

ЛАБОРАТОРНАЯ РАБОТА №2

- ТЕМА:** Знакомство со средой CLIPS 6.2. Работа с правилами в среде CLIPS.
- ЦЕЛЬ:** Научиться общим приемам работы в среде CLIPS. Научиться использовать имеющиеся возможности CLIPS для работы с правилами. Изучить представление правил и их внутренние алгоритмы обработки, стратегии разрешения конфликтов, синтаксис левой части (LHS). Научиться использовать команды и функции для работы с правилами.

ПЛАН ЗАНЯТИЯ:

1. Создание правил.
2. Синтаксис LHS правила.
3. Команды и функции для работы с правилами.
4. Контрольные вопросы

1. СОЗДАНИЕ ПРАВИЛ

CLIPS поддерживает процедурную и эвристическую парадигму представления знаний.

Для представления эвристик или так называемых "эмпирических правил", которые определяют набор действий, выполняемых при возникновении некоторой ситуации, в CLIPS используют *правила*, которые определяются разработчиком экспертной системы. Правила состоят из *предпосылок* и *следствия*. Предпосылки называются также **ЕСЛИ-частью правила**, *левой частью правила* или **LHS** правила (left-hand side of rule). Следствие называется **ТО-частью правила**, *правой частью правила* или **RHS** правила (right-hand side of rule).

Предпосылки правила представляют собой набор условий (или условных элементов), которые должны удовлетвориться, для того чтобы правило выполнилось.

Следствие правила представляется набором некоторых действий, которые необходимо выполнить, в случае если правило применимо к текущей ситуации. Таким образом, действия, заданные вследствие правила, выполняются по команде механизма логического вывода, если все предпосылки правила удовлетворены. В случае если в данный момент применимо более одного правила, механизм логического вывода

использует так называемую *стратегию разрешения конфликтов* (conflict resolution strategy), которая определяет, какое именно правило будет выполнено.

Чтобы лучше понять сущность правил в CLIPS, их можно представить в виде оператора *if-then*, используемого в процедурных языках программирования.

1.1. Конструктор *defrule*

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

Определение 1. Синтаксис конструктора *defrule*

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

Имя правила должно быть значением типа *symbol*. В качестве имени правила нельзя использовать зарезервированные слова CLIPS. Повторное определение существующего правила приводит к удалению правила с тем же именем, даже если новое определение содержит ошибки.

Комментарии являются необязательными, и, как правило, описывают назначения правила. Комментарии необходимо заключать в кавычки.

Определение правила может содержать объявление свойств правила, которое следует непосредственно после имени правила и комментариев.

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

Левая часть правила задается набором условных элементов, который обычно состоит из условий, примененных к некоторым образцам. Заданный набор образцов используется системой для сопоставления с имеющимися фактами и объектами. Все условия в левой части правила объединяются с помощью неявного логического оператора *and*.

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

Если в левой части правила не указан ни один условный элемент, CLIPS автоматически подставляет условие-образец *initial-fact* или *initial-object*. Таким образом, правило активизируется всякий раз при появлении в базе знаний факта *initial-fact* или объекта *initial-object*. Если в правой части правила не определено ни одно действие, правило может быть активировано и выполнено, но при этом ничего не произойдет.

В качестве демонстрации применения правил, напомним простейшую CLIPS-программу, которая по традиции здоровается со всем миром, сразу после своего рождения. На языке CLIPS такая программа будет выглядеть следующим образом:

Пример 1. Программа "Hello-World!"

```
(clear)
(defrule
  Hello-World
  "My First CLIPS Rule"
  =>
  (printout t crlf crlf)
  (printout t ***** crlf)
  (printout t "* HELLO WORLD!!! *" crlf)
  (printout t ***** crlf)
  (printout t crlf crlf)
)
(reset)
(run)
```

Разберем все действия данной программы на языке CLIPS и то, каким образом они выполняются.

Функция **clear** полностью очищает систему, т. е. удаляет все правила, факты и прочие объекты базы знаний CLIPS, добавленные конструкторами, приводит систему в начальное состояние, необходимое для каждой новой программы.

