СПбГЭТУ «ЛЭТИ»
Кафедра Вычислительной техники
Дисциплина «Искусственный интеллект»

Лекция 7

CLIPS - среда разработки ЭС

Что такое CLIPS

CLIPS (*C Language Integrated Production System*) - программная среда для разработки экспертных систем

- разработана в начале 1980-х годов в Космическом центре Джонсона NASA

Литература:

1. http://www.clipsrules.net/ – сайт проекта

2. *Частиков А. П., Гаврилова Т. А., Белов Д. Л.* Разработка экспертных систем. Среда CLIPS. – СПб.: БХВ-Петербург, 2003. 608 с.

3. *Джарратано Дж., Райли Г.* Экспертные системы: принципы разработки и программирование». – М.: Вильямс, 2007.

4. Пантелеев М.Г., Родионов С.В. Модели и средства построения экспертных систем: Учеб. пособие. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2003.

Общая характеристика среды CLIPS

- Среда CLIPS поддерживает **три** основных **способа представления знаний**:
  - *продукционные правила* для представления эвристических знаний;
  - *функции* для представления процедурных знаний;
  - *объектно-ориентированное* программирование.
- Поддерживаются **5 основных черт ООП**: *классы*, *обработчики сообщений*, *абстракции*, *инкапсуляция*, *наследование* и *полиморфизм*
- Приложения могут разрабатываться с использованием *только **правил***, *только **объектов*** (в этом случае машина вывода не используется) или их ***комбинации***:
  - Начиная с версии **6.0** правила могут сопоставляться не только с фактами, но и с объектами
- Поддерживается **интеграция с другими средствами**:
  - CLIPS может *вызываться из процедурных языков*, выполнять свои функции и затем возвращать управление вызвавшей программе.
  - *процедурный код* может быть определен как внешняя функция и *вызван из CLIPS*

Базовые типы данных в CLIPS. Числовые данные

- В CLIPS поддерживаются восемь базовых типов данных:
  - *целые* числа (integer);
  - *вещественные* числа (float);
  - *символьные* данные (symbol);
  - *строковые* данные (string);
  - *внешний адрес* (external-address);
  - *адрес факта* (fact-address);
  - *имя экземпляра* (instance-name);
  - *адрес экземпляра* (instance-address);

Базовые типы данных в CLIPS: Числовые данные

- *Целые числа* состоят из знака (необязательного для положительных чисел) и последовательности десятичных цифр
  - Например: 27; +125; -38
- *Вещественные* числа содержат *мантиссу* (дробная часть отделяется точкой) и *необязательный порядок*, состоящий из символа "e" и целого числа
  - Например: 12.0; -1.59; 237e3; -32.3e-7

Базовые типы данных. Символьные и строковые значения

- *Символьное* значение – последовательность *отображаемых* ASCII-символов, продолжающаяся *до ограничителя*
  - *Ограничители*: любой *неотображаемый* ASCII-символ (пробел, табуляция, возврат каретки, перевод строки), *кавычка*, *открывающая* и *закрывающая скобки*, *амперсанд* (&), *вертикальная черта* (|), *знак «меньше»* (<) *тильда* (~). Точка с запятой (;) – ограничитель, используемый для указания на начало комментария. Ограничители в символьное значение *не включаются* !
  - *Не может начинаться* с символа "?" или пары символов "$?", (но может содержать их внутри себя).
  - CLIPS чувствителен к регистру (abc и ABC – два разных значения)
  - Примеры: bad_value; 456-93-039; @+=%
- *Строковое* значение – *заключенная в кавычки* последовательность (возможно, пустая) отображаемых символов
  - для включения в строковое значение кавычек перед ними необходимо поставить символ "\". Для включения в строковое значение символа "\", перед ним необходимо поместить еще один символ "\"
  - Примеры: "abc"; "a & b"; "a\"quote"; "fgs\\85"

Базовые типы данных: адреса

- *Внешний адрес* – адрес внешней структуры данных, ***возвращаемый*** интегрированной в CLIPS *функцией* (написанной на языках C или Ada).
  - создается *только как результат вызова функции – невозможно специфицировать вводом значения*,
  - отображаемое представление внешнего адреса:

    <Pointer-XXXXXX>, где XXXXXX – внешний адрес.

