первое правила модуля, на котором был установлен фокус. Если в текущем плане выполнения нет удовлетворенных правил, то фокус перемещается по стеку фокусов и устанавливается на следующий модуль в списке. Если стек фокусов пуст, выполнение прекращается. Иначе шаг 1 выполняется еще один раз.
2. Выполнение действий, описанных в правой части выбранного правила. Использование функции return может менять положение фокуса в стеке фокусов. Число запусков данного правила увеличивается на единицу для определения предела выполнения правила.
3. В результате выполнения шага 2 некоторые правила могут быть активированы или дезактивированы. Активированные правила (т.е. правила, условия которых удовлетворяются в данный момент) помещаются в план решения задачи модуля, в котором они определены. Размещение в плане определяется приоритетом правила (salience) и текущей стратегией разрешения конфликтов (эти понятия будут описаны ниже). Дезактивированные правила удаляются из текущего плана решения задачи. Если для правила установлен режим просмотра активаций, то пользователь получит соответствующее информационное сообщение при каждой активации или дезактивации правила (режим просмотра активаций можно установить с помощью диалогового окна Watch options. Для этого выберите пункт Watch в меню Execution и установите флажок Activations).
4. Если установлен режим динамического приоритета (dynamic salience), то для всех правил из текущего плана решения задачи вычисляются новые значения приоритета. После этого цикл повторяется с шага 1.

Свойства правил позволяют задавать характеристики правил до описания левой части правила. Для задания свойства правила используется ключевое слово declare. Однако правило может иметь только одно определение свойства, заданное с помощью declare.

<определение-свойства-правила> :: = (declare <свойство-правила>)
<свойство-правила> :: = (salience <целочисленное выражение>) | (auto-focus TRUE | FALSE)

Свойство правила salience позволяет пользователю назначать 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 старайтесь не указывать в этом выражении функций, имеющих побочное действие. В случае, если приоритет правила явно не задан, ему присваивается значение по умолчанию, т.е. 0.

Значение приоритета может быть вычислено в одном из трех случаев: при добавлении нового правила, при активации правила и 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 команду set-salience-evaluation.

Каждый метод вычисления приоритета содержит в себе предыдущий (т.е. если приоритет вычисляется на каждом шаге основного цикла выполнения правил, то он вычисляется и при активации правила, а также при его добавлении в систему).

Свойство auto-focus позволяет автоматически выполняться команде focus при каждой активации правила.

План решения задачи – это список всех правил, имеющих удовлетворенные условия при некотором, текущем состоянии списка фактов и объектов (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 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 упорядочивается вместе с другими правилами, которые были активированы. Заметьте, что в этом случае порядок, в котором правила были добавлены в систему, оказывает произвольный эффект на разрешение конфликта (который в высшей степени зависит от текущей реализации правил).

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. Текущая стратегия может быть установлена командой set-strategy (которая переупорядочит текущий план решения задачи, базируясь на новой стратегии).

• Стратегия глубины. Только что активированное правило помещается выше всех правил с таким же приоритетом. Например, допустим, что факт – А активировал правила 1 и 2 и факт Б активировал правило 3 и правило 4, тогда, если факт А добавлен перед фактом Б, в плане решения задачи правила 3 и 4 будут располагаться выше, чем правила 1 и 2. Однако позиция правила 1 относительно правила 2 и правила 3 относительно правила 4 будет произвольной.

• Стратегия ширины. Только что активированное правило помещается ниже всех правил с таким же приоритетом. Например, допустим, что факт А активировал правила 1 и 2 и факт Б активировал правила 3 и 4, тогда, если факт А добавлен перед фактом В, в плане решения задачи правила 1 и 2 будут располагаться выше, чем правила 3 и 4. Однако позиция правила 1 относительно правила 2 и правила 3 относительно правила 4 будет произвольной.

• Стратегия упрощения. Между всеми правилами с одинаковым приоритетом только что активированные правила размещаются выше всех активированных правил с равной или большей определенностью (specificity). Определенность правила вычисляется по числу сопоставлений, которые нужно сделать в левой части правила. Каждое сопоставление с константой или заранее связанной с фактом переменной добавляет к определенности единицу. Каждый вызов функции в левой части правила, являющийся частью условных элементов : , = или test, также добавляет к определенности единицу. Логические функции and, or и not не увеличивают определенность правила, но их аргументы могут это сделать. Вызовы функций, сделанные внутри функций, не увеличивают определенность правила. Например, следующее правило имеет определенность, равную 5.

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

Сравнение заранее связанной переменной >x с константой и вызовы функций numberp, < и > добавляют единицу к определенности правила. В итоге получаем определенность, равную 5. Вызовы функций and и + не увеличивают определенность правила.

• Стратегия усложнения. Между правилами с одинаковым приоритетом только что активированные правила размещаются выше всех активированных правил с равной или меньшей определенностью.