Затем, с помощью конструктора **defrule** в систему добавляется новое правило с именем *Hello-World* и соответствующими комментариями. Левая

часть правила в данном случае отсутствует, поэтому CLIPS автоматически формирует предпосылки, состоящие из единственного условного выражения (*initial-fact*). Это выражение является образцом простейшего типа. При запуске программы на выполнение механизм логического вывода CLIPS будет искать в списке фактов факт (*initial-fact*) и если он там будет найден — активизирует правило. Правая часть нашего правила состоит из нескольких вызовов функции *printout*, которая выводит текстовое выражение в один из потоков вывода. Параметр *t* задает стандартный поток вывода — экран. Он аналогичен, например, стандартному потоку *cout* в C++. Выражение *crlf* служит для перехода на новую строку.

Функция *reset*, как уже упоминалось ранее, очищает список фактов и заносит в него факт (*initial-fact*), что очень важно для нормального функционирования нашей программы. И, наконец, функция *run* запускает механизм логического вывода и приводит нашу программу в движение. Если описанные выше действия были выполнены правильно, то вы должны увидеть результат, аналогичный приведенному на рис. 1 — фразу "HELLO WORLD!" в красивой рамочке из звездочек:



```
Dialog Window
CLIPS> (clear)
CLIPS>
(defrule
  Hello-World
  "My First CLIPS Rule"
  =>
  (printout t crlf crlf)
  (printout t ***** crlf)
  (printout t "**  HELLO  WORLD!  *" crlf)
  (printout t ***** crlf)
  (printout t crlf crlf)
)
CLIPS> (reset)
==> f-0      (initial-fact)
CLIPS> (run)

*****
*  HELLO  WORLD!  *
*****

CLIPS>
```

Рис 1. Результат работы программы "Hello-World!"

Чтобы запустить нашу программу на выполнение еще раз, достаточно вызвать функции *reset* и *run*. Эти функции можно вводить с клавиатуры, кроме того, они доступны в меню **Execution** и имеют "горячие" клавиши <Ctrl>+<E> и <Ctrl>+<R> соответственно.

CLIPS поддерживает ряд функций, команд и визуальных средств, необходимых для эффективной работы с правилами.

Рассмотрим визуальный инструмент, доступный пользователям Windows-версии среды CLIPS — **Defrule Manager** (Менеджер правил). Для запуска менеджера правил в меню **Browse** выберите пункт **Defrule Manager**. Внешний вид этого инструмента показан на рис. 2.

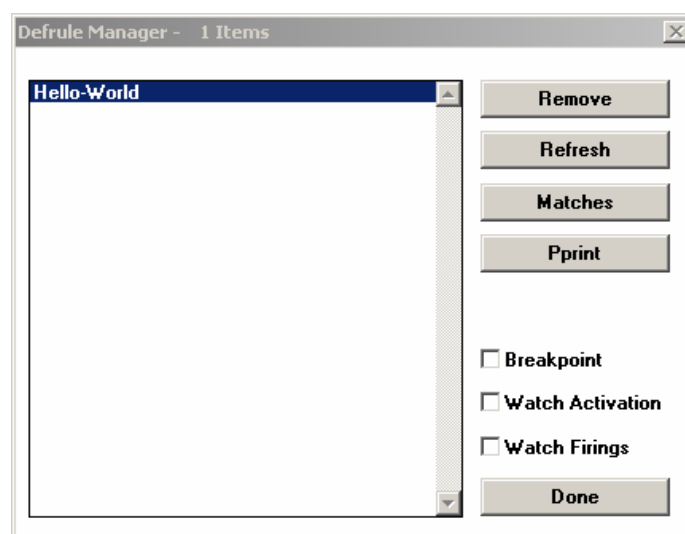


Рис 2. Окно **Defrule Manager**

Менеджер отображает список правил, присутствующих в системе в данный момент, и позволяет выполнять над ними ряд операций. Например, с помощью кнопки **Remove** можно удалить выбранное правило из системы, а с помощью **Pprint** вывести в окне CLIPS определение выделенного правила вместе с введенными комментариями. Общее количество правил отображается в заголовке окна менеджера — **Defrule Manager — 1 Items**.

1.2. Свойства правил

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

Определение 2. Синтаксис свойств правил

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

Свойство правила *salience* позволяет пользователю назначать приоритет для своих правил. Объявляемый приоритет должен быть выражением, имеющим целочисленное значение из диапазона от -10 000 до +10 000. Выражение, представляющее приоритет правила, может использовать глобальные переменные и функции. Однако старайтесь не указывать в этом выражении функций, имеющих побочное действие. В случае если приоритет правила явно не задан, ему присваивается значение по умолчанию — 0.

