Министерство образования и науки Российской Федерации

Калужский филиал
федерального государственного бюджетного образовательного
учреждения высшего образования
**«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»**
(КФ МГТУ им. Н.Э. Баумана)

Ю.С. Белов, С.С. Гришунов

ОБЪЕКТЫ В СРЕДЕ CLIPS

Методические указания по выполнению лабораторной работы
по курсу «Экспертные системы»

Калуга - 2017

УДК 004.891
ББК 32.813
Б435

Б435 Белов Ю.С., Гришунов С.С. Объекты в среде CLIPS. Методические указания по выполнению лабораторной работы по курсу «Экспертные системы». — М.: Издательство МГТУ им. Н.Э. Баумана, 2017. — 40 с.

Методические указания по выполнению лабораторной работы по курсу «Экспертные системы» содержат краткое описание синтаксиса встроенного объектно-ориентированного языка CLIPS, рассмотрены этапы создания объектов, основные подходы ООП применительно к экспертным системам, а также задание на лабораторную работу..

Предназначены для студентов 4-го курса бакалавриата КФ МГТУ им. Н.Э. Баумана, обучающихся по направлению подготовки 09.03.04 «Программная инженерия».

УДК 004.891
ББК 32.813

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Настоящие методические указания составлены в соответствии с программой проведения лабораторных работ по курсу «Экспертные системы» на кафедре «Программное обеспечение ЭВМ, информационные технологии и прикладная математика» факультета фундаментальных наук Калужского филиала МГТУ им. Н.Э. Баумана.

Методические указания, ориентированные на студентов 4-го курса направления подготовки 09.03.04 «Программная инженерия», содержат краткое описание синтаксиса встроенного объектно-ориентированного языка CLIPS, рассмотрены этапы создания объектов, основные подходы ООП применительно к экспертным системам, а также задание на лабораторную работу.

Методические указания составлены для ознакомления студентов со средой NASA CLIPS v.6.2 и овладения навыками работы с объектами. Для выполнения лабораторной работы студенту необходимы минимальные знания основ объектно-ориентированного программирования, а также навыки программирования на любом высокоуровневом языке программирования.

ЦЕЛЬ И ЗАДАЧИ РАБОТЫ, ТРЕБОВАНИЯ К РЕЗУЛЬТАТАМ ЕЕ ВЫПОЛНЕНИЯ

Целью выполнения лабораторной работы является формирование практических навыков по работе с объектами в среде CLIPS.

Основными задачами выполнения лабораторной работы являются:

1. изучить основные команды для работы с объектами в среде CLIPS
2. научиться создавать абстрактные и конкретные классы
3. научиться использовать механизм наследования классов в CLIPS
4. формирование навыков назначения обработчиков событий
5. научиться создавать, удалять и изменять объекты.

Результатами работы являются:

- Созданные в среде CLIPS классы, объекты и правила
- Сохраненные в файлах скрипты
- Подготовленный отчет

КРАТКАЯ ХАРАКТЕРИСТИКА ОБЪЕКТА ИЗУЧЕНИЯ, ИССЛЕДОВАНИЯ

CLIPS поддерживает концепцию объектно-ориентированного программирования и имеет встроенный объектно-ориентированный язык *CLIPS Object-Oriented Language (COOL)*. Объекты позволяют объединить данные со способами их обработки. Объекты *CLIPS* можно использовать в правилах и функциях в качестве данных почти так же, как факты или переменные.

Системные классы

COOL имеет семнадцать системных классов: `OBJECT`, `USER`, `INITIAL-OBJECT`, `PRIMITIVE`, `NUMBER`, `INTEGER`, `FLOAT`, `INSTANCE`, `INSTANCE-NAME`, `INSTANCE-ADDRESS`, `ADDRESS`, `FACT-ADDRESS`, `EXTERNAL-ADDRESS`, `MULTIFIELD`, `LEXEME`, `SYMBOL` и `STRING`. Пользователь не имеет права удалять и изменять эти классы. На рис. 1 изображена диаграмма наследования этих классов.

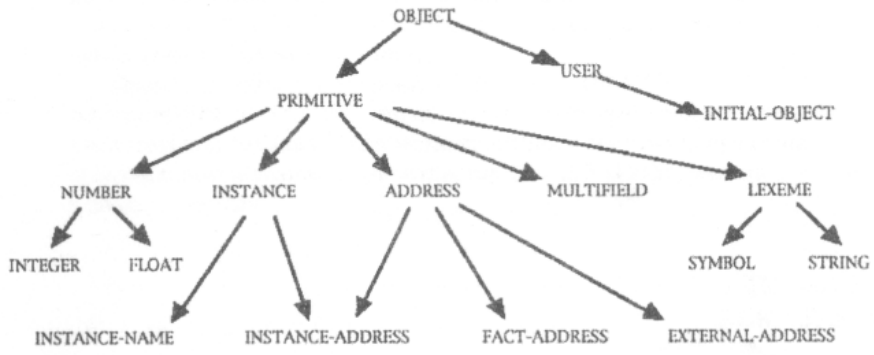

Рис.1 - Связи наследования системных классов

Все эти классы, кроме `INITIAL-OBJECT`, являются абстрактными и могут использоваться только для наследования. Класс `OBJECT` является родительским для всех остальных классов, включая также классы, определенные пользователем. Все классы, определяемые пользователем, должны наследоваться от класса `USER`, так как этот

класс имеет все стандартные функции, например, такие как инициализация и удаление представителей класса.

Системный класс `PRIMITIVE` и все его подклассы используются в основном для ограничений в функциях и методах, но при необходимости пользователем могут использоваться обработчики сообщений для этих классов, а также определены классы-наследники. Исключение составляют лишь классы `INSTANCE`, `INSTANCE-NAME`, `INSTANCE-ADDRESS`, которые могут быть использованы только в методах.

Системный класс `INITIAL-OBJECT` используется для создания объекта по умолчанию `initial-object`, который создается всякий раз при использовании команды `reset`. Этот системный класс является конкретным и активным (эти понятия будут введены чуть позже), поэтому он может быть использован для сопоставления в левой части правила.