• Стратегия LEX. Между правилами с одинаковым приоритетом только что активированные правила размещаются с применением одноименной стратегии, впервые использованной в системе OPSS. Для определения места активированного правила в плане решения задачи используется "новизна" образца, который активировал правило. CLIPS маркирует каждый факт или объект временным тегом для отображения относительной новизны каждого факта или объекта в системе. Образцы, ассоциированные с каждой активацией правила, сортируются по убыванию тегов для определения местоположения правила. Активация правила, выполненная более новыми образцами, располагается перед активацией, осуществленной более поздними образцами. Для определения порядка размещения двух активаций правил, поодиночке сравниваются отсортированные временные теги для этих двух активаций, начиная с наибольшего временного тега. Сравнение продолжается до тех пор, пока не остается одна активация с наибольшим временным тегом. Эта активация размещается выше всех остальных в плане решения задачи.

Если активация некоторого правила выполнена большим числом образцов, чем активация другого правила, и все сравниваемые временные теги одинаковы, то активация другого с большим числом временных тегов помещается перед активацией с меньшим. Если две активации имеют одинаковое количество временных тегов и их значения равны, то правило с большей определенностью помещается перед активацией с меньшей. В отличие от системы OPSS, условный элемент not в CLIPS имеет псевдовременной тег, который также используется в данной стратегии разрешения конфликтов. Временной тег условного элемента not всегда меньше, чем временной тег образца.

В качестве примера рассмотрим следующие шесть активаций правил, приведенные в LEX-порядке (запятая в конце строки активации означает наличие логического элемента not). Учтите, что временные теги фактов не обязательно равны индексу, но если индекс факта больше, то больше и его временной тег. Для данного примера примем, что временные теги равны индексам.

rule-6: f-1, f-4
rule-5: f-1, f-2, f-3,
rule-1: f-1, f-2, f-3
rule-2: f-3, f-1
rule-4: f-1, f-2,
rule-3: f-2, f-1

Далее показаны те же активации с индексами фактов в том порядке, в котором они сравниваются стратегией LEX.

rule-6: f-4, f-1
rule-5: f-3, f-2, f-1,
rule-1: f-3, f-2, f-1
rule-2: f-3, f-1
rule-4: f-2, f-1,
rule-3: f-2, f-1

• Стратегия MEA. Между правилами с одинаковым приоритетом только что активированные правила размещаются с использованием одноименной стратегии, впервые использованной в системе OPSS. Основное отличие стратегии MEA от LEX в том, что в стратегии MEA не производится сортировка образцов, активировавших правило. Сравниваются только временные теги первых образцов двух активаций. Активация с большим тегом помещается в план решения задачи перед активацией с меньшим. Если обе активации имеют одинаковые временные теги, ассоциированные с первым образцом, то для определения размещения активации в плане решения задачи используется стратегия LEX. Как и в стратегии LEX, условный элемент not имеет псевдовременной тег.

В качестве примера рассмотрим следующие шесть активаций, приведенные в MEA-порядке (запятая на конце активации означает наличие логического элемента not).

rule-2: f-3, f-1
rule-3: f-2, f-1
rule-6: f-1, f-4
rule-5: f-1, f-2, f-3,
rule-1: f-1, f-2, f-3
rule-4: f-1, f-2,

• Случайная стратегия. Каждой активации назначается случайное число, которое используется для определения местоположения среди активаций с одинаковым приоритетом. Это случайное число сохраняется при смене стратегий, таким образом, тот же порядок воспроизводится при следующей установке случайной стратегии (среди активаций в плане решения задачи, когда стратегия заменена на исходную).

Для описания функций пользователя служит следующий конструктор.

(deffunction <имя-функции>
[<комментарии>] <обязательные-параметры>
[<групповой-параметр>] <действия>)
<обязательные-параметры> ::= <выражение-простое-поле>
<групповой-параметр> : := <выражение-составное-поле>

Синтаксис конструктора deffunction 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 в момент вызова функции.

В зависимости от того, задан ли групповой параметр, функция, созданная конструктором, может принимать точное число параметров или число параметров не меньшее, чем некоторое заданное. Обязательные параметры определяют минимальное число аргументов, которое должно быть передано функции при ее вызове. В действиях функции можно ссылаться на каждый из этих параметров как на обычные переменные, содержащие простые значения. Если был задан групповой параметр, то функция может принимать любое количество аргументов большее или равное минимальному числу. Если групповой параметр не задан, то функция может принимать число аргументов точно равное числу обязательных параметров. Все аргументы функции, которые не соответствуют обязательным параметрам, группируются в одно значение составного поля. Ссылаться на это значение можно, используя символ группового параметра. Для работы с групповым параметром могут использоваться стандартные функции CLIPS, предназначенные для работы с составными полями, такие как length и nth. Определение функции может содержать только один групповой параметр.