Значение приоритета может быть вычислено в одном из трех случаев:

- ☐ при добавлении нового правила,
- ☐ при активации правила
- ☐ на каждом шаге основного цикла выполнения правил.

Два последних варианта называются *динамическим приоритетом* (dynamic salience). По умолчанию значение приоритета вычисляется только во время добавления правила. Для изменения этой установки можно использовать команду *set-salience-evaluation*.

Кроме того, пользователи Windows-версии среды CLIPS могут изменить эту настройку с помощью диалогового окна **Execution Options**. Для этого выберите пункт **Options** в меню **Execution**, в появившемся диалоговом окне укажите необходимый режим вычисления приоритета с помощью раскрывающегося списка **Salience Evaluation**, как показано на рис. 3.

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

Свойство *auto-focus* позволяет автоматически выполняться команде *focus* при каждой активации правила. Если свойство *auto-focus* установлено в значение TRUE, то команда *focus* в модуле, в котором определено данное правило, автоматически выполняется всякий раз при активации правила. Если свойству *auto-focus* присвоено значение FALSE,

то при активации правила не происходит никаких действий. По умолчанию это свойство установлено в FALSE.

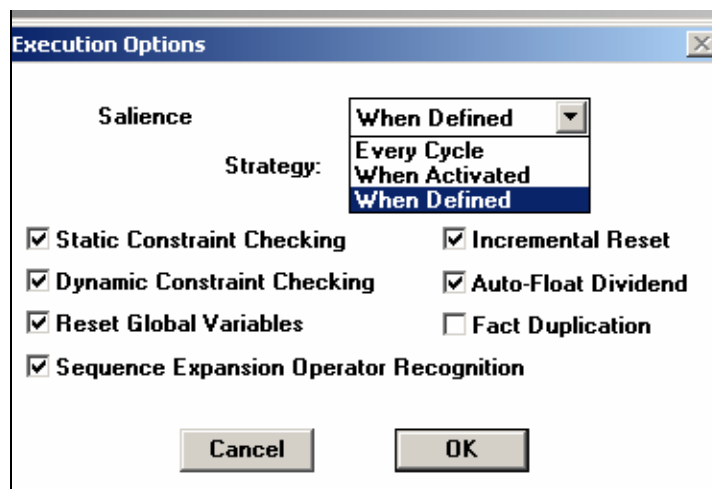


Рис 3. Установка способа вычисления приоритетов правил

1.3. Стратегии разрешения конфликтов

CLIPS поддерживает семь различных стратегий разрешения конфликтов:

- ☐ стратегия глубины (depth strategy),
- ☐ стратегия ширины (breadth strategy),
- ☐ стратегия упрощения (simplicity strategy),
- ☐ стратегия усложнения (complexity strategy),
- ☐ LEX (LEX strategy),
- ☐ MEA (MEA strategy),
- ☐ случайная стратегия (random strategy).

По умолчанию в CLIPS установлена стратегия глубины. Текущая стратегия может быть установлена командой **set-strategy** (которая переупорядочит текущий план решения задачи, базируясь на новой стратегии). Кроме того, пользователи Windows-версии среды CLIPS могут указать необходимую стратегию поиска с помощью диалогового окна **Execution Options** (см. рис.3). Для этого выберите пункт **Options** в меню **Execution**, в появившемся диалоговом окне выберите необходимую стратегию с помощью раскрывающегося списка **Strategy**.

2. СИНТАКСИС LHS ПРАВИЛА

Рассмотрим синтаксис, используемый в левой части правил. Левая часть правил содержит список *условных элементов* (conditional elements или CEs), которые должны удовлетворяться, для того чтобы правило было помещено в план решения задачи. Существует восемь типов условных элементов, используемых в левой части правил: *CEs-образцы*, *test CEs*, *and CEs*, *or CEs*, *not CEs*, *exists CEs*, *forall CEs*, *logical CEs*.

Образцы — наиболее часто используемый условный элемент. Он содержит ограничения, которые служат для определения, удовлетворяет ли какой-нибудь элемент данных (факт или объект) образцу.

Условие *test* используется для оценки выражения, как части процесса сопоставления образов.