- *Адрес факта* – используется для ссылки на факты
  - Факт – список атомарных значений, на которые можно ссылаться либо позиционно (в упорядоченных фактах), либо по имени (в неупорядоченных фактах).
  - Отображаемый формат адреса факта:

    <Fact-XXX>, где XXX – индекс факта

Базовые типы данных: имя и адрес экземпляра

- *Имя экземпляра* – используется для *ссылки на экземпляры классов*
  - *Экземпляр – объект*, являющийся представителем некоторого *класса*. Объектами в CLIPS по определению являются *целые* и *вещественные числа*, *символьные* и *строковые* значения, *многоместные* значения, *внешние адреса*, *адреса фактов* или *экземпляры определенных пользователем классов* (создаваемых с помощью конструкции **`defclass`**);
  - *Имя экземпляра* представляется **символьным типом, заключенным в квадратные скобки** (скобки не являются частью имени, а только указывают тип значения);
  - Например: [pump-1]; [foo]; [+++]; [123-890];
- *Адрес экземпляра* – может быть *получен путем связывания значения, возвращаемого функцией* **`instance-address`**, или связывания переменной с экземпляром, сопоставляемым с объектным образцом в левой части правила
  - адрес экземпляра невозможно специфицировать вводом значения!
  - отображаемое представление адреса экземпляра в CLIPS:

    <Instance-XXX>, где XXX – имя экземпляра.
- На экземпляры определяемых пользователем классов *можно ссылаться по имени либо по адресу*. Адреса экземпляров должны использоваться, когда *критично время решения*.

Поля, многоместные значения и факты

- *Поле* (field) – место, занимаемое одним значением базового типа данных
  - все значения базовых типов являются одноместными (single-field value);
- *Многоместное значение* (multifield value) – последовательность из нуля или более одноместных значений.
  - отображаются в скобках, где одноместные значения разделяются пробелами.
  - примеры многоместных значений: (a 123); (); (x 3.0 "red" 567)
- Факт – одна из основных форм представления информации в CLIPS-системах
  - *используются правилами* для вывода новых фактов из имеющихся
  - все текущие факты в CLIPS помещаются в список фактов (**fact-list**)
- По формату представления в CLIPS выделяют *два типа фактов*: *упорядоченные* и *неупорядоченные*.

Упорядоченные факты

- *Упорядоченный факт* – состоит из заключенной в скобки последовательности одного или более разделенных пробелами полей
  - *первое поле должно быть символьного типа*, т.к. оно *специфицирует отношение*, которое применяется к остальным полям факта
  - остальные поля могут быть любыми базовыми типами данных
  - Примеры:
    (высота 100);
    (студент Сидоров);
    (отец Иван Петр);
    (однокурсники Иванов Петров Сидоров)

- *Индекс факта* – *уникальный целочисленный индекс*, приписываемый факту при его добавлении или модификации.
  - индексация фактов *начинается с нуля* и инкрементируется при каждом новом или измененном факте

- *Идентификатор факта* (fact-identifier) – краткая нотация, используемая для отображения факта
  - состоит из символа “f”, тире и *индекса факта*.
  - Например, f-10 ссылается на факт с индексом 10.

Команды для работы с фактами

- Для работы с фактами используются следующие команды:

  `assert` – добавляет факт в факт-список;

  `retract` – удаляет факт из списка;

  `modify` – модифицирует список;

  `duplicate` – дублирует факт;

- Например, команда:
  `(assert (length 150) (width 15) (weight "big"))`
  добавит в список фактов *три факта*, каждый из которых состоит из двух полей.