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

```
CLIPS> (print-args a b c d)
a b and 2 extras: (c d)
CLIPS> (print-args a)
[ARGACCESS4] Function print-args expected at least 2 argument(s)
CLIPS>
```

В данном примере с помощью конструктора deffunction определяется функция print-args, которая принимает два обязательных параметра: ?a и ?b, и имеет групповой параметр $?c. Функция выводит на экран свои обязательные параметры, а также число полей в составном параметре и его содержимое.

При вызове функции интерпретатор CLIPS последовательно выполняет действия в порядке, заданном конструктором. Функция возвращает значение, равное значению, которое вернуло последнее действие или вычисленное выражение. 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.

Функции могут быть само- и взаимно рекурсивными. Саморекурсивная функция просто вызывает сама себя из списка своих собственных действий. В качестве примера можно привести функцию, вычисляющую факториал.

```
(deffuction factorial (?a)
 (if (or  (not  (integerp ?a))  (< ?a 0))  then
(printout t  "Factorial error! " crlf)
else
   (if (= ?a 0)  then 1 else
(* ?a  (factorial  (- ?a 1))))))
```

Взаимная рекурсия между двумя функциями требует предварительного объявления одной из этих функций. Для предварительного объявлении функции в CLIPS используется конструктор deffunction с пустым списком действий. В следующем примере функция foo предварительно объявлена и таким образом может быть вызвана из функции bar. Окончательная реализация функции foo выполнена конструктором после объявления функции bar.

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

Команда ppdeffunction выводит определение заданной функции на экран.

(ppdeffunction <имя-функции>)

Команда list-deffunctions предназначена для отображения в диалоговом окне списка имен всех определенных в системе функций.

(list-deffunctions)

Для удаления функций, определенных пользователем с помощью конструкторов deffunction, предназначена команда undeffunction.

(undeffunction <имя-функции>)

В качестве параметра <имя-функции> возможно использование символа *. В этом случае команда попытается удалить все определенные пользователем функции. Удаление функции закончится неудачей, если выбранная функция в данный момент используется или выполняется (например, правилом).

CLIPS поддерживает следующие процедурные функции, реализующие возможности ветвления, организации циклов в программах и т.п.:

If	– оператор ветвления;
While	– цикл с предусловием;
loop-for-count	– итеративный цикл;
prong	– объединение действий в одной логической команде;
prong$	– выполнение набора действий над каждым элементом поля;
return	– прерывание функции, цикла, правила и т.д.;
break	– то же, что и return, но без возвращения параметров;
switch	– оператор множественного ветвления;
bind	– создание и связывание переменных.

Среди логических функций (возвращающих значения true или false) следует выделить такие группы:

- функции булевой логики: and, or, not;
- функции сравнения чисел: =, ≠, >, ≥, <, ≤;
- предикативные функции для проверки принадлежности проверяемому типу: integerp, floatp, stringp, symbolp, pointerp (относится ли аргумент к xternal-address), numberp (относится ли аргумент к integer или float), lexemerp (относится ли аргу-

мент к string или symbol), evenp (проверка целого на четность), oddp (проверка целого на нечетность), multifildp (является ли аргумент составным полем);

- функции сравнения по типу и по значению: eq, neq.

Среди математических функций следует выделить следующие группы:

- Стандартные: +, –, *, /, max, min, div (целочисленное деление), abs (абсолютное значение), float (преобразование в тип float), integer (преобразование в тип integer);
- Расширенные: sqrt (извлечение корня), round (округление числа), mod (вычисление остатка от деления);
- Тригонометрические: sin, sinh, cos, cosh, tan, tanh, acos, acosh, acot, acoth, acsc, acsch, asec, asech, asin, asinh, atan, atanh, cot, coth, csc, csch, sec, sech, deg-grad (преобразование из градусов в секторы), deg-rad (преобразование из градусов в радианы), grad-deg (преобразование из секторов в градусы), rad-deg (преобразование из радиан в градусы);
- Логарифмические: log, log10, exp, pi.

Среди функций работы со строками следует назвать функции:

str-cat	– объединение строк;
sym-cat	– объединение строк в значение типа symbol;
sub-string	– выделение подстроки;
str-index	– поиск подстроки;
eval	– выполнение строки в качестве команды CLIPS;
build	– выполнение строки в качестве конструктора CLIPS;
upcase	– преобразование символов в символы верхнего регистра;
lowcase	– преобразование символов в символы нижнего регистра;
str-compare	– сравнение строк;
str-length	– определение длины строки;
check-syntax	– проверка синтаксиса строки;
string-to-field	– возвращение первого поля строки.