Условие *and* применяется для определения группы условий, каждое из которой должно быть удовлетворено.

Условие *or* — для определения одного условия из некоторой группы, которое должно быть удовлетворено.

Условие *not* — для определения условия, которое не должно быть удовлетворено.

Условие *exists* — для проверки наличия, по крайней мере одного, совпадения факта (или объекта) с некоторым заданным образцом.

Условие *logical* позволяет выполнить добавление фактов и создание объектов в правой части правила, связанных с фактами и объектами, совпавшими с заданным образцом в левой части правила (поддержка достоверности фактов в базе знаний).

Синтаксис условного элемента можно формализовать следующим образом:

Определение 3. Синтаксис условного элемента

```
<условный-элемент> ::= <pattern-CE> |  
                        <assigned-pattern-CE> |  
                        <not-CE> |  
                        <and-CE> |  
                        <or-CE> |  
                        <logical-CE> |  
                        <test-CE> |  
                        <exists-CE> |  
                        <forall-CE>
```


2.1. Образец (*pattern CE*)

Этот условный элемент состоит из списка *ограничений полей*, *групповых символов* (wildcards) и *переменных*, которые используются для поиска множества фактов или объектов, которые соответствуют заданному образцу. Таким образом, образец как бы определяет маску, которой должны соответствовать данные. Такой условный элемент удовлетворяется любым фактом или объектом, соответствующим заданным ограничениям.

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

- ☐ объединяющие ограничения,
- ☐ предикатные ограничения и ограничения,
- ☐ возвращающие значения.

Групповые символы используются при сопоставлении образцов в ситуации, когда простое поле или группа полей могут принимать любые значения.

Переменные применяются для хранения значения поля, которое может быть впоследствии использовано в левой части правила для другого условного элемента или в правой части, как аргумент действия.

Первое поле любого образца обязательно должно быть значением типа *symbol* и не может принимать значения других типов. CLIPS использует первое поле для определения: является ли данный образец упорядоченным фактом, шаблоном или объектом. Ключевое слово ***object*** зарезервировано для создания образцов, предназначенных для сопоставления с объектами. Любое другое значение типа *symbol* должно соответствовать имени шаблона, созданного с помощью конструктора ***deftemplate*** или неявно созданного шаблона. Для задания имен слотов также должны использоваться значения типа *symbol*.

В слотах простых полей образцов, предназначенных для объектов и шаблонов, может содержаться только одно ограничение поля, и не могут присутствовать групповые символы или переменные. В составных слотах может содержаться любое количество ограничений поля.

Пример 2. Необходимые для дальнейшей работы шаблоны и факты

```

(deffacts      data-facts
  (data 1.0 blue "red")
  (data 1 blue)
  (data 1 blue red)
  (data 1 blue RED)
  (data 1 blue red 6.9))

(deftemplate   person
  (slot name)
  (slot age)
  (multislot friends))

(deffacts      people
  (person (name Joe) (age 20))
  (person (name Bob) (age 20))
  (person (name Joe) (age 34))
  (person (name Sue) (age 34))
  (person (name Sue) (age 20))
)

```

2.1.1. Символьные ограничения

Основные ограничения, использующиеся в образцах, — это ограничения, определяющие точное соответствие между полями факта и образцом. Эти ограничения называются *символьными*. Символьное ограничение полностью состоит из констант, таких как вещественные и целые числа, значения типа *symbol*, строки или имена объектов. Они не могут содержать групповых символов или переменных. Все символьные ограничения при сопоставлении образцов должны точно совпадать по всем указанным полям, иначе факт не будет считаться подошедшим данному образцу.

Условный элемент, представляющий собой образец для неупорядоченного факта, в котором присутствуют только символьные ограничения, имеет следующий синтаксис:

Определение 4. Синтаксис символьных ограничений для неупорядоченного факта

(<ограничение-1> ... <ограничение-n>)

Условный элемент, представляющий собой образец для шаблона, в котором присутствуют только символьные ограничения, выглядит так:

Определение 5. Синтаксис символьных ограничений для шаблона

(<имя-шаблона > (<имя-слота-1> <ограничение-1>)
(<имя-слота-n> <ограничение-n>))

Рассмотрим пример правил, использующих в качестве образца — образец фактов (как упорядоченных, так и шаблонов) с символьными ограничениями. Для работы этого примера необходимо ввести в CLIPS все конструкторы предопределенных фактов и шаблонов. После этого следует выполнить команду **reset** для инициализации списка фактов. Для проверки правильности выполненных операций откройте окно **Facts**. Если все описанные действия были выполнены без ошибок, вы должны увидеть результат, приведенный на рис.4. Для нормальной работы примеров не забывайте выполнять команду **reset** перед каждым запуском правил.