Конструктор defclass

Конструкция `defclass` создает новый пользовательский класс в среде *CLIPS*. Он определяет свойства (слоты) и поведение (обработчики сообщений) класса объектов. Данная конструкция состоит из нескольких элементов:

- имя;
- список родительских классов, от которых данный класс наследует свойства и обработчики сообщений;
- идентификатор, определяющий роль данного класса;
- идентификатор, определяющий, смогут ли представители данного класса быть использованы в левой части правила;
- список свойств (слотов), определяющих данный класс.

Все классы, объявляемые пользователем, должны наследоваться хотя бы от одного класса, и *COOL* предоставляет систему классов, от которых могут быть порождены новые классы.

Конструкция `defclass` имеет следующий синтаксис:

```
(defclass <имя> [<комментарии>]
(is-a <имя родительского класса>+)
[<role>]
[<pattern-match-role>]
<slot>*
<handler-documentation>*)

<role>  :: =  (role concrete  |  abstract)

<pattern-match-role>
         ::=  (pattern-match reactive  |  non-
       reactive)

<slot>  ::=  (slot <имя> <ограничения>*)   |
          (single-slot < имя > < ограничения >*)
          |
          (multislot < имя > < ограничения >*)

<ограничения>::=<default-facet> I  <storage-
   facet>  |
   <access-facet> |  <propagation-facet> |
   <source-facet> | <pattern-match-facet>  |
   <visibility-facet>  |  <create-accessor-
   facet>
<override-message-facet>  |  <constraint-
   attributes>

<default-facet>  ::=
         (default ?DERIVE I ?NONE |  <expression>*)
         | (default-dynamic <expression>*)
<storage-facet> ::=  (storage local  |  shared)

<access-facet>  ::=  (access  read-write  |  read-
only
         |initialize-only)
```

```
  <propagation-facet> ::= (propagation inherit |
no-inherit)

  <source-facet> ::=  (source exclusive  |
  composite)

  <pattern-match-facet>     ::=     (pattern-match
reactive | non-reactive)

  <visibility-facet> ::=  (visibility private |
  public)

  <create-accessor-facet>  ::=  (create-accessor  ?
NONE
            |read | write  |  read-write)
  <override-message-facet>::=(override-message
            ?DEFAULT|<message-name>)
  <handler-documentation> ::= (message-handler
            <name> [<handler-type>])
  <handler-type> ::= primary |  around I before  |
  after
```

При создании класса с дублирующим именем будут удалены все подклассы и обработчики сообщений текущего класса. Ошибка возникнет только в том случае, если у текущего класса или у одного из его подклассов существуют конкретные представители.

Множественное наследование

Если один класс наследуется от другого, то первый класс называется *наследником, подклассом* или *потомком,* а второй класс — *предком* или *суперклассом* первого. Каждый класс, объявляемый пользователем, должен иметь одного непосредственного предка, который объявляется после `is-a` в конструкции `defclass`. Если класс имеет более одного предка, то такая ситуация называется *множественным наследованием. COOL* строит список классов

предков для каждого нового класса, называемый *списком предшествующих классов (class precedence list)*. Новый созданный класс наследует все слоты и обработчики сообщений от каждого из классов, указанных в данном списке. Слово «предшествующий» означает, что слоты и обработчики класса в списке будут иметь более высокий приоритет по сравнению с теми, которые будут объявлены позже. Класс, попавший в список раньше любого другого, называется *наиболее специфичным* по отношению к данному классу. Все классы списка предшествующих классов оканчиваются системным классом OBJECT, и большинство (если не все) пользовательских классов ограничиваются системными классами `USER` и `OBJECT`. Посмотреть список предшествующих классов можно при помощи функции `describe-class`.

COOL имеет иерархию наследования от непосредственных суперклассов для определения списка предшествующих классов для нового класса. *COOL* рекурсивно определяет два правила для непосредственных (прямых) суперклассов:

- Класс имеет более высокий приоритет, чем любой из его суперклассов.
- Определение класса задает приоритет между его непосредственными суперклассами.

Если класс имеет более одного списка предшествующих классов, то *COOL* выбирает наиболее простой список, используя для этого метод поиска в глубину. Эта эвристика максимально долго сохраняет родственное дерево класса. Например, если ребенок наследует генетические черты от мамы и папы, а они в свою очередь наследуют черты от своих родителей, тогда список предшествования для ребенка будет следующий: ребенок, мама, бабушка и дедушка по маминой линии, папа, бабушка и дедушка по папиной линии. Также для ребенка может быть и другой список предшествования, например если поменять местами родителей мамы и папы. В таком случае *COOL* выберет первый список, так как он сохраняет фамильное дерево настолько долго, насколько это возможно.

Приведем несколько примеров, иллюстрирующих построение списка предшествующих классов.

Исходный код:

```
(defclass A  (is-a USER))
```

Класс A непосредственно наследуется от системного класса USER. Список предшествования для класса A будет иметь следующий вид: A USER OBJECT.

Исходный код:

```
(defclass B (is-a USER))
```

Класс B непосредственно наследуется от системного класса USER. Список предшествования для класса в будет иметь следующий вид: B USER OBJECT.

Исходный код:

```
(defclass C (is-a A B))
```

Класс C непосредственно наследуется от классов A и B. Список предшествования для класса с будет иметь следующий вид: C A USER OBJECT B .

Исходный код:

```
(defclass D (is-a B A))
```

Класс D непосредственно наследуется от классов B и A. Список предшествования для класса D будет иметь следующий вид: D B USER OBJECT A.

Исходный код:

```
(defclass E (is-a A C))
```

По правилам класс A должен предшествовать классу с. Однако, класс C наследуется от класса A и не может следовать за ним в списке предшествующих классов. Поэтому данный пример является неверным.