Функции работы с составными величинами являются одной из отличительных особенностей языка CLIPS. В их число входят:

create$	– создание составной величины;
nth$	– получение элемента составной величины;
members	– поиск элемента составной величины;
subset$	– проверка одной величины на подмножество другой;
delete$	– удаление элемента составной величины;
explode$	– создание составной величины из строки;
implode$	– создание строки из составной величины;
subseq$	– извлечение подпоследовательности из составной величины;
replace$	– замена элемента составной величины;
insert$	– добавление новых элементов в составную величину;
first$	– получение первого элемента составной величины;
rest$	– получение остатка составной величины;
length$	– определение числа элементов составной величины;
delete-member$	– удаление элементов составной величины;
replace-member$	– замена элементов составной величины.

Функции ввода-вывода используют следующие логические имена устройств:

stdin	– устройство ввода;
stdout	– устройство вывода;
wclips	– устройство, используемое как справочное;
wdialog	– устройство для отправки пользователю сообщений;
wdisplay	– устройство для отображения правил, фактов и т.п.;
werror	– устройство вывода сообщений об ошибках;
wwarning	– устройство для вывода предупреждений;
wtrase	– устройство для вывода отладочной информации.

Собственно функции ввода-вывода следующие:

open	– открытие файла (виды доступа r, w, r+, a, wb);

close – закрытие файла;
printout – вывод информации на заданное устройство;
read – ввод данных с заданного устройства;
readline – ввод строки с заданного устройства;
format – форматированный вывод на заданное устройство;
rename – переименование файла;
remove – удаление файла.

Среди двух десятков команд CLIPS следует назвать основные команды при работе со средой CLIPS:

load – загрузка конструкторов из текстового файла;
load+ – загрузка конструкторов из текстового файла без отображения;
reset – сброс рабочей памяти системы CLIPS;
clear – очистка рабочей памяти системы;
run – выполнение загруженных конструкторов;
save – сохранение созданных конструкторов в текстовый файл;
exit – выход из CLIPS.

CLIPS предоставляет возможность разбиения базы данных и решения задачи на отдельные независимые модули. Для создания таких модулей служит конструктор defmodule. С помощью модулей можно группировать вместе отдельные элементы базы знаний и управлять процессом доступа к этим элементам во время решения некоторой задачи. Подобный процесс управления доступом к данным напоминает механизмы пространства имен, используемый в C++, и глобальных и локальных областей видимости в языках C и Ada. Однако, в отличие от механизмов в перечисленных выше языках, области видимости в CLIPS строго иерархичны и однонаправлены: если модуль A может видеть данные модуля B, это не означает, что модуль B может видеть данные модуля A. С помощью управления с ограничением доступа к данным, содержащимся в различных модулях, при решении сложных задач модули могут реализовывать концепцию доски объявлений (blackboard strategy – стратегия решения задач с использованием разнородных источников знаний, взаимодействующих через общее информационное поле). В этом случае отдельный модуль позволяет видеть правилам из других модулей строго определенный набор фактов и объектов. Кроме того, модули используются для управления потоком вычисления правил.

(defmodule <имя-молуля> [<комментарий>]
<спецификации-импорта-экспорта>*)
<спецификация-импорта-экспорта> : :=
(export <элемент-спецификация>) |
(import <имя-модуля> <элемент-спецификации>)
<элемент-спецификации> : := ?ALL | ?NONE |
<конструктор> ?ALL | <конструктор> ?NONE |
<конструктор> <имя-конструктора>
<конструкция>:: = deftemplate | defclass |
defglobal | deffunction | defgeneric

После своего создания модуль не может быть переопределен или удален (за исключением системного модуля MAIN, который пользователь может один раз переопределить). Единственный способ удалить существующий модуль – выполнить команду clear. Во время запуска системы и при вызове команды clear CLIPS автоматически создает предопределенный системный модуль: (defmodule MAIN).

Явное задание модуля выполняется с помощью имени модуля, разделенного с именем конструкции при помощи двойного двоеточия :: . Имя модуля и символ :: называются спецификатором модуля (module specifier). Например, запись MAIN::find-stuff ссылается на конструкцию find-stuff из модуля MAIN.

Неявное задание модуля выполняется с помощью установки текущего активного модуля. Текущий модуль меняется при каждом определении нового модуля или при вызове функции set-current-module.

В CLIPS факты и объекты принадлежат не тому модулю, в котором они были созданы, а тому, в котором был определен соответствующий конструктор deftemplate или defclass. Таким образом, факты и объекты становятся видимыми в тех модулях, которые импортируют соответствующие конструкции deftemplate или defclass. Это позволяет разбивать базу знаний таким образом, чтобы правила или другие конструкции могли видеть только необходимые им факты и объекты. Далее приведен пример создания и взаимосвязи двух модулей.

CLIPS> (clear)
CLIPS> (defmodule A (export deftemplate foo bar))
CLIPS> (deftemplate A::foo (slot x))
CLIPS> (deftemplate A::bar (slot y))
CLIPS> (deffacts A::info (foo (x 3)) (bar (y 4)))
CLIPS> (defmodule B (import A deftemplate foo))
CLIPS> (reset)
CLIPS> (facts A)

```
f-1   (foo (x 3) )
f-2   (bar (y 4) )
For a total of 2 facts.
CLIPS> (facts B)
f-1   (foo (x 3) )
For a total of 1 fact.
CLIPS>
```

Таким образом, имя объекта можно указать тремя способами.