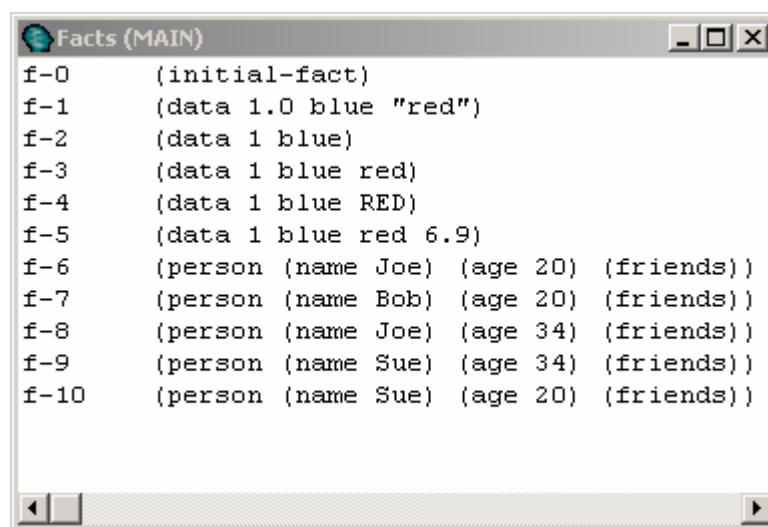


Рис 4. Список необходимых фактов

Введите в CLIPS определения следующих правил:

Пример 3. Правила с символьными ограничениями

```

(defrule Find-data
  (data 1 blue red)
=>
  (printout t crlf "Found data (data 1 blue red)"
crlf))

(defrule Find-Bob-20
  (person (name Bob) (age 20))
=>
  (printout t crlf "Found Bob-20 (person (name Bob)
                    (age 20))" crlf))

(defrule Find-Bob-30
  
```

```

(person (name Bob) (age 30))
=>
(printout t crlf "Found Bob-30 (person (name Bob)
               (age 30))" crlf))

```



Рис 5. Выполнение правил с символьными ограничениями

Выполните команды **reset** и **run**. Вы должны получить результат, приведенный на рис. 5.

Как мы видим, были активированы и выполнены два правила: *Find-data* и *Find-Bob-20*. Это произошло потому, что образцы, заданные в левой части этих правил, нашли в списке фактов данные, полностью соответствующие заданным символьным ограничениям.

2.1.2. Групповые символы для простых и составных полей

В CLIPS имеется два различных групповых символа, которые используются для сопоставления полей в образцах. CLIPS интерпретирует эти групповые символы как место для подстановки некоторых частей данных, удовлетворяющих образцам. Групповой символ для простого поля записывается с помощью знака **?**, который соответствует одному любому значению, сохраненному в заданном поле. Групповой символ составного

поля записывается с помощью знака $\$?$ и соответствует, возможно, пустой последовательности полей, сохраненной в составном поле. Групповые символы для простых и составных полей могут комбинироваться в любой последовательности. Нельзя использовать групповой символ составного поля для простых полей. По умолчанию не заданный в образце простой слот шаблона или объекта сопоставляется с неявно заданным групповым символом для простого поля. Аналогично не заданный в образце составной слот сопоставляется с неявно заданным групповым символом для составного поля.

Условный элемент, представляющий собой образец для неупорядоченного факта, в котором присутствуют только символьные ограничения и групповые символы, будет иметь следующий вид:

Определение 6. Синтаксис ограничений для неупорядоченного факта

```
(<ограничение-1> ... <ограничение-n>)
<ограничение> ::= <символьное-ограничение> > | ? | $?
```

Соответственно для шаблона образец примет вид:

Определение 7. Синтаксис ограничений для шаблона

```
(<имя-шаблона> (<имя-слота-1> <ограничение-1>)
...
(<имя-слота-n> <ограничение-n>)
)
```

В качестве примера можно привести следующее правило:

Пример 4. Правило Find-data

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

В нашем списке фактов присутствуют два факта, подходящие заданному шаблону и способные активировать данное правило:

Пример 5. Факты, активирующие правило Find-data

```
(data 1 blue red)
(data 1 blue red 6.9)
```

Рассмотрим еще одно правило:

Пример 6. Правило match-all-persons

```
(defrule match-all-persons
  (person)
  =>
)
```

Поскольку *person* является шаблоном, а в образце данного правила не определен ни один слот шаблона, CLIPS автоматически поставит в соответствие каждому простому слоту групповой символ для простого поля, а составному слоту — символ для составного. Таким образом, правило преобразуется в следующее:

Пример 7. Преобразованное правило match-all-persons

```
(defrule match-all-persons
  (person
    (name ?)
    (age ?)
    (friends $?))
  =>
)
```

Это правило будут активировать все факты шаблона *person*.

Групповые символы для составного поля можно комбинировать с символьными ограничениями, что приводит к получению более мощных возможностей сопоставления образцов. Образец, который сопоставляется со всеми фактами, имеющими значение *YELLOW* в любом поле (включая первый), может быть записан так:

Пример 8. Образец со значением YELLOW в любом поле

```
(data $? YELLOW $?)
```

Вот несколько фактов, соответствующих этому образцу:

Пример 4. Факты со значением YELLOW в любом поле

```
(data YELLOW blue red green)
(data YELLOW red)
(data red YELLOW)
(data YELLOW)
(data YELLOW data YELLOW)
```

Последний факт будет соответствовать образцу дважды, т.к. *YELLOW* присутствует в нем дважды. Использование группового символа для составного поля позволяет создавать гораздо более общие образцы, чем те, которые можно сформировать с помощью групповых символов для простого поля. Однако подобная общность приводит к тому, что процесс сопоставления образцов, использующих групповые символы, иногда занимает гораздо больше времени, чем аналогичный процесс с образцами, использующими только групповые символы для простых полей.

2.1.3. Переменные, связанные с простыми и составными полями

Групповые символы заменяют любые поля образца и могут принимать какие угодно значения этих полей. Значение поля может быть связано с переменными для последующего сопоставления, отображения и других действий. Это выполняется с помощью применения имени переменной следующим непосредственно после группового символа.

Таким образом, синтаксис ограничения, применяемого в образце, примет следующий вид:

Определение 8. Синтаксис ограничений

<ограничение>	$::= \text{<символьное-ограничение>} \mid$ $? \mid$ $\$? \mid$ $\text{<переменная-простого-поля>} \mid$ $\text{<переменная-составного-поля>}$
<переменная~простого-поля>	$::= ?\text{<имя-переменной>}$
<переменная-составного-поля>	$::= \$?\text{<имя-переменной>}$

Имя переменной должно быть значением типа *symbol* и обязательно начинаться с буквы. В имени переменной не разрешается использовать кавычки, т. е. строка не может использоваться как имя переменной или ее часть.

Правила сопоставления образцов при использовании переменных и ограничениях образца аналогичны правилам, использующимся для групповых символов. В момент первого появления имени переменной она ведет себя так же, как и соответствующий групповой символ. В этот момент CLIPS (называет значения поля с заданной переменной. Эта связь будет действовать только в рамках правила, в котором она возникла. Каждое правило имеет свой собственный список имен переменных со значениями, связанными с ними, эти переменные локальны для правил.

Связанные переменные могут быть использованы во внешних функциях. Символ \$ имеет особое значение в левой части правил — этот оператор отображает, что некоторая, возможно пустая, последовательность полей требует сопоставления. В правой части правила символ \$ ставится перед переменной для обозначения того, что перед использованием переменной в качестве аргумента функции необходимо раскрыть последовательность полей, содержащихся в переменной. Таким образом, при использовании переменных в качестве параметров функций (как в

левой, так и правой части правил) перед именем переменной, содержащей значение составного поля, не должен стоять символ \$ (за исключением случаев, когда требуется раскрыть последовательность полей). При использовании переменной, содержащей значение составного поля, в других случаях, перед ее именем должен стоять символ \$. Нельзя применять переменную составного поля при операциях с простым полем образца шаблона или объекта.

И качестве примера введите в среду CLIPS следующее правило:

Пример 10. Правило Find-data

```
(defrule Find-data
  (data ? blue ?