Абстрактные и конкретные классы

Поле <role> является полем указания абстрактности и может принимать два значения: abstract и concrete. Класс, описанный как abstract, является абстрактным. Все неабстрактные классы описываются как concrete. По умолчанию все классы являются неабстрактными.

Абстрактные классы используются только для наследования, и не может быть создано ни одного конкретного представителя данного класса. Конкретные классы могут иметь конкретных представителей. Если класс указан как абстрактный, то это говорит о том, что *COOL* будет генерировать ошибку всегда при попытке использования функции make-instance по отношению к данному классу. Если в

описании класса не указано, класс какого типа создается (абстрактный или конкретный), то он будет иметь тип класса, от которого он наследуется. Если класс наследуется от системного класса `USER` и в его описании не указан тип класса, то он будет конкретным.

Активные и неактивные классы

Поле `<pattern-match-role>` указывает на сопоставимость объекта в правилах. При установке этого поля в `reactive` объект является активным, а в случае `non-reactive` — неактивным. Если это поле не определено в описании класса, то оно определяется соответствующим полем от наследуемого класса.

Объекты активных классов могут ставиться в соответствие образцам объектов в правилах. Объекты неактивных классов не могут ставиться в соответствие образцам объектов в правилах, и они не могут выступать в качестве образцов объектов для сопоставления. Абстрактный класс не может быть активным. Если в описании класса не указано, класс какого типа создается (активный или неактивный), то он будет иметь тип класса, от которого он наследуется.

Слоты класса

Слоты — это «заполнители» (текстовый элемент электронного шаблона, заменяемый реальным элементом) значений, связанные с представителями классов, объявленных пользователем. Количество слотов в классе ограничено только пределами памяти вашего компьютера. Имя слота может принимать любое символьное значение. Исключение составляют лишь ключевые слова `is-a` и `name`, которые зарезервированы для использования в образцах объектов.

Приведем простой пример работы с классами.

Исходный код:

```
(defclass A  (is-a USER)
    (slot fooA)
    (slot barA)
)
```

```
(defclass B  (is-a A)
   (slot fooB)
   (slot barB)
)
```

Список предшествования для класса A будет иметь следующий вид: A USER OBJECT. Представители класса A будут имеет два слота: fooA и barA. Список предшествования для класса B будет иметь следующий вид: B A USER OBJECT. Представители класса B будут имеет четыре слота: fooB, barB, fooA и barA.

Так же, как слоты определяют классы, *грани (facets)* определяют слоты. Грани описывают различные особенности слота, которые истинны для всех объектов, имеющих данный слот. К ним относятся: значение по умолчанию, способ хранения, тип доступа, способ наследственного воспроизведения, источники, активность при сопоставлении образцов, видимость, автоматическое создание обработчиков сообщений для доступа к слотам, переопределение сообщений, ограничения. Каждый объект может иметь свое собственное значение конкретного слота, исключения составляют лишь совместные (общие) слоты.

Слот может состоять либо из одного поля, либо из мультиполя. По умолчанию слот определяется как одиночное поле. Ключевое слово multisiot означает, что слот состоит из мультиполя, которое может содержать ноль или более полей. Ключевые слова slot и single-slot определяют, что данный слот может состоять только из одного поля. Значения мультиполей слотов хранятся как обычные мультиполя, и с ними можно работать посредством обычных функций, таких как nth$ и length$, но при помощи посылки сообщений. Также существует функция для установки мультиполей в слотах — slot-insert$. Одиночные поля слота хранятся в *CLIPS* как обычные примитивные типы данных, такие как integer или string.

Приведем небольшой пример.

Исходный код:

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a
USER)
   (role concrete)
   (multisiot foo (create-accessor read)
     (default abc def ghi))
)
CLIPS> (make-instance a of A)
[a]
CLIPS> (nth$ 2 (send [a] get-foo))
def
CLIPS>
```

Свойства `default` и `default-dynamic` могут быть использованы для указания начального значения атрибута, когда создается новый представитель класса. Если у слота не указано ни одно из этих свойств, значение слота по умолчанию будет получено из наложенных на него ограничений. Значения по умолчанию напрямую устанавливаются слотам без использования сообщений, в отличие от определения слотов в вызовах функции `make-instance`.

Свойство `default` статическое, т.е. оно вычисляется один раз при объявлении класса. Это значение хранится вместе с классом и назначается соответствующему слоту каждый раз при создании нового экземпляра класса. Ключевое слово `?DERIVE` используется тогда, когда начальное значение получается из ограничения для слота. По умолчанию этот атрибут слота имеет следующее значение (`default ?DERIVE`). Если используется ключевое слово `?NONE`, то атрибут не имеет начального значения.

Свойство `default-dynamic` динамическое, т.е. начальное значение вычисляется всякий раз при создании нового экземпляра класса, а результат вычисления устанавливается соответствующему слоту.

Приведем пример использования этих свойств.

Исходный код:

```
CLIPS> (clear)
CLIPS> (setgen
1) 1
CLIPS>
 (defclass A  (is-a
    USER)
    (role concrete)
    (slot foo  (default-dynamic
(gensym))
     (create-accessor read))
)
CLIPS> (make-instance a1 of
A)
[a1]
CLIPS> (make-instance a2 of
A)
[a2]
CLIPS> (send [a1] get-
foo)
gen1
CLIPS> (send [a2] get-foo)
gen2
CLIPS>
```

Фактическое значение копии экземпляра может храниться либо вместе с самим экземпляром, либо непосредственно с классом. Свойство `local` указывает на то, что значение экземпляра будет храниться вместе с экземпляром класса; именно это значение данного свойства используется по умолчанию. Свойство `shared` указывает на то, что значение экземпляра будет храниться вместе с самим классом. Если значение экземпляра хранится с помощью свойства `local,` то любой экземпляр данного класса может иметь свои собственные значения слотов. Однако если значения слотов хранятся вместе с классом, то все представители данного класса будут иметь

одинаковые значения слотов. Если это значение изменить для какого-либо слота, то оно измениться и для слотов всех остальных представителей.

Слоты с указанным свойством `shared` всегда подбирают динамические значения по умолчанию при инициализации экземпляра, но этот слот будет игнорировать статическое значение по умолчанию, пока в нем не появится значение. Любые изменения в таких слотах говорят *CLIPS* о том, что необходимо изменить все активные объекты, содержащие данный слот.

Приведем небольшой пример работы с этим свойством.

Исходный код:

```
CLIPS> (clear)
CLIPS>
 (defclass A (is-a USER)
   (role concrete)
   (slot foo  (create-accessor write)
     (storage shared)(default 1))
   (slot bar (create-accessor write)
     (storage shared) (default-dynamic 2))
   (slot woz (create-accessor write)
     (storage local))
)
CLIPS> (make-instance a of A)
[a]
CLIPS> (send  [a]  print)
[a] of A
(foo 1)
(bar 2)
(woz nil)
CLIPS> (send [a] put-foo 56)
56
CLIPS> (send [a] put-bar 104)
104
CLIPS> (make-instance b of A)
[b]
```

```
CLIPS> (send [b] print)
[b] of A
(foo 56)
(bar 2)
(woz nil)
CLIPS> (send [b] put-foo 34)
34
CLIPS> (send [b] put-woz 68)
68
CLIPS>  (send  [a]  print)
[a] of A
(foo 34)
(bar 2)
(woz nil)
CLIPS>  (send  [b]  print)
[b] of A
(foo 34)
(bar 2)
(woz 68)
CLIPS>
```

Свойство <access-facet> указывает на правила доступа к данному атрибуту. Существует три основных режима доступа к атрибуту объекта:

- только инициализация (initialize-only);
- только чтение (read-only);
- чтение и запись (read-write).

Свойство read-write используется по умолчанию и говорит о том, что в слот можно либо записывать, либо из него можно читать. Свойство read-only говорит о том, что значение слота можно только прочитать, а установка значения в таком случае происходит при объявлении слота в значениях по умолчанию. Свойство initialize-only такое же, как read-only, за исключением того, что значение данного слота может быть установлено при помощи определения слотов в вызовах make-instance и init. Рассмотрим небольшой пример.

Исходный код:

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a
   USER)
  (role concrete)
  (slot foo (create-accessor write)
    (access read-write))
  (slot bar (access read-only)(default abc))
  (slot woz  (create-accessor write)
  (access initialize-only))
)
CLIPS>
(defmessage-handler A put-bar (?value)
 (dynamic-put  (sym-cat bar)  ?value))
CLIPS> (make-instance a of A (bar 34))
[MSGFUN3] bar slot in [a] of A: write access
denied.
[PRCCODE4]  Execution halted during  the
actions of message-handler put-bar
primary in class A
FALSE
CLIPS> (make-instance a of A (foo 34) (woz 65))
[a]
CLIPS> (send [a] put-bar 1)
[MSGFUN3] bar slot in [a]  of A: write access
denied.
[PRCCODE4] Execution halted during  the  actions
of message-handler put-bar primary in class A
FALSE
CLIPS> (send [a] put-woz 1)
[MSGFUN3] woz slot in [a] of A: write access
denied.
```

```
[PRCCODE4]  Execution halted during  the
actions  of message-handler put-bar primary in
class A
FALSE
CLIPS> (send [a]
print)
[a] of A
(foo 34)
(bar abc)
(woz 65)
CLIPS>
```

Свойство <propagation-facet> указывает на то, будет ли данный атрибут наследоваться последующими объектами. Свойство может принимать два значения: inherit и no-inherit, что означает, соответственно, «наследуется» и «не наследуется». Приведем пример использования этого свойства.

Исходный код:

```
CLIPS>
(clear)
CLIPS>
 (defclass A (is-a
    USER)
    (role concrete)
    (slot foo  (propagation
    inherit))
    (slot bar  (propagation no-inherit))
)
CLIPS> (defclass B  (is-a
A))
CLIPS>  (make-instance a of
A)
[a]
 CLIPS> (make-instance b of
B)
 [b]
```

```
CLIPS> (send [a] print)
[a] of A
(foo nil)
(bar nil)
CLIPS> (send [b] print)
[b] of B
(foo nil)
CLIPS>
```

В момент создания объект получает слоты, определенные в классах, находящихся в его списке наследования. При получении некоторого слота по умолчанию его свойства определяются гранями, заданными в наиболее определенном классе (находящемся левее в списке предшествующих классов). Незаданные грани получают значения по умолчанию. Для изменения этого поведения служит свойство `source` источника. Это свойство может принимать одно из двух значений: `exclusive` и `composite`. Значение `exclusive` реализует поведение по умолчанию. Если используется значение `composite`, то при создании слота неопределенные свойства, не заданные в наиболее определенном классе, берутся из следующего, менее определенного, класса и т.д. Таким образом, в формировании свойств слотов могут участвовать несколько классов.

Каждый раз при изменении значения слота в представителе класса происходит сопоставление образцу. Однако это не всегда необходимо. Для этого и используется следующее свойство. Свойство `<source-facet>` обеспечивает правильность сопоставления. Если оно установлено как `reactive`, то это означает, что изменения слота будут влиять на сопоставление. Этот режим используется по умолчанию. Если же используется `non-reactive`, то изменение слота не оказывает влияния на процесс сопоставления. Покажем работу этого свойства на примере.