<имя-объекта> ::= [<имя>] |
[::<имя>] |
[<модуль> :: <имя>]

Скобки являются обязательным синтаксисом CLIPS.

Каждый модуль имеет свой собственный процесс сопоставления образцов для своих правил и свой план решения задачи. По команде run начинает выполняться план решения задачи модуля, на который в данный момент установлен фокус. Команды reset и clear автоматически устанавливают фокус на модуль MAIN. Выполнение правил продолжается до тех пор, пока в плане решения задачи не останется применимых правил, и другой модуль не получит фокус, либо правая часть одного из выполняемых правил не вызовет функцию return. После того как в плане решения задачи модуля, имеющего фокус, заканчиваются правила, текущей модуль удаляется из стека фокусов (focus stack) и находящийся в стеке следующий модуль получает фокус. Перед выполнением правила текущим становится модуль, в котором данное правило определено. Управлять стеком фокусов можно с помощью команды focus.

В завершение следует иметь в виду, что CLIPS может неудовлетворительно работать в реальном времени, когда потребуется время реакции менее 0,1 с. В этом случае надо исследовать на разработанном прототипе механизмы вывода для всего множества правил предметной области на различных по производительности компьютерах. Как правило, современные персональные компьютеры обеспечивают работу с продукционными системами объемом 1000 – 2000 правил в реальном времени. Web-ориентированные средства на базе JAVA (системы Exsys Corvid, JESS) являются более медленными, чем, например, CLIPS 6 или OPS-2000. Поэтому CLIPS – лучший на сегодня выбор для работы в реальном времени среди распространяемых свободно оболочек ЭС, разработанных на C++.

РАЗРАБОТКА ПРОТОТИПА ЭКСПЕРТНОЙ СИСТЕМЫ AUTO

Рассмотрим пример создания диагностической экспертной системы, которая позволяет установить причину неисправности автомобиля и выдать соответствующую рекомендацию.

Разработку любой экспертной системы следует начинать с выявления основных сущностей, имеющих значение при решении конкретной задачи и законов, скорее всего эмпирических, действующих над этими сущностями.

В результате работы с экспертом были установлены следующие эмпирические правила:

1. Двигатель обычно находится в одном из 3 состояний: он может работать нормально, работать неудовлетворительно или не заводиться.
2. Если двигатель работает нормально, то это означает, что он нормально вращается, система зажигания и аккумулятор находятся в норме и никакого ремонта не требуется.
3. Если двигатель запускается, но работает ненормально, то это говорит, по крайней мере, о том, что аккумулятор в порядке.
4. Если двигатель не запускается, то нужно узнать, пытается ли он вращаться. Если двигатель вращается, но при этом не заводится, то это может говорить о наличии плохой искры в системе зажигания. Если двигатель даже не пытается заводиться, то это говорит о том, что искры нет в принципе.
5. Если двигатель не заводится, но вращается, нужно проверить наличие топлива. Если топлива нет – то, скорей всего, для ремонта машины нужно просто заправиться.
6. Если двигатель не заводится, нужно также проверить, заряжен ли аккумулятор, если нет, то его следует зарядить.
7. Если двигатель не заводится и существует вероятность плохой искры в системе зажигания, то необходимо проверить контакты. Контакты могут быть в одном из трех состояний – чистые, опаленные и грязные, в случае опаленных контактов их необходимо заменить, в случае если контакты грязные, их достаточно просто почистить.
8. Если двигатель не заводится, искры нет и аккумулятор заряжен, то нужно проверить катушку зажигания на электрическую проводимость. В случае, если ток не проходит через катушку, то ее необходимо заменить. Если катушка зажигания в порядке, значит необходимо заменить распределительные провода.
9. Если двигатель запускается, но при этом ведет себя инертно, не сразу реагирует на подачу топлива, то необходимо прочистить топливную систему.
10. Если двигатель запускается, но происходят перебои с зажиганием, то это говорит о наличии плохой искры в системе зажигания, для устранения данной неисправности необходимо отрегулировать зазоры между контактами.
11. Если двигатель запускается и стучит, то необходимо отрегулировать зажигание.
12. Если двигатель запускается, но не развивает нормальной мощности, то это может говорить об опаленных или загрязненных контактах (см. правило 7).
13. Возможны ситуации, когда состояние двигателя нельзя описать приведенными выше факторами и машине может потребоваться более детальный анализ состояния.

Из приведенных выше правил можно выделить следующие сущности, имеющие значение при решении задачи.