- Команды `retract`, `modify` и `duplicate` требуют, чтобы *факты были идентифицированы* с помощью *индекса факта* (fact-index) либо *адреса факта* (fact-address)

- Команды могут исполняться *как в режиме командной строки*, так и *включаться в CLIPS-программы* (и исполняться интерпретатором)

Задание фактов: **deffacts**

- Для задания исходного множества фактов используется конструкция **deffacts**, имеющая следующий синтаксис:

```
(deffacts <имя_группы_фактов> ["<комментарий>"] <факт>*),
```

где **<имя_группы_фактов>** – идентификатор символьного типа;

<комментарий> – необязательное поле комментария;

<факт>* – произвольная последовательность фактов, записанных через разделитель;

- Пример использования конструкции **deffacts**:

```
(deffacts stud "Студент"
     (student name John)
     (student spec "COMPUTER"))
```

- Факты, определенные конструкцией **deffacts**, добавляются в список фактов всякий раз при выполнении команды **reset**

Неупорядоченные факты

- *Неупорядоченные факты* - *список взаимосвязанных именованных полей*, называемых *слотами*
  - дают возможность *доступа к полям по именам*, в отличие от упорядоченных фактов, где поля специфицируются своим местоположением в факте
  - два *типа слотов*: *одиночные* и *мультислоты*. Одиночный слот (или просто слот) содержит единственное поле, мультислот может содержать любое число полей
- *Шаблон* – конструкция, используемая *для задания состава неупорядоченных фактов* (содержащихся в них слотов).
- Синтаксис шаблона :

```
(deftemplate <имя шаблона> ["<комментарий>"]
    <определение слота-1>
    .  .  .
    <определение слота-N>)
```

- Пример шаблона, содержащего три одиночных слота, представлен ниже:

```
(deftemplate object "Шаблон объекта"
    (slot name)
    (slot location)
    (slot weight))
```

- Пример конкретного неупорядоченного факта на основе данного шаблона:

```
(object (name car) (location 100) (weight 600))
```

Представление правил в базе знаний CLIPS

- Правила – основной способ представления знаний в CLIPS
- Для задания правил используется конструкция **defrule**, имеющая синтаксис:

```
(defrule <имя_правила> ["<комментарий>"]
     [<объявление>];
     <условный элемент>*                    ; Левая часть правила (антецедент)
     =>
     <действие>*)                           ; Правая часть правила (консеквент)
```

где **<имя_правила>** - идентификатор символьного типа, уникальный для данной группы правил;

<комментарий> - необязательное поле комментария;

<объявление> - необязательный элемент, позволяющий задавать дополнительные свойства правила (*например, значимость*) с помощью оператора **declare**;

<условный элемент>* - произвольная *последовательность условных элементов*;
<действие>* - произвольная последовательность действий.

Пример правила

- Правило, использующее упорядоченные факты:

```
(defrule R1
 (days 3)
 (works 100)
 =>
 (printout t crlf "Свободного времени нет" crlf)
 (assert (freetime "no")))
```

(days 3) – факт, указывающий число дней, оставшихся до зачета;

(works 100) – факт, указывающий процент невыполненных лаб. работ;

(printout t crlf "Свободного времени нет" crlf) – команда вывода сообщений в консоль (параметр **t** задает стандартный режим вывода; **crlf** – символ возврата и перевода курсора на новую строку);

(assert (freetime "no") – команда добавления в список фактов нового факта о том, что свободного времени нет;

Типы условных элементов

- Антецедент (левая часть) правила состоит из последовательности *условных элементов* (УЭ)
- Если текущее состояние базы данных удовлетворяет всем УЭ правила, то правило помещается в *список готовых к выполнению правил – агенду*
- В CLIPS используется следующие типы УЭ:
  - УЭ-образцы (Pattern Conditional Elements);
  - УЭ-проверки (Test Conditional Elements);
  - УЭ “ИЛИ” (Or Conditional Elements);
  - УЭ “И” (And Conditional Elements);
  - УЭ “НЕ” (Not Conditional Elements);
  - УЭ “Существует” (Exists Conditional Elements);
  - УЭ “Для всех” (Forall Conditional Elements);
  - логические УЭ (Logical Conditional Elements)

УЭ-образец

- *УЭ-образец* (Pattern Conditional Element) состоит из *совокупности ограничений на поля*, *масок полей* (wildcards) и *переменных*, используемых при сопоставлении УЭ с образцом – фактом или экземпляром объекта
  - УЭ-образец *удовлетворяется* каждой сущностью, которая удовлетворяет его ограничениям
- *Ограничения на поля* используются для проверки одного поля или слота факта либо экземпляра объекта.
  - ограничение на поле может состоять из единственного литерала или из нескольких связанных ограничений.
- В УЭ-образцах используются следующие конструкции:
  - литеральные ограничения (Literal Constraints);
  - одно и многоместные маски (Single- and Multifield Wildcards);
  - одно и многоместные переменные (Single- and Multifield Variables);
  - ограничения со связками (Connective Constraints);
  - предикатные ограничения (Predicate Constraints);
  - ограничения возвращаемым значением (Return Value Constraints).