Исходный код:

```
CLIPS> (clear)
CLIPS>
```

```
(defclass A (is-a USER)
  (role concrete)
  (pattern-match reactive)
  (slot foo  (create-accessor write)
    (pattern-match non-reactive))
)
CLIPS>
(defclass B  (is-a USER)
  (role concrete)
  (pattern-match reactive)
  (slot foo  (create-accessor write)
    (pattern-match reactive))
)
CLIPS>
(defrule Create
  ?ins<-(object  (is-a A B))
=>
  (printout t "Create "  (instance-name ?ins)
  crlf)
)


CLIPS>
(defrule Foo-Access
  ?ins<-(object  (is-a A B)   (foo ?))
=>
  (printout t "Foo-Access "
    (instance-name ?ins) crlf)
)
CLIPS> (make-instance a of A)
[a]
CLIPS> (make-instance b of B)
[b]
CLIPS>  (run)
Create  [b]
```

```
Foo-Access [b]
Create [a]
CLIPS> (send [a] put-foo 1)
1
CLIPS> (send [b] put-foo 1)
1
CLIPS> (run)
Foo-Access
[b]
CLIPS>
```

Обработчики сообщений имеют доступ только к слотам, объявленным внутри данного класса. Однако существует возможность разрешить доступ к слотам обработчикам всех классов-передков и классов-наследников. Для этого используется свойство `<visibility>`. Существует два режима видимости атрибутов: `private` и `public`. `Private` означает, что данный слот будет виден только в пределах данного класса, а `public` — во всех наследуемых классах, а также во всех суперклассах. Приведем небольшой пример.

Исходный код:

```
CLIPS> (clear)
CLIPS>
(defclass A  (is-a USER)
  (slot foo  (visibility private))
)
CLIPS>
(defclass B  (is-a A)
  (role concrete)
)
CLIPS>
(defmessage-handler B get-foo () ?self:foo)
[MSGFUN6] Private slot foo of class A cannot be
accessed directly by handlers attached to class
B
  [PRCCODE3] Undefined variable self:foo
referenced  in message-handler.
  ERROR:
(defmessage-handler MAIN::B get-foo
()
?self:foo
)
CLIPS>
```

Свойство `<create-accessor-facet>` инструктирует *CLIPS* автоматически создавать явные обработчики сообщений для чтения и/или записи. Имеет четыре режима работы — `?NONE`, `read`, `write` и `read-write`. По умолчанию используется последнее значение.

Если данное свойство указано как `?NONE`, то обработчиков сообщений не создается.

Если данное свойство указано как `read`, то создается обработчик сообщения следующего вида:

`(defmessage-handler <class> get-<slot-name> primary () ?self:<slot-name>)`

Если данное свойство указано как `write`, то создается обработчик сообщения следующего вида:

```
(defmessage-handler <class> put-<slot-name> primary (?value) (bind ?self:<slot-name> ?value)
```

или следующий обработчик для мультиполей:

```
(defmessage-handler <class> put-<slot-name> primary ($?value) (bind ?self:<slot-name> ?value)
```

Если данное свойство указано как `read-write`, то создаются сразу два указанных выше обработчика.

Значением свойства `<access-facet>` является значение свойства `<create-accessor-facet>` по умолчанию. Соответствие значений указано в табл. 1.

Таблица 1

Соответствие значений <access-f acet> И <create-accessor-f acet>

<access-facet>	<create-accessor-facet>
read-only	read
write-only	write
initialize-only	?NONE

Приведем небольшой пример работы с данным свойством.

Исходный код:

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
  (role concrete)
  (slot foo  (create-accessor write))
  (slot bar (create-accessor read))
)
CLIPS> (make-instance a of A (foo 36))
[a]
CLIPS> (make-instance b of A (bar 45))
[MSGFUN1] No applicable primary message-handlers found for put-bar.
FALSE
```

```
CLIPS>
```

Некоторые версии COOL поддерживают функции, которые устанавливают слоты с помощью посылки сообщений, таких как `make-instance`, `initialize-instance`, `message-modify-instance` и `message-duplicate-instance`. По умолчанию все эти функции пытаются установить значение слота с помощью функции `put-<имя-слота>`. Однако пользователю может понадобиться другая функция для установки значения слота. Для этого и используется свойство `<override-message>`. Рассмотрим его работу на примере.

Исходный код:

```
CLIPS>
(clear)
CLIPS>
(defclass A  (is-a
    USER)
   (role concrete)
   (slot special  (override-message special-
put))
)
CLIPS>
(defmessage-handler A special-put primary  (?
value)
    (bind ?self:special ?value))
CLIPS> (watch messages)
CLIPS> (make-instance a of A (special 65))
MSG >> create ED:1  <Instance-a>)
MSG << create ED:1  (<Instance-a>)
MSG >> special-put ED:1  (<Instance-a> 65)
MSG << special-put ED:1  (<Instance-a> 65)
MSG >> init ED:1   <Instance-a>)
MSG << init ED:1   <Instance-a>)
[a]
CLIPS> (unwatch messages)
CLIPS>
```

В данном примере была использована функция (watch messages), которая позволяет пошагово посмотреть создание представителя класса.

CLIPS поддерживает проверку статических и динамических ограничений для классов и объектов классов. Проверка статических ограничений осуществляется при выполнении конструктора или команды, определяющих значение слота. Режим статической проверки ограничений включен по умолчанию. Эту установку можно изменить с помощью функции set-static-constraint-checking.

Кроме статической проверки *CLIPS* поддерживает возможность динамической проверки ограничений слотов. Если этот режим включен, то значения слотов проверяются при каждом изменении. По умолчанию этот режим выключен. Данную установку можно изменить с помощью функции set-dynamic-constraint-checking.

Если нарушение ограничения происходит в момент выполнения программы, то выполнение будет завершено.

Свойство <handler-documentation> используется только для указания дополнительной справочной информации о классе и игнорируется *CLIPS*.

ЗАДАЧИ И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Функции работы с объектами

Объекты управляются посредством посылки им сообщений. Для этого используется функция `send`, которая имеет следующий синтаксис:

```
(send        <имя-
   объекта>
   <имя-сообщения>
   <выражение>*
)
```

Функция `send` возвращает результат выполнения сообщения. Для создания экземпляра какого-либо класса используется следующая конструкция:

```
(make-instance <instance-definition>)
```

Создание экземпляра класса с помощью конструкции `make-instance` происходит в несколько шагов:

1. Если уже существует представитель класса с таким именем, то ему посылается сообщение `delete`. Если это сообщение по каким-либо причинам не выполняется, то происходит прерывание создания экземпляра класса.
2. Создается новый экземпляр класса с уникальным именем.
3. Новый экземпляр получает сообщение `create`.
4. Вычисляются значения слотов. Если это происходит с ошибками, то удаляется весь экземпляр и конструкция вернет значение

   `FALSE`.
5. Созданный экземпляр получает сообщение **`init`.**

Подобно конструктору `deffacts` существует конструктор `definstances`. Объекты, объявленные при помощи данной конструкции, будут загружаться в рабочую память всегда при вызове команды `reset`. Приведем пример работы с данной конструкцией.

Исходный код:

```
CLIPS> (clear)
```

```
CLIPS>
(defclass A (is-a USER)  (role concrete)
   (slot x  (create-accessor write)  (default
   1))
)
CLIPS>
(definstances A-
   OBJECTS
   (al of A)
   (of A  (x 65))
)
CLIPS> (watch
instances)
CLIPS>  (reset)
==> instance [initial-object] of INITIAL-OBJECT
==> instance [a1]  of A
==> instance  [gen1]  of A
CLIPS> (reset)
<== instance  [initial-object]  of INITIAL-
OBJECT
<== instance [a1] of A
<== instance [gen1] of A
==> instance [initial-object] of INITIAL-OBJECT
==> instance [a1]  of A
==> instance [gen2] of A
CLIPS> (unwatch instances)
CLIPS>
```

Также для создания экземпляра класса можно использовать конструкцию active-make-instance, которая имеет следующий синтаксис:

```
(active-make-instance <instance-definition>)
<instance-definition> ::= [<имя экземпляра>]  of
   <имя класса>
   <изменяемые слоты>*
```

<изменяемые слоты> ::= (<имя слота> <выражение>*)

Данная конструкция в случае удачного выполнения возвращает имя созданного класса, в противном случае возвращается FALSE.

Для установки и чтения значения слота можно использовать функции dynamic-get и dynamic-put. Приведем небольшой пример работы с этими функциями.

Исходный код:

```
CLIPS>
(clear)
CLIPS>
(defclass A (is-a
   USER) (role
   concrete)
   (slot x  (create-accessor read)
   (default abc))
)
CLIPS> (make-instance a of A)
[a]
CLIPS> (sym-cat  (send [a] get-x)  def)
abcdef
CLIPS> (send  [a]  put-x "New value.")
"New value."
CLIPS>
```

Для удаления экземпляра объекта используется сообщение delete. Синтаксис будет иметь следующий вид:

```
(send <имя-экземпляра> delete)
```

Для изменения значения экземпляра используется функция modify-instance, которая имеет следующий синтаксис:

```
(modify-instance <имя-экземпляра> <слоты>*)
```

Приведем небольшой пример использования данной функции.

Исходный код:

```
CLIPS>
(clear)
```

```
CLIPS>
(defclass A (is-a
   USER)
   (role concrete)
   (slot foo)
   (slot bar)
)
CLIPS> (make-instance a of A)
[a]
CLIPS> (modify-instance a  (foo 0))
TRUE
CLIPS>
```

Для дублирования значения экземпляра используется функция duplicate-instance, которая имеет следующий синтаксис:

(duplicate-instance <имя-экземпляра1> [to <имя-экземпля-ра2>] <слоты>*)

Приведем небольшой пример использования данной функции.

Исходный код:

```
CLIPS> (clear)
CLIPS> (setgen 1)
1
CLIPS>
(defclass A (is-a USER)
(role concrete)
  (slot foo (create-accessor write))
  (slot bar  (create-accessor write))
)
CLIPS> (make-instance a of A (foo 0) (bar 4))
[a]
CLIPS> (duplicate-instance a)
[gen1]
CLIPS>
```

Функция (defclass-module <defclass-name>) возвращает имя модуля, в котором данный класс определен.

Функция (superclassp <имя класса 1> <имя класса 2>) возвращает TRUE, если первый класс является родительским по отношению ко второму. В противном случае она возвращает FALSE.

Функция (list-defclasses [<имя модуля>]) отображает список всех классов, объявленных в текущем модуле.

Функция (browse-classes [<имя класса>]) отображает все наследственные связи данного класса, т.е. связи со всеми наследуемыми от заданного класса классами.

Функция (describe-ciass <имя класса>) показывает полное описание класса.

Для просмотра конкретного класса можно использовать команду ppdefclass:

```
(ppdefclass имя_класса)
```

Обработчики сообщений

Объекты управляются посредством посылки сообщений с помощью функции send. Defmessage-handler — это конструкция, с помощью которой точно определяется поведение объектов в ответ на посылку одиночных сообщений. Реализация сообщения состоит из кусков процедурного кода, называемого *обработчиками сообщений* (или просто обработчиками). Каждый класс в списке предшествующих классов объекта может иметь свои обработчики сообщений. В этом случае объект класса и все его суперклассы расширяют область действия обработчика сообщения. Каждый обработчик класса обрабатывает только часть сообщений, назначенных данному классу. В пределах класса обработчики отдельных сообщений могут быть разделены на четыре группы: primary, before, after и around. Смысл каждого из них раскрыт в табл. 2.

Обработчики before и after используются только для вспомогательной работы и все возвращаемые ими значения игнорируются. Обработчик before выполняет свои действия до начала работы обработчика primary, а after — после. Обычно воспринимаются только возвращаемые значения обработчика

primary, но иногда и обработчик around также может вернуть значение. Обработчик around позволяет пользователю скрывать код отсутствующего обработчика primary. Эти обработчики начинают работать ранее других обработчиков, а возобновляют свои действия после завершения их работы.

Таблица 2

Описание обработчиков

Вид	Описание
primary	Выполняет большинство действий с сообщениями
before	Выполняет вспомогательную работу до начала работы обработчика primary
after	Выполняет вспомогательную работу после окончания работы обработчика primary
around	Настраивает работу оболочки во время отсутствия работы обработчика primary

Обработчик сообщений состоит из семи частей:

1. имя класса, за которым закрепляется обработчик (класс с таким именем должен быть заранее объявлен);
2. имя сообщения;
3. тип обработчика (по умолчанию primary);
4. комментарии;
5. список параметров, который будет передаваться обработчику во время его выполнения;
6. групповой параметр;
7. группа выражений, которые выполняются всякий раз при вызове данного обработчика.

Конструктор defmessage-handler имеет следующий синтаксис:

```
(defmessage-handler <имя класса> <имя
   сообщения>
  [<тип сообщения>]   [<комментарии>]
  (<parameter>*  [<wildcard-parameter>])
```

```
   <действия>*
)
<тип сообщения> ::= around | before  | primary
|  after <parameter> ::= <single-field-
variable>
<wildcard-parameter> ::= <multifield-variable>
```

Обработчики сообщений уникально идентифицируются классом, именем и типом. Обработчики сообщений никогда не вызываются напрямую. Когда пользователь посылает сообщение объекту, *CLIPS* выбирает и назначает соответствующему обработчику прикрепиться к данному объекту и работать вместе с ним. Этот процесс называется *отправкой сообщений (message dispatch)*.

Все обработчики имеют неявный параметр **?self,** который связывается с активным экземпляром объекта в сообщении. Имя этого параметра зарезервировано и не может быть явно объявлено в списке параметров. Приведем небольшой пример работы с данным параметром.

Исходный код:

```
CLIPS> (clear)
CLIPS>  (defclass A  (is-a USER)   (role
concrete))
CLIPS> (make-instance a of A)
[a]
CLIPS>
(defmessage-handler A print-args   (?a ?b $?c)
  (printout t  (instance-name ?self)   " " ?a " "
                          ?b
      " and "  (length$ ?c)   " extras:  " ?c
                        crlf)
)
CLIPS>  (send  [a]  print-args 1 2)
[a]  1 2 and 0 extras:   ()
CLIPS> (send [a] print-args a b c d )
[a]  a b and 2 extras:   (c d)
CLIPS>
```

Для задания значения какому-либо слоту можно также использовать уже известную функцию `bind`. В данном случае она будет иметь следующий синтаксис:

```
(bind ?self:<slot-name> <value>*)
```

Приведем пример использования этой функции.

Исходный код:

```
CLIPS>
(defmessage-handler A set-foo  (?
  value)
   (bind ?self:foo ?value)
)
CLIPS> (send [a]  set-foo
34)
34
CLIPS>
```

В *COOL* существуют несколько стандартных для всех объектов обработчиков сообщений:

```
(defmessage-handler USER init primary ())
(defmessage-handler USER delete primary ())
(defmessage-handler USER print primary  ())
(defmessage-handler USER direct-modify primary
  (?slot-override-expressions))
(defmessage-handler USER message-modify primary
  (?slot-override-expressions)
(defmessage-handler USER direct-duplicate primary
  (?new-instance-name ?slot-override-expressions))
(defmessage-handler USER message-duplicate primary
  (?new-instance-name ?slot-override-expressions)
(defmessage-handler USER create primary  ())
```

Пример работы с классами

Приведем пример работы с классами. Структурная схема классов имеет следующий вид (рис. 2).

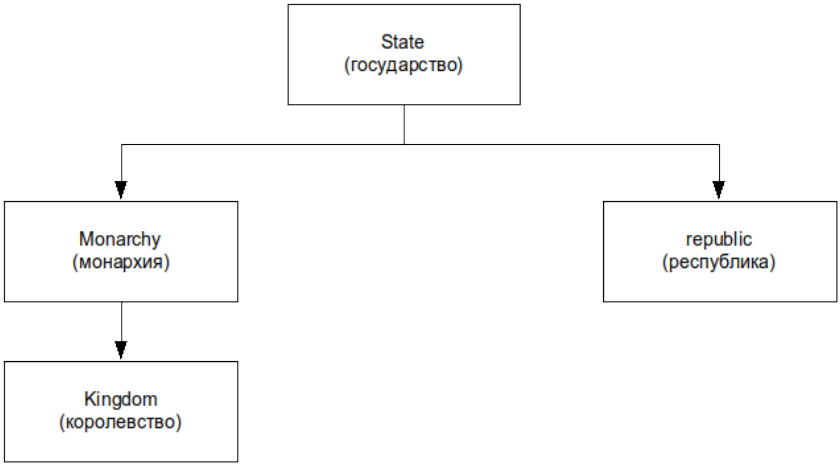

Рис.2 - Структурная схема классов

Исходный код:

```
;класс государство
(defclass state  (is-a USER)
   (role abstract)
   (slot name-1  (type STRING))
   (slot area   (type INTEGER)   (default 0))
   (slot population  (type INTEGER)   (default
   0))
)



;класс республика
(defclass republic  (is-a state)
   (role concrete)
   (slot president  (type STRING))
)
;класс монархия
(defclass monarchy (is-a state)
   (slot type-of-monarchy (type SYMBOL))
)
;класс королевство
(defclass kingdom (is-a monarchy)
```

```
   (role concrete)
   (slot king  (type STRING))
   (slot prime-minister  (type STRING))
)
;сообщение выводит на экран имя президента
(defmessage-handler republic print-president  ()
(printout t ?selfpresident crlf)
)
```

Результат работы программы имеет следующий вид.

Исходный код:

```
CLIPS> (load 3.txt)
Defining defclass: example
Defining defclass:  state
Defining defclass:  republic
Defining defclass: monarchy
Defining defclass:  kingdom
Defining defmessage-handler:  republic
   Handler print-president primary defined.
TRUE
CLIPS> (make-instance republicl of republic
(name-1 "France")
(area 2000)
(population 1000)
        (president "Oland"))
[republicl]
CLIPS> (send [republicl] print-president)
Oland
CLIPS>
```

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

В среде CLIPS создать абстрактный класс согласно варианту задания.

На основе абстрактного класса создать 2-3 конкретных класса. При создании классов необходимо задать необходимые обработчики сообщений, а также использовать описанные в теоретической части грани слотов.

Создать по 2-3 объекта каждого класса.

Продемонстрировать изменение и удаление объектов.

Создать 2-3 правила, использующих созданные объекты.

ТРЕБОВАНИЯ К РЕАЛИЗАЦИИ

Объекты (сущности) выбираются в соответствии с вариантом задания, который назначается преподавателем.

Все факты должны быть сохранены в файл посредством соответствующих команд *CLIPS*.

ВАРИАНТЫ ЗАДАНИЙ

1. ЭВМ.
2. Геометрические фигуры в пространстве.
3. Животные.
4. Автомобили.
5. Корабли.
6. Компьютерные игры.
7. Языки программирования.
8. Небесные тела.
9. Оружие.
10. Растения.
11. Строения.
12. Населенные пункты.
13. Телефоны.
14. Фирмы.
15. Мебель.

16. Одежда.
17. Водоемы.
18. Носители информации.
19. Строительные материалы.
20. Кондитерские изделия.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Перечислите системные классы. Поясните их назначение.
2. Укажите основные элементы синтаксиса создания класса.
3. Чем отличается простое наследование от множественного?
4. Для чего необходим список предшествующих классов?
5. Как выбирается список предшествующих классов при множественном наследовании?
6. Для чего необходимы абстрактные и конкретные классы? Укажите их отличия друг от друга.
7. Перечислите функции активных и неактивных классов.
8. Какую функцию выполняют грани?
9. Как задать место хранения экземпляра атрибута?
10. Укажите синтаксис задания правила доступа к атрибуту.
11. Как задать видимость атрибута?
12. Приведите пример автоматического создания явного обработчик сообщений для чтения и/или записи.
13. Приведите пример создания экземпляра класса.
14. Перечислите действия для управления объектами.
15. На какие группы делятся обработчики сообщений?

ФОРМА ОТЧЕТА ПО ЛАБОРАТОРНОЙ РАБОТЕ

На выполнение лабораторной работы отводится 2 занятия (4 академических часа: 3 часа на выполнение и сдачу лабораторной работы и 1 часа на подготовку отчета).

Номер варианта студенту выдается преподавателем.

Отчет на защиту предоставляется в печатном виде.

Структура отчета (на отдельном листе(-ах)): титульный лист, формулировка задания (вариант), этапы выполнения работы (со скриншотами), диаграмма классов, результаты выполнения работы (скриншоты и содержимое файлов), выводы.

ОСНОВНАЯ ЛИТЕРАТУРА

1. Малышева Е.Н. Экспертные системы [Электронный ресурс]: учебное пособие по специальности 080801 «Прикладная информатика (в информационной сфере)»/ Малышева Е.Н.— Электрон. текстовые данные.— Кемерово: Кемеровский государственный институт культуры, 2010.— 86 с.— Режим доступа: http://www.iprbookshop.ru/22126.— ЭБС «IPRbooks»
2. Павлов С.Н. Системы искусственного интеллекта. Часть 1 [Электронный ресурс]: учебное пособие/ Павлов С.Н.— Электрон. текстовые данные.— Томск: Томский государственный университет систем управления и радиоэлектроники, Эль Контент, 2011.— 176 с.— Режим доступа: http://www.iprbookshop.ru/13974.— ЭБС «IPRbooks»
3. Павлов С.Н. Системы искусственного интеллекта. Часть 2 [Электронный ресурс]: учебное пособие/ Павлов С.Н.— Электрон. текстовые данные.— Томск: Томский государственный университет систем управления и радиоэлектроники, Эль Контент, 2011.— 194 с.— Режим доступа: http://www.iprbookshop.ru/13975.— ЭБС «IPRbooks»
4. Чернышов, В.Н. Системный анализ и моделирование при разработке экспертных систем : учебное пособие / В.Н. Чернышов, А.В. Чернышов ; Министерство образования и науки Российской Федерации, Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования «Тамбовский государственный технический университет». - Тамбов : Издательство ФГБОУ ВПО «ТГТУ», 2012. - 128 с. : ил. - Библиогр. в кн. ; То же [Электронный ресурс]. - URL: //biblioclub.ru/index.php?page=book&id=277638 (22.02.2017).

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

5. Воронов, А.Е. Технология использования экспертных систем / А.Е. Воронов. - М. : Лаборатория книги, 2011. - 109 с. : ил. - ISBN

978-5-504-00525-6 ; То же [Электронный ресурс]. - URL: //biblioclub.ru/index.php?page=book&id=142527 (22.02.2017).
6. Трофимов, В.Б. Интеллектуальные автоматизированные системы управления технологическими объектами : учебно-практическое пособие / В.Б. Трофимов, С.М. Кулаков. - Москва-Вологда : Инфра-Инженерия, 2016. - 232 с. : ил., табл., схем. - Библиогр. в кн.. - ISBN 978-5-9729-0135-7 ; То же [Электронный ресурс]. - URL: //biblioclub.ru/index.php?page=book&id=444175 (22.02.2017).
7. Интеллектуальные и информационные системы в медицине: мониторинг и поддержка принятия решений : сборник статей / . - М. ; Берлин : Директ-Медиа, 2016. - 529 с. : ил., схем., табл. - Библиогр. в кн. - ISBN 978-5-4475-7150-4 ; То же [Электронный ресурс]. - URL: //biblioclub.ru/index.php?page=book&id=434736 (22.02.2017).
8. Джарратано Дж., Райли Г. Экспертные системы. Принципы разработки и программирование, 4-е издание.: Пер. с англ. – М.: ООО «И. Д. Вильямс», 2007. – 1152 с.: ил. – Парал. тит. англ.

Электронные ресурсы:

9. https://ru.wikipedia.org/wiki/CLIPS - CLIPS — Википедия
10. http://clipsrules.sourceforge.net/ - A Tool for Building Expert Systems (англ.)
11. http://clipsrules.sourceforge.net/WhatIsCLIPS.html - What is CLIPS? (англ.)