– Во-первых, для решения задачи экспертной системе необходимо знать, в каком состоянии находится машина, диагностика которой производится. Эксперт выделил три возможных состояния: нормальная работа двигателя, двигатель работает неудовлетворительно, не заводится (см. правило 1).

– Во-вторых, большинство приведенных правил, помимо состояния двигателя в целом, используют понятие состояния вращения двигателя. Согласно этим правилам двигатель может находиться в одном из двух состояний, которые определяются в зависимости от того, способен он вращаться (работать) или нет.

– В-третьих, в некоторых правилах (см. правила 4, 7, 8, 10) используется понятие состояния системы зажигания. Система зажигания может быть в одном из трех состояний: нормальное состояние, нерегулярная работа и нерабочее состояние.

– В-четвертых, в правилах 6 и 8 используется понятие состояния аккумулятора. Аккумулятор может быть в одном из двух состояний: заряженным и разряженным.

Таким образом, можно выделить следующие факты, описывающие состояние автомобиля и его узлов:

Группа фактов, описывающая состояние машины:

working-state engine normal – нормальная работа;
working-state engine unsatisfactory – неудовлетворительная работа;
working-state engine does-not-start – не заводится.

Группа фактов, описывающая состояние двигателя:

rotation-state engine rotates – двигатель вращается;
rotation-state engines does-not-rotate – двигатель не вращается.

Группа фактов, описывающая состояние системы зажигания:

spark-state engine normal – зажигание в порядке;
spark-state engine irregular-spark – искра не регулярна;
spark-state engine does-not-spark – искры нет.

Группа фактов, описывающая состояние системы питания:
charge-state battery charged – аккумулятор заряжен;
charge-state battery dead – аккумулятор разряжен.

Обратите внимание, что факты, входящие в одну группу (содержат одинаковое первое поле), являются взаимоисключающими, т.е. наличие в системе сразу двух фактов из одной группы лишено смысла.

Из постановки задачи следует, что наша экспертная система должна предоставлять пользователю рекомендации, позволяющие устранить найденную неисправность. Из приведенных выше правил можно выделить следующие рекомендации: добавить топливо (правило 5); зарядить аккумулятор (правило 6); заменить или почистить контакты (правило 7 или 12); заменить катушку зажигания или распределительные провода (правило 8); прочистить топливную систему (правило 9); отрегулировать зазоры между контактами (правило 10); отрегулировать зажигание (правило 11). Необходимо помнить также о двух крайних случаях: ремонт не требуется в принципе; экспертная система не смогла поставить диагноз.

Таким образом, получатся следующие рекомендации:

– repair "Add gas";
– repair "Charge the battery";
– repair "Replace the points";
– repair "Clean the points";
– repair "Replace the ignition coil";
– repair "Repair the distributor lead wire";
– repair "Clean the fuel line";
– repair "Point gap adjustment";
– repair "Timing adjustment";
– repair "No repair needed";
– repair "Take your car to a mechanic".

Обратите также внимание, что одни и те же рекомендации могут выводиться как правилом 7, так и правилом 12. Однако состояние машины при этой поломке отличается. Для того, чтобы иметь возможность обрабатывать эту ситуацию с помощью одного правила CLIPS, введем еще два дополнительных факта:

symptom engine low-output – низкая мощность;
symptom engine not-low-output – нормальная мощность.

Как упоминалось выше, для работы нашей системы можно заставить пользователя вручную вводить факты, описывающие проявление возникшей неисправности. Однако такой метод имеет ряд серьезных недостатков: пользователь может забыть о каких-нибудь существенных деталях или, наоборот, указать слишком много информации, что может помешать нормальной работе системы. Кроме того, факты, описывающие проявление неисправности, должны иметь строго определенный формат, и система не сможет их обработать в случае ошибки со стороны пользователя.

В нашей экспертной системе мы реализуем правила диагностики, которые в зависимости от той или иной ситуации будут задавать пользователю необходимые вопросы и получать ответ в строго заданной форме. Дальнейшая диагностика будет производиться с учетом предыдущих ответов на вопросы, заданные пользователю. Эти ответы будут формировать описание текущей ситуации с помощью фактов, приведенных выше.

Для реализации подобной архитектуры будет необходимо реализовать функцию, задающую пользователю произвольный вопрос и получающую ответ из заданного набора корректных ответов. Далее приведена одна из возможных реализаций такой функции.

```
(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)
```

Функция принимает два аргумента: простую переменную question, которая содержит текст вопроса, и составную переменную allowed-values с набором допустимых ответов. Сразу после своего вызова функция выводит на экран соответствующий вопрос и читает ответ пользователя в переменную answer. Если переменная answer содержит текст, то она будет принудительно приведена к прописному алфавиту. После этого функция проверяет, является ли полученный ответ одним из заданных корректных ответов. Если нет, то процесс повторится до получения корректного ответа, иначе функция вернет ответ, введенный пользователем.