Литеральное ограничение

- *Литеральное ограничение* не содержит переменных и масок, а задает точное значение (константу целого, вещественного, символьного или строкового типа, либо имя экземпляра), которое должно сопоставляться с полем образца.
- Все литеральные ограничения должны совпадать с соответствующими полями сопоставляемой сущности.
- Упорядоченный УЭ-образец содержит только литералы, имеет следующий синтаксис: (<constant-1> ... <constant-n>).
- Например, (data 1 one “two”)
- Пример УЭ-образца для неупорядоченных фактов:

  (person (name Bob) (age 20))

Одно- и многоместные маски

- *Одно- и многоместные маски* позволяют игнорировать некоторые поля в процессе сопоставления
  - *Одноместная маска* обозначается символом “?” и сопоставляется с любым значением, занимающим точно одно поле в соответствующем месте сопоставляемой сущности
  - *Многоместная маска*, обозначается парой символов “$?” и сопоставляется с любыми значениями, занимающими произвольное число полей в сопоставляемой сущности.

- Маски могут использоваться в одном образце в любых комбинациях
  - не допускается лишь использование многоместной маски в одноместном слоте (содержащем единственное поле) неупорядоченных фактов или объектов

- Например, УЭ **`(data ? blue red $?)`** будет сопоставляться со следующими упорядоченными фактами:

```
(data 1 blue red),
(data 5 blue red 6.9 "avto")
```

но не будет сопоставлен со следующими фактами:

```
(data 1.0 blue "red"),
(data 1 blue)
```

Одно- и многоместные переменные

- *Одно- и многоместные переменные* используются для запоминания значений полей, с целью их *дальнейшего использования в других условных элементах* антецедента или *в операторах консеквента* правила
- *Одноместные переменные* начинаются с символа “?”, за которым следует символьное значение, начинающееся с буквы
  - Например: **`?x`**, **`?var`**, **`?age`**
- *Многоместные переменные* начинаются с префикса “$?”, за которым также следует символьное значение, начинающееся с буквы.
  - Например: **`$?y`**, **`$?zum`**
- В именах переменных *не допускается использование кавычек*
- При первом появлении переменная работает так же, как в маске, т.е. связывается с любым значением в данном поле(ях)
- Последующие появления переменной требуют, чтобы поле сопоставлялось со связанным значением переменной
- *Имена переменных являются локальными* в пределах каждого правила.

Пример

- Пусть имеется три факта:

```
(data 2 blue green),
(data 1 blue),
(data 1 blue red)
```

и правило:

```
(defrule find-data-1
(data ?x ?y ?z)
=>
(printout t ?x " : " ?y " : " ?z crlf))
```

- УЭ данного правила будет сопоставляться с первым и третьим фактом, поэтому в результате срабатывания правила будет выведено:

```
1 : blue : red
2 : blue : green
```

Ограничения со связками

- *Ограничения со связками* – используются для связывания индивидуальных ограничений и переменных друг с другом с помощью связок & (“и”), | (“или”) и ~ (“не”), используемых в традиционном смысле
  - Старшинство операций обычное, за исключением случая, когда *первым ограничением является переменная, за которой следует связка* &. В этом случае первая переменная трактуется как отдельное ограничение, которое также должно удовлетворяться
  - Например, ограничение `?x&red|blue` трактуется как `?x&(red|blue)`, а не как `(?x&red)|blue`
- Пример правила с УЭ, содержащим ограничения со связками:

```
(defrule r1
(data (value ?x&~red&~green))
=>
(printout t "slot value = " ?x crlf))
```

- Например, для факта `(data (value blue))` это правило выведет сообщение:

```
slot value = blue
```

