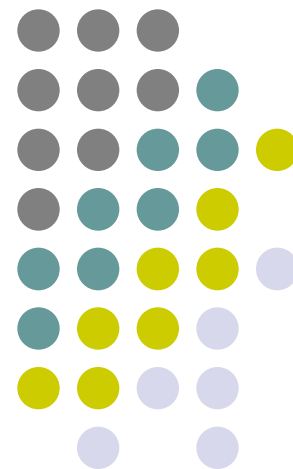
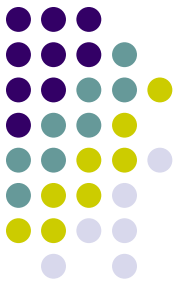


ПРОДУКЦИОННАЯ МОДЕЛЬ:

ЯЗЫК CLIPS

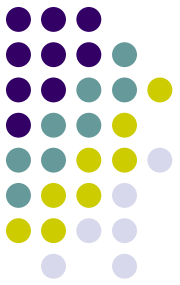


СОДЕРЖАНИЕ



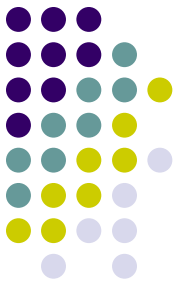
1. Общая характеристика CLIPS
 - особенности управляющей стратегии
2. Факты рабочей памяти
 - виды фактов, шаблоны
3. Создание правил продукций
 - левая часть: образцы и условные элементы
 - правая часть: операции с РП
4. Средства работы с системой
 - загрузка/выгрузка РП
 - работа с правилами
 - определение функций
5. Практическое задание: экспертные системы
 - этапы разработки ЭС
 - пример ЭС: сущности, правила, интерфейс

ИСТОРИЯ СОЗДАНИЯ CLIPS



- Появление языка CLIPS – 1984 г., NASA (*C Language Integrated Production System*)
- Особое внимание – совместимость с ранее созданными языками ПЗ для ЭС, поэтому синтаксис лисповский
- Первоначально: только продукционная парадигма
- 1991 г. – введены 2 новые парадигмы: процедурное + объектно-ориентированное программирование =>
CLIPS Object-Oriented Language (COOL)
- Свободно распространяемый программный продукт, для компиляции может быть использован любой ANSI C или C++ компилятор

CLIPS: ОСОБЕННОСТИ УПРАВЛЯЮЩЕЙ СТРАТЕГИИ



- CLIPS – стратегия прямого безвозвратного вывода
- Две области памяти: РП и БЗ, для каждой – свой набор функций и *конструкторов* (= процедур)
- Запуск программы: загрузка БЗ и фактов в РП, затем запуск на выполнение
- Выбор и задание стратегии разрешения конфликта (7 возможных стратегий)
- Повторение цикла *сопоставление – разрешение конфликта – выполнение правила* до тех пор, пока есть применимые правила (но также возможно выполнение заданного пользователем количества правил)
- Правила выполняются в порядке уменьшения приоритета, *приоритет правила* задается при определении правила

СТРАТЕГИИ РАЗРЕШЕНИЯ КОНФЛИКТА

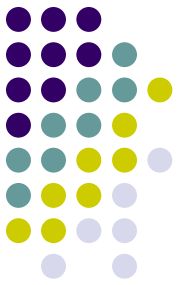


CLIPS: 7 стратегий разрешения конфликта

- Стратегия глубины (приоритет + только что активированное правило ставится выше)
- Стратегия ширины (приоритет + только что активированное правило ставится ниже)
- Стратегия упрощения (приоритет + наименьшая определённость)
- Стратегия усложнения (приоритет + наибольшая определённость)
- Стратегия LEX (приоритет + определённость + новизна)
- Стратегия MEA (приоритет + новизна)
- Стратегия случайного выбора

Определённость – число сопоставлений в образце правила
Новизна – временной тег

CLIPS: ТИПЫ ДАННЫХ



Простые типы данных:

- Числовые: `float`, `integer` `-32.3e-7` `+12.5` `245`
- `symbol` – для именования объектов: `2my` `127a` `@=%`
любая последовательность неуправл. символов ASCII
до первого ограничителя, не могут начинаться с `?` или `$?`
- `string`: `"x and y"` `"x\"z"` (строка `x"z`)

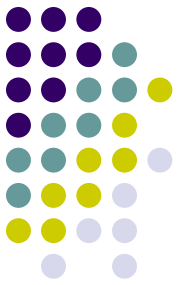
Составная конструкция – последовательность данных
простых типов, записанная в круглых скобках

(`z` `27.4` `"Bob"` `23`)

Ограничители: пробел, табуляция, переход на
новую строку, символы `"` `(` `)` `&` `|` `<` `~` `;`

`;` – символ начала комментария (ограничен концом строки)

ФАКТЫ В РАБОЧЕЙ ПАМЯТИ



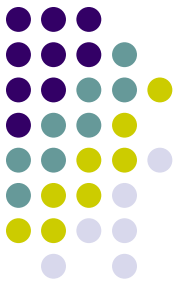
Рабочая память (РП) – набор фактов

В процессе работы ПС: добавление/удаление фактов

Виды фактов:

- *Факты с упорядоченными атрибутами (ordered facts)*
- *Факты с неупорядоченными именованными атрибутами (unordered facts)*
- Факты с неупорядоченными именованными атрибутами – аналог записей или структур, их можно считать упрощёнными фреймами
- *Шаблон факта* – описание структуры такого факта, аналог фрейма-прототипа
- Сам факт с неупорядоченными именованными атрибутами – аналог фрейма-экземпляра

ФАКТЫ С УПОРЯДОЧЕННЫМИ АТТРИБУТАМИ



- Факт – составная конструкция,
первый элемент – типа symbol
(parent Bob Liz) (Mike age 27 tel 5552233)
- Первым элементом не м.б. зарезервированные слова:
test, and, or, not, declare, logical, object, exists, forall
- Важен порядок записи элементов конструкции
- Каждый такой факт может описывать один или несколько атрибутов некоторой сущности вместе с их значениями

; Текущее состояние холодильника

(refrigerator light on) ; свет включен

(refrigerator door open) ; дверь открыта

(refrigerator temp (+ 5 10 15)); t° 30 градусов

ФАКТЫ С НЕУПОРЯДОЧЕННЫМИ ИМЕНОВАННЫМИ АТТРИБУТАМИ



- Атрибуты сущностей именуются:

```
(point (x-coord 15) (y-coord -25))
```

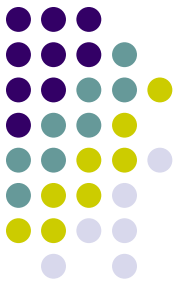

; бакалейные товары: число их и их список

```
(grocery-list (#-of-items 3)  
              (items bread milk eggs))
```
- Порядок атрибутов (слотов) не важен, доступ к значению – через имя слота
- Факты определяются конструктором *deftemplate*:

```
(deftemplate grossery-list  
  (slot #-of-items) ; количество товаров  
  (multislot items) ) ; список товаров
```

 - `slot` – для определения атрибута (слота) простого типа
 - `multislot` – для атрибута с несколькими значениями
 - количество слотов произвольное

ШАБЛОНЫ ФАКТОВ



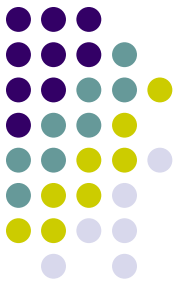
При определении шаблона фактов можно задать:

- ограничение на тип значения слота;
для мультислота – ограничение на тип каждого элемента

- значение слота по умолчанию: *default*

```
(deftemplate grossery-list
  (slot #-of-items (type INTEGER)
           (default 2))
  (multislot items (type SYMBOL)))
```

- ❖ Значение по умолчанию может быть задано как
 - `default` – статическое, вычисляется при создании шаблона
 - `default-dynamic` – динамическое, вычисляется для любого факта-экземпляра при его создании
- ❖ Значение по умолчанию может быть задано явно (в виде константы) или неявно (в виде вызова функции)



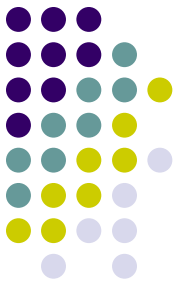
ШАБЛОНЫ: ЗНАЧЕНИЯ ПО УМОЛЧАНИЮ

```
(deftemplate my_example
  (slot z (default (my_function)))
  (slot w (default-dynamic
              (my_function)))
  (slot x (default ?NONE))
  (slot y (default ?DERIVE)))
```

Если вместо значения по умолчанию – ключевое слово

- `?NONE` , то значение слота/атрибута должно быть обязательно задано при создании факта-экземпляра
- `?DERIVE` , то значение слота/атрибута может извлекаться из ограничений (для всех слотов по умолчанию)

ПРАВИЛА CLIPS: СОЗДАНИЕ



Для создания продукции – конструктор *defrule*

```
(defrule <имя правила>      [ "комментарии" ]  
  [ (declare (salience <целочисл. выражение>)) ]  
  <условие применения>      ; левая часть  
  =>  
  <действия>                ) ; правая часть
```

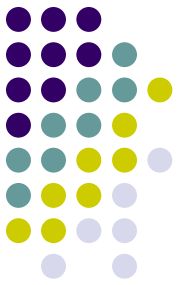
- Комментарии могут отсутствовать
- *Левая часть* – условия применения правила, состоит из *условных элементов*, чаще всего – *образцов фактов*
- *Правая часть* обычно содержит действия, добавляющие или удаляющие факты из РП
- Свойство `salience` устанавливает приоритет правила (целое в диапазоне от -10000 до +10000), по умолчанию приоритет устанавливается равным нулю

ЛЕВЫЕ ЧАСТИ ПРАВИЛ



- Левая часть правила (условие) состоит из *условных элементов*
- Для выполнения правила должны выполняться все условные элементы, т.е. все условные элементы неявно конъюнктивно объединены
- Типичный условный элемент – *образец факта*
- Образец факта позволяет задать общий вид факта, в нём могут использоваться константы и переменные (начинаются с префикса ? или \$?): 2b \$?h ?fg
- Переменная может быть именованной (для использования несколько раз) или *анонимной*: ? \$?
(используется только для сопоставления)
- Переменная может сопоставляться с одним объектом (?) или с последовательностью объектов (\$?)

ПРИМЕР ОБРАЗЦА ПРАВИЛА

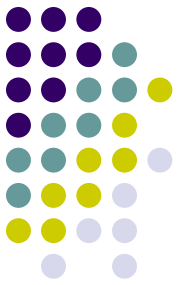


```
(defrule Find-data (data ? blue 56 $?)  
=> ...)
```

Поиск факта `data` с упорядоченными атрибутами по образцу, в котором:

- первый атрибут не важен (? – анонимная переменная, сопоставляющаяся с одним значением)
- второй атрибут – значение `blue`
- третий атрибут – значение `56`
- может быть ещё произвольное количество атрибутов (они сопоставятся с анонимной переменной `$?`)

ОБРАЗЦЫ ФАКТОВ: ЭЛЕМЕНТЫ



Образец факта может включать:

- константы – значения простых типов
- переменные – все вхождения одной переменной сопоставляются с одним и тем же значением
- логические операции (конъюнкция: $\&$, дизъюнкция: $|$, отрицание: $\sim$)
- предикаты – их вызов начинается с символа $:$
- функции – их вызов начинается с символа $=$

Пустой образец автоматически заменяется на условие-образец `initial-fact`, т.е.

```
(defrule empty_cond => ...)
```

заменится на

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

ПРИМЕР ПРАВИЛА С ОБРАЗЦАМИ



```
(defrule Find-2-Coeval-Person
  (person (name ?x) (age ?z) )
  (person (name ?y&~?x) (age ?z) )
=>
  (printout t ?x " " ?y " " ?z crlf))
```

- Среди фактов `person` с именованными атрибутами поиск пары фактов, описывающих людей-ровесников
- Для каждой найденной пары фактов печатаются имена этих людей и их возраст
- Чтобы имена людей в паре не совпадали, используются логические связки конъюнкция и отрицание

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ ПРЕДИКАТОВ И ФУНКЦИЙ В ОБРАЗЦАХ



Пример1: `(defrule Find-data1
 (mydata1 ?x ?y&= (* 2 ?x)) => ...)`

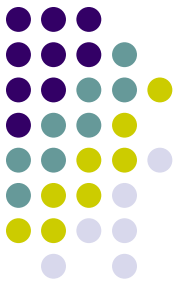
поиск факта `mydata1` с упорядоч.и атрибутами, у которого значение второго атрибута в два раза больше первого

Пример2: `(defrule Find-data2
 (mydata2 ?x&:(floatp ?x) &:(> ?x 0)
 $?y ?z&:(stringp ?z)) => ...)`

поиск факта `mydata2` с упорядоченными атрибутами

- первый атрибут имеет тип *float* и значение, большее нуля
- последний атрибут имеет тип *string*
- количество атрибутов – произвольно, но не меньше двух

УСЛОВНЫЕ ЭЛЕМЕНТЫ В ЛЕВОЙ ЧАСТИ ПРАВИЛ

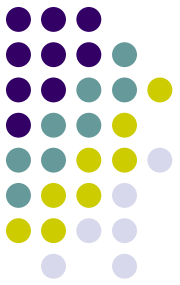


- Условные элементы:
test, and, or, not, logical, exists, forall
- Элемент `test` содержит выражение, возвращающее логическое значение
- Пример:

```
(defrule example  
  (data ?x)      (data ?y)  
  (test (>= (abs (- ?x ?y)) 5)) => ...)
```

поиск в РП двух фактов `data` с атрибутом-числом, абсолютная разница между значениями атрибутов которых не меньше пяти

УСЛОВНЫЕ ЭЛЕМЕНТЫ: AND, OR И NOT



- ❖ Условные элементы `not`, `or` и `and` позволяют задавать наборы фактов (правило применимо на любом подходящем наборе)

Пример:

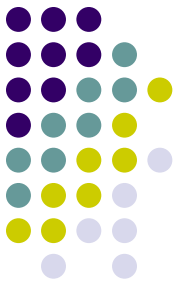
```
(defrule exam1
  (myfact1 ?z)
  (or (and (myfact2 ~?z) (myfact3 ?))
    (not (myfact4 ?z))) => ...)
```

правило применимо в случае, если в РП

- есть факт `myfact1` с некоторым значением атрибута
- и либо одновременно есть факт `myfact2` с другим значением этого атрибута и факт `myfact3` либо в РП нет факта `myfact4` с таким же значением атрибута

УСЛОВНЫЕ ЭЛЕМЕНТЫ

EXISTS И FORALL



Задание условий на наборы фактов

- Элемент `exists` – проверка, если есть хотя бы один набор фактов, удовлетворяющих условию:

```
(defrule ex-for-exists
  (exists (a ?x) (b ?x) (c ?x)) => ...)
```

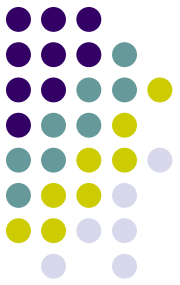
проверка наличия 3 фактов с именами a, b и c,
содержащих одно и то же значение

- Элемент `forall` – заданное условие должно выполняться для всех включенных в него образцов

```
(defrule all-student-passed
  (forall (student ?name) (reading ?name)
  (writing ?name) (arithmetic ?name)) => ...)
```

проверка, что все студенты прошли чтение, письмо и
арифметику или же в РП нет данных о студентах

УСЛОВНЫЙ ЭЛЕМЕНТ *LOGICAL*

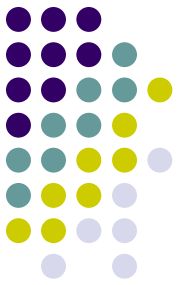


- Позволяет задавать логическую зависимость данных правой части правила от данных левой
- Применяется только к фактам с упорядоченными атрибутами
- Может быть применён только к первым n условным элементам

```
(defrule examp-for-logical
  (logical (A) )
  (logical (B) )
  (C)    => (assert (D) ) )
```

Факт D добавляется в РП и логически зависит от фактов A и B, поэтому будет автоматически удалён при удалении любого из этих фактов (но не при удалении C).

ПРАВАЯ ЧАСТЬ ПРАВИЛ



Правая часть может включать действия:

- ввод/вывод: *printout, read ...*
- изменение рабочей памяти – добавление, удаление, модификация фактов:
assert, retract, modify ...
используются номера/адреса фактов
- изменение БЗ – добавление, удаление правил: *defrule, undefrule ...*
- вызов вспомогательных функций для обработки данных
- работа с объектами (для COOL, например, посылка сообщений и т.д.)

РП: НОМЕРА И АДРЕСА ФАКТОВ



- ❖ Каждый факт после добавления в РП имеет номер вида $f - \langle \text{число} \rangle$
- ❖ Каждый новый факт при добавлении получает очередной номер
- ❖ В результате модификации факта его номер меняется
- ❖ Нумерация фактов начинается с нуля, при загрузке системы в РП заносится факт с номером нуль – `initial-fact`
- ❖ На факт можно сослаться по его номеру (число без префикса f) или его адресу, получаемому с помощью специальной функции $\leftarrow$
- ❖ Такие ссылки на факты используются в функциях для работы с фактами (удаление и модификация)

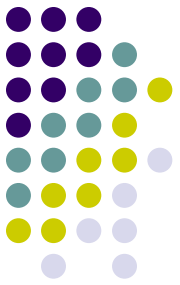
ПРИМЕР: ИСПОЛЬЗОВАНИЕ АДРЕСОВ



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

- Правило удаляет все факты с именем `data`
- Переменная `?data-facts`, связанная с адресом факта, может сравниваться с другой переменной или использоваться внешней функцией
- В данном примере она используется функцией `retract`

ДОБАВЛЕНИЕ И ИЗМЕНЕНИЕ ФАКТОВ



- `assert` – добавление фактов в РП:
`(assert <факт>{<факт>})`
 - `retract` – удаление фактов:
`(retract <номер или адрес факта>
{< номер или адрес факта>} | *)`
 - `modify` – изменение факта, эквивалентно применению пары `retract-assert` :
`(modify <номер или адрес факта>
<новое значение слота>{<новое значение слота>})`
 - `duplicate` – создание нового факта, копируется информация из уже существующего факта
`(duplicate <номер или адрес факта>
<новое значение слота>{<новое значение слота>})`
- `modify` и `duplicate`: только для фактов с имен. атрибутами

РАБОТА С СИСТЕМОЙ



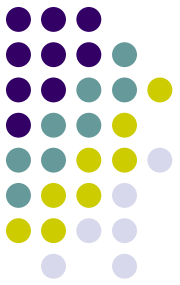
Программа на CLIPS может содержаться
в двух текстовых файлах:

- файл с начальным содержимым РП (т.е. с фактами)
- файл с правилами, шаблонами, функциями ...

Типичные действия

- выполняемые пользователем в среде CLIPS и/или
- встречающиеся в правой части правил продукции:
- Загрузка в систему шаблонов, фактов и правил
- Перед загрузкой фактов с неупорядоченными именованными атрибутами должны быть загружены соответствующие шаблоны
- После загрузки РП и БЗ системой составляется список применимых в данный момент правил
- Запуск программы на выполнение - команда *run*: (`run`)

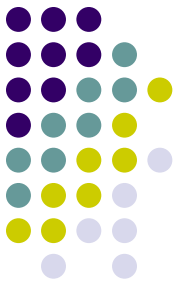
ЗАГРУЗКА И ОЧИСТКА РП



- Изначально рабочая память пуста, в неё автоматически добавляется факт `initial-fact`.
- Добавление списка фактов в рабочую память (пример):

```
(deffacts startup "Refrigerator status"  
  (refrigerator light on)  
  (refrigerator door open)  
  (refrigerator temp (+ 5 10 15)))
```
- Факты, описанные в конструкторах `deffacts`, автоматически добавляются после выполнения команды `(reset)`
- Команда `(clear)` – очистка РП, она становится пустой
- Команда `(reset)` – очистка РП, после выполнения автоматически добавляются все факты из конструкций `deffacts` и факт `initial-fact`

СОХРАНЕНИЕ ФАКТОВ И ЗАГРУЗКА В РП



- Сохранение всех/некоторых фактов из РП в текстовый файл:

```
(save-facts <имя файла>
```

```
[<границы видимости> <список имён шаблонов>])
```

```
< границы видимости > ::= visible | local
```

- Загрузка фактов из файла:

```
(load-facts <имя файла>)
```

Факты с именованными атрибутами могут быть загружены из файла только после загрузки в систему соответствующего определения шаблона

- Загрузка конструкторов из файла:

```
(load "myfile1.CLP")
```

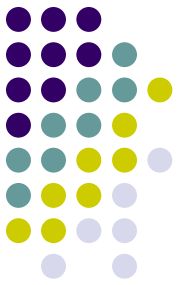
аргумент – имя текстового файла, может быть указан путь к нему, по умолчанию он ищется в текущей директории

СРЕДСТВА РАБОТЫ С ПРАВИЛАМИ



- Загрузка файла с правилами и конструкторами шаблонов: `(load "имя файла")`
- Просмотр определения правила:
`(ppdefrule <имя правила>)`
- Получение полного списка правил, присутствующих в CLIPS на данный момент:
`(list-defrules <имя модуля>)`
аргумент не обязателен (по умолчанию – текущий модуль,
* – из всех модулей)
- Удаление правил: `(undefrule <имя правила>)`
или `(undefrule *)`
- Сохранение в текстовый файл всех определённых в системе в данный момент конструкторов (правил, шаблонов, функций): `(save "имя файла ")`

ОПРЕДЕЛЕНИЕ ФУНКЦИЙ



В правой части правил можно применять функции:

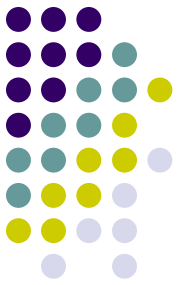
- определённые пользователем, внешние
- системные, внутренние
- определённые в CLIPS с помощью конструктора *deffunction* (их отличие от внешних – они выполняются интерпретатором)

Примеры:

```
(deffunction print-args (?x ?y $?z)
  (printout t ?x " " ?y " and " (length ?z)
    " extras: " ?z crlf))

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

ЗАПУСК И ОТЛАДКА



Запуск программы:

(run <целочисленное выражение>)

- Если аргумент задан и >0 , то выполняется соответствующее количество правил из списка применимых правил
- Аргумент может быть опущен или <0 , тогда весь список выполнится полностью
- В процессе выполнения список применимых правил может меняться
- ❖ В любой момент возможно посмотреть содержимое РП: (facts) – вывод на экран текущего содержимого РП
- ❖ Можно посмотреть список применимых в данный момент правил с помощью команды (agenda)

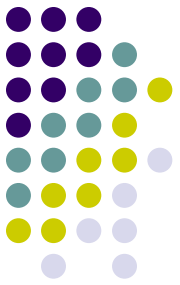
ПРАКТИЧЕСКОЕ ЗАДАНИЕ № 3

(обязательно для допуска к экзамену)



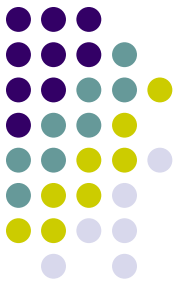
- Разработка ЭС на базе языка CLIPS (прямой вывод)
- Работа индивидуально или в группах по 2-3 человека
- Разделение работы обсуждается с преподавателем:
 - ☐ анализ ПО и выявление экспертных правил
 - ☐ программирование правил (создание БЗ)
 - ☐ программирование вывода в ЭС
 - ☐ отладка БЗ и тестирование ЭС
- Выбор ПО и типа ЭС – свободный, но оговаривается с преподавателем. Например: диагностика неисправностей ноутбука, рекомендации по покупке холодильника
- Обязательная сдача **отчета по заданию** (1-2 стр.)
- Задание сдается по этапам **Н.В.Груздевой**
- Можно использовать готовую **оболочку ЭС** (или некоторые средства из нее), но это соответственно снижает оценку
- Контрольный срок сдачи – **12 мая**, выбор ПО – **21 апреля**

ЭТАПЫ РАЗРАБОТКИ ЭС



1. Выявление сущностей и их атрибутов
2. Выявление экспертных правил
3. Представление сущностей и их атрибутов в виде фактов
4. Определение схемы вывода решения
5. Реализация экспертных правил правилами *CLIPS*
6. Программирование вспомогательных функций для запроса информации у пользователя (интерфейс с пользователем)
7. Программирование схемы вывода и всей ЭС
8. Тестирование и отладка БЗ

РАЗРАБОТКА ЭС: ПРИМЕР

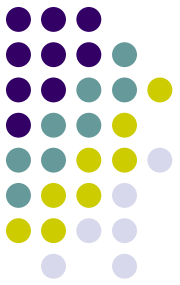


Разработка ЭС диагностики неисправности мотора автомобиля по внешним признакам

Экспертные правила:

- Двигатель обычно находится в одном из трёх состояний: работает нормально, работает неудовлетворительно, не заводится.
- Двигатель работает нормально, если он нормально вращается, система зажигания и аккумулятор находятся в норме, тогда никакого ремонта не требуется.
- Если двигатель запускается, но работает ненормально, то это говорит, по крайней мере о том, что аккумулятор в порядке.
- Если двигатель не запускается, то нужно узнать, пытается ли он вращаться. Если он вращается, но 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.
- ...

ПРИМЕР ЭС: СУЩНОСТИ

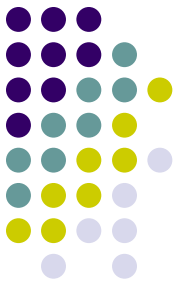


- Состояние двигателя: нормальная работа, неудовлетворительная работа, не заводится
- Состояние вращения двигателя: способен вращаться или нет
- Состояние аккумулятора: заряжен или разряжен
- и т.д.

Каждому состоянию можно поставить в соответствие факты вида:
имя сущности – атрибут – значение:

- ; Группа фактов, описывающих состояние двигателя*
(working-state engine normal)
(working-state engine unsatisfactory)
(working-state engine does-not-start)
- ; Группа фактов: состояние вращения двигателя*
(rotation-state engine rotates)
(rotation-state engine does-not-rotate)
- ; Состояние аккумулятора*
(charge-state battery charged) ; заряжен
(charge-state battery dead) ; разряжен

ПРИМЕР ЭС: ПРАВИЛА



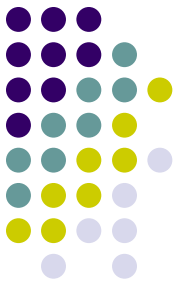
Первому экспертному правилу соответствует:

```
(defrule determine-engine-state
  (not (working-state engine ?))
  (not (repair ?)) ; в РП нет факта-рекомендации
  =>
  (if (yes-or-not-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))))
```

В результате выполнения правила, в зависимости от ответов пользователя в РП заносится один из трёх фактов о состоянии двигателя

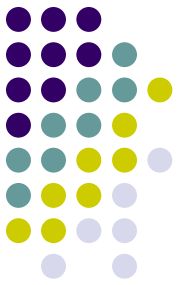
Правило применимо только если в РП нет фактов о состоянии двигателя и факта `repair`

ИНТЕРФЕЙС С ПОЛЬЗОВАТЕЛЕМ



```
; ЭС задаёт вопрос и вводит ответ, который д.быть  
; одним из заданных возможных вариантов ответа  
(deffunction ask-question (?question $?allowed-values)  
  (printout t ?question) (bind ?answer (read))  
  (if (lexemep ?answer)  
      then (bind ?answer (lowercase ?answer)))  
  (while (not (member ?answer ?allowed-value)) do  
    (printout t ?question) (bind ?answer (read))  
    (if (lexemep ?answer)  
        then (bind ?answer (lowercase ?answer)))))  
  ?answer )  
  
; ЭС задаёт вопрос, на который ответ - да или нет  
(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)    )
```

ПРИМЕР ЭС: ЗАВЕРШАЮЩЕЕ ПРАВИЛО



В случае, когда в ходе опроса не удалось выработать рекомендацию, применяется следующее правило:

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

В РП заносится факт, рекомендуемый пользователю показать машину механику

Правило применимо только в том случае, если больше нет никаких применимых правил (т.к. у него приоритет -10, приоритет остальных правил = 0 по умолчанию), и не была выработана рекомендация пользователю



СПАСИБО ЗА ВНИМАНИЕ!