Будет также очень полезно определить функцию, задающую пользователю вопрос и допускающую ответ в виде да/нет, так как это один из самых распространенных типов вопросов.

```
(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))
```

Функция yes-or-no-p вызывает функцию ask-question с постоянным набором допустимых ответов: yes, no, y и n. В случае, если пользователь ввел ответ yes или y, функция возвращает значение TRUE, иначе FALSE. Обратите внимание, что поскольку функция yes-or-no-p использует функцию ask-question, то она должна быть определена после нее.

Для упрощения реализации нашей экспертной системы введем следующее ограничение: за один запуск система может предоставить пользователю только одну рекомендацию по исправлению неисправности. В случае, если в машине несколько неисправностей, то систему нужно будет последовательно вызывать несколько раз, удаляя обнаруженную на каждом новом шаге неисправность. Таким образом, одним из образцов всех диагностических правил будет (not (repair ?)), гарантирующий, что диагноз еще не поставлен.

Первым реализуем правило, определяющее общее состояние двигателя (см. правило 1).

```
(defrule determine-engine-state ""
(not (working-state engine ?))
(not (repair ?))
=>
(if (yes-or-no-p "Does the engine start (yes/no)? ")
 then
 (if (yes-or-no-p "Does the engine run normally (yes/no)? ")
  then (assert (working-state engine normal))
  else (assert (working-state engine unsatisfactory)))
 else
 (assert (working-state engine does-not-start))))
```

Условный элемент (not (working-state engine ?)) гарантирует, что общее состояние двигателя еще не определено. Если это так, то пользователю задаются соответствующие вопросы и в систему добавляется факт, описывающий текущее общее состояние двигателя.

Теперь реализуем правило, определяющее, пытается ли двигатель вращаться в случае, если он не заводится.

```
(defrule determine-rotation-state ""
 (working-state engine does-not-start)
 (not (rotation-state engine ?))
 (not (repair ?))
 =>
 (if (yes-or-no-p "Does the engine rotate (yes/no)? ")
  then
  (assert (rotation-state engine rotates))
  (assert (spark-state engine irregular-spark))
  else
  (assert (rotation-state engine does-not-rotate))
  (assert (spark-state engine does-not-spark))))
```

Это правило выполняется в случае, если общее состояние двигателя определено и известно, что он не заводится. Кроме того, условный элемент (not (rotation-state engine ?)) гарантирует, что это правило еще не вызывалось. В зависимости от того или иного ответа пользователя правило добавляет соответствующий набор фактов (см. правило 4).

Далее реализуем довольно простые правила 5 и 6. Выполняемые ими действия вы поймете без дополнительных комментариев.

```
(defrule determine-gas-level ""
 (working-state engine does-not-start)
 (rotation-state engine rotates)
 (not (repair ?))
 =>
 (if (not (yes-or-no-p "Does the tank have any gas in it (yes/no)? "))
  then
  (assert (repair "Add gas."))))

(defrule determine-battery-state ""
 (rotation-state engine does-not-rotate)
 (not (charge-state battery ?))
 (not (repair ?))
 =>
 (if (yes-or-no-p "Is the battery charged (yes/no)? ")
  then
  (assert (charge-state battery charged))
  else
  (assert (repair "Charge the battery."))
  (assert (charge-state battery dead))))
```

Обратите внимание, что правило determine-battery-state, помимо определения возможной неисправности, также применяется для добавления в систему факта, описывающего текущее состояние аккумулятора, который может быть использован другими правилами.

При реализации правила 7 необходимо обратить внимание на то, что рекомендации, предоставляемые этим правилом, подходят для двух в корне отличающихся ситуаций. Во-первых, в случае, если двигатель не заводится и существует вероятность плохой искры в системе зажигания (правило 7). Во-вторых, в случае, если двигатель запускается, но не развивает нормальной мощности (правило 12). Поэтому выполним реализацию этих правил так, как представлено ниже.

```
(defrule determine-low-output ""
 (working-state engine unsatisfactory)
 (not (symptom engine low-output | not-low-output))
 (not (repair ?))
 =>
 (if (yes-or-no-p "Is the output of the engine low (yes/no)? ")
  then
  (assert (symptom engine low-output))
  else
  (assert (symptom engine not-low-output))))

(defrule determine-point-surface-state ""
(or (and (working-state engine does-not-start)
  (spark-state engine irregular-spark))
 (symptom engine low-output))
(not (repair ?))
=>
(bind ?response
 (ask-question "What is the surface state of the points (normal/burned/contaminated)? "
    normal burned contaminated))
(if (eq ?response burned)
 then
 (assert (repair "Replace the points."))
 else (if (eq ?response contaminated)
   then (assert (repair "Clean the points.")))))
```

Правило determine-low-output определяет, имеет ли место низкая мощность двигателя или нет. Правило determine-point-surface-state адекватно реагирует на условия, заданные в правилах 7 и 12. Обратите внимание на использование условных элементов or и end, которые обеспечивают одинаковое поведение правила в двух абсолютно разных ситуациях. Кроме того, правило determine-point-surface-state отличается от приведенных ранее тем, что непосредственно использует функцию ask-question вместо yes-or-no-p, так как в данный момент пользователю задается вопрос, подразумевающий три варианта ответа.