Предикатное ограничение

- *Предикатное ограничение* – позволяет ограничить значение поля, основываясь на истинности булевого выражения
  - Для этого используется предикатная функция, вызываемая в процессе сопоставления с образцом и возвращающая в случае неудачи символьное значение FALSE
  - если возвращается значение FALSE, то ограничение не удовлетворяется, в противном случае оно удовлетворяется
- Предикатное ограничение задается с помощью символа “:”, за которым следует вызов предикатной функции
- Может использоваться в комбинации с ограничением со связками, а также связанной переменной.
  - в последнем случае переменная сначала связывается некоторым значением, а затем к ней применяется предикатное ограничение
  - в таком варианте предикатные ограничения часто применяются для проверки типов данных
  - при этом в качестве предикатных функций используются встроенные функции CLIPS

Встроенные предикатные функции

- Встроенные функции CLIPS для проверки типов данных:
  - **(numberp <выражение>)** – функция возвращает значение TRUE, если <выражение> имеет числовой тип (integer или float), в противном случае возвращается символ FALSE;
  - **(floatp <выражение>)** – функция возвращает значение TRUE, если <выражение> имеет тип float, иначе возвращается символ FALSE;
  - **(integerp <выражение>)** – функция возвращает значение TRUE, если <выражение> имеет тип integer, иначе – символ FALSE;
  - (**symbolp <выражение>**) – функция возвращает значение TRUE, если <выражение> имеет тип symbol, иначе – символ FALSE;
  - **(stringp <выражение>)** – функция возвращает значение TRUE, если <выражение> имеет тип string, иначе – символ FALSE;

Пусть заданы факты: **(data 1) (data 2) (data red)).** Тогда для определения значений числового типа можно использовать УЭ

- **(data ?x&:(numberp ?x))**,

который сопоставится с первыми двумя фактами.

Тот же результат может быть получен использованием УЭ

(data ?x&~:(symbolp ?x))

Ограничение возвращаемым значением

- *Ограничение возвращаемым значением* – использует в качестве ограничения значение, возвращаемое внешней функцией.
  - Эта функция вызывается непосредственно из УЭ-образца с использованием следующего синтаксиса:

```
=<вызов-функции>
```

- Возвращаемое функцией значение одного из базовых типов подставляется непосредственно в УЭ-образец на позицию, из которой была вызвана функция, и используется далее как литеральное ограничение.
- Например, следующее правило, содержащее УЭ-образец с ограничением возвращаемым значением:

```
(defrule twice
(data (x  ?x) (y  =(* 2 ?x)))
=>)
```

будет сопоставляться со всеми неупорядоченными фактами, у которых значение в слоте y равно удвоенному значению слота x

Условный элемент-проверка

- *Условный элемент-проверка* имеет следующий синтаксис:

```
(test <function-call>).
```

- УЭ-проверка удовлетворяется, если функция, вызываемая из него, возвращает значение, отличное от FALSE.
  - Как и в предикатном ограничении, можно сравнивать уже связанную некоторым значением переменную, используя любые функции (алгебраическое и логическое сравнение, вызов внешних функций)
- В УЭ-проверку могут быть встроены внешние функции любого вида.
- В следующем правиле проверяется, что модуль разности двух чисел не меньше трех:

```
(defrule example-1
     (data ?x)
     (value ?y)
     (test (>= (abs (- ?y ?x)) 3))
     =>)
```

Условный элемент “ИЛИ”

- *Условный элемент “ИЛИ”* задается следующей конструкцией:

  (or <УЭ-1> ... < УЭ-N>)

  и удовлетворяется, если удовлетворяется хотя бы один УЭ внутри этой конструкции
  - Наличие такого УЭ позволяет сократить число правил, т.к. то же самое можно было бы записать множеством правил с одинаковой правой частью. При этом правило будет активизироваться несколько раз, по числу удовлетворяемых комбинаций
- Например, правило

```
(defrule r1
   (man stud)
   (or (spec computeer) (age 20))
  =>)
```

эквивалентно двум следующим:

```
(defrule r2
   (man stud)
   (spec computeer)
  =>)
(defrule r3
   (man stud)
   (age 20)
   =>)
```