Реализация оставшихся диагностических правил (8 – 11) также не должна вызвать у вас затруднений.

```
(defrule determine-conductivity-test ""
 (working-state engine does-not-start)
```

```
 (spark-state engine does-not-spark)
 (charge-state battery charged)
 (not (repair ?))
 =>
 (if (yes-or-no-p "Is the conductivity test for the ignition coil positive (yes/no)? ")
  then
  (assert (repair "Repair the distributor lead wire."))
  else
  (assert (repair "Replace the ignition coil."))))
(defrule determine-sluggishness ""
 (working-state engine unsatisfactory)
 (not (repair ?))
 =>
 (if (yes-or-no-p "Is the engine sluggish (yes/no)? ")
  then (assert (repair "Clean the fuel line."))))
 (defrule determine-misfiring ""
 (working-state engine unsatisfactory)
 (not (repair ?))
 =>
 (if (yes-or-no-p "Does the engine misfire (yes/no)? ")
  then
  (assert (repair "Point gap adjustment."))
  (assert (spark-state engine irregular-spark))))
 (defrule determine-knocking ""
 (working-state engine unsatisfactory)
 (not (repair ?))
 =>
 (if (yes-or-no-p "Does the engine knock (yes/no)? ")
  then
  (assert (repair "Timing adjustment."))))
```

Внимательно взглянув на список правил, мы увидим, что некоторые правила (2, 3 и 13) остались до сих пор нереализованными.

Реализация правила 13 будет следующая.

```
(defrule no-repairs ""
 (declare (salience –10))
 (not (repair ?))
 =>
 (assert (repair "Take your car to a mechanic.")))
```

Обратите внимание на использование приоритета при определении этого правила. Все правила, приведенные в предыдущем разделе, определялись с приоритетом, по умолчанию равным нулю. Использование для правила no-repairs приоритета, равного –10, гарантирует, что правило не будет выполнено, пока в плане решения задачи находится, по крайней мере, одно из диагностических правил. Если все активированные диагностические правила опрошены и ни одно из них не смогло подобрать подходящую рекомендацию по устранению неисправности, то система запустит данное правило, рекомендующее пользователю обратиться к механику.

Реализация правил 2 и 3 приведена ниже.

```
(defrule normal-engine-state-conclusions ""
 (declare (salience 10))
 (working-state engine normal)
 =>
 (assert (repair "No repair needed."))
 (assert (spark-state engine normal))
 (assert (charge-state battery charged))
 (assert (rotation-state engine rotates)))
(defrule unsatisfactory-engine-state-conclusions ""
 (declare (salience 10))
 (working-state engine unsatisfactory)
 =>
 (assert (charge-state battery charged))
 (assert (rotation-state engine rotates)))
```

В этих правилах, наоборот, используется более высокий приоритет, что гарантирует их выполнение до запуска любого диагностирующего правила (в случае выполнения соответствующих условий). Это избавит нашу систему от лишних проверок, а пользователя от лишних вопросов.

Экспертная система фактически готова к работе. Единственное, чего ей не хватает - это метода вывода итоговой информации и правила, сообщающего пользователю о начале работы системы. Ниже приведена реализация этих правил.

```
(defrule system-banner ""
 (declare (salience 10))
 =>
 (printout t crlf crlf)
 (printout t "The Engine Diagnosis Expert System")
 (printout t crlf crlf))

(defrule print-repair ""
 (declare (salience 10))
 (repair ?item)
 =>
 (printout t crlf crlf)
 (printout t "Suggested Repair:")
 (printout t crlf crlf)
 (format t " %s%n%n%n" ?item))
```

Теперь для того, чтобы запустить экспертную систему, достаточно выполнить команду reset, которая добавит факт initial-fact, необходимый для правила system-banner, и команду run. После этого вы сразу увидите сообщение "The Engine Diagnosis Expert System", которое означает, что система начала работать, и получите серию вопросов, ответы на которые помогут экспертной системе оценить состояние вашей машины и подобрать соответствующую рекомендацию по ремонту.

СПИСОК ЛИТЕРАТУРЫ

1. http://www.ghg.net/clips/CLIPS.html
2. Гаскаров, Д.В. Интеллектуальные информационные системы / Д.В. Гаскаров. - М. : Высшая школа, 2003. - 431 с.
3. Частиков, А.П. Разработка экспертных систем. Среда CLIPS / А.П. Частиков, Т.А. Гаврилова, Д.Л. Белов. - СПб. : БХВ-Петербург, 2003. - 608 с.