Условный элемент “И”

- *Условный элемент “И”* задается следующей конструкцией:

```
(and <УЭ-1> ... < УЭ-N>)
```

и удовлетворяется, если удовлетворяются все УЭ внутри этой конструкции. В CLIPS все УЭ в антецедентах правил неявно объединены по “И”, однако использование УЭ “И” для явного задания конъюнктивной связи позволяет комбинировать УЭ “И” и УЭ “ИЛИ” в любых сочетаниях. Пример такой комбинации :

```
(defrule r1
(sys-mode search)
(or  (and (distance high)  (resol little))
     (and (distance low)  (resol big)))
=>)
```

- *Условный элемент “НЕ”* задается следующей конструкцией:

```
(not <УЭ>)
```

и удовлетворяется, если содержащийся внутри него УЭ не удовлетворяется. Предварительно связанные переменные могут использоваться внутри УЭ “НЕ” как свободные. Однако, переменные, которые связываются внутри УЭ “НЕ”, могут использоваться только в этом образце. Следующее правило ищет факты, у которых второе поле – red, а третье и четвертое поля *не* совпадают:

```
(defrule not-double
(not (data red ?x ?x))
=>)
```

Условный элемент “Существует”

- *Условный элемент “Существует”* имеет следующий синтаксис:

  (exists <УЭ-1> ... <УЭ-N>)

  и используется для определения, удовлетворяется ли *группа УЭ*, специфицированных внутри условного элемента “Существует”, хотя бы одним набором образцов-сущностей в базе данных

- Например, правило:

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

  будет активизировано, если в базе данных имеется хотя бы одна пара фактов, содержащих в первых полях значения **a** и **b**, а вторые поля которых совпадают

Условный элемент *“Для всех”*

- *Условный элемент “Для всех”* имеет следующий синтаксис:

  (forall <УЭ-1> ... <УЭ-N>)

  и используется для определения, удовлетворяется ли группа УЭ, специфицированных внутри условного элемента “Для всех”, *для каждого появления УЭ-1*.

- Например, следующее правило активизируется, если каждый студент научился чтению, письму и арифметике:

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

Логические условные элементы

- *Логические условные элементы* – обеспечивают возможность *поддержания истинности* различных сущностей (фактов и экземпляров), создаваемых правилами, использующими логические УЭ.
- Сущность-образец, создаваемая оператором правой части правила, может быть сделана *логически зависимой* от сущностей-образцов, сопоставляемых с *логическим* УЭ в антецеденте правил
- Сущности-образцы, сопоставляемые с логическими УЭ в антецеденте правил обеспечивают *логическую поддержку* фактам и экземплярам, создаваемым в консеквенте правила
- Сущность-образец может *логически поддерживаться несколькими группами сущностей-образцов* из одного или различных правил
- Если *любая поддерживающая сущность удаляется* из группы поддерживающих сущностей и не существует никаких других поддерживающих групп, то *поддерживаемая сущность удаляется из рабочей памяти*
- Сущность-образец имеет *безусловную поддержку*, если она *создается без логической поддержки*, т.е. с помощью конструкций deffacts, definstaces, с помощью высокоуровневых команд или правил без логической поддержки образцов.
  - безусловная поддержка сущности удаляет всю логическую поддержку (без удаления самой сущности), при этом дальнейшая логическая поддержка безусловно поддерживаемой сущности игнорируется

Логические условные элементы

- Удаление правила, генерировавшего для сущности логическую поддержку, *удаляет логическую поддержку*, генерируемую этим правилом, но *не влечет удаления сущности*, даже если для нее не осталось логической поддержки. Логический УЭ имеет следующий синтаксис:

```
(logical <УЭ>+)
```

- Логический УЭ группирует образцы точно так же, как УЭ “И” и может использоваться в сочетании с УЭ “И”, УЭ “ИЛИ” и УЭ “НЕ”. Однако *логические УЭ можно применять только в первых образцах правила*. Например, следующее правило допустимо:

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

- Вместе с тем, следующее правило является недопустимым:

```
(defrule not-ok-1
  (logical (a))
  (b)
  (logical (c))
  =>
  (assert (d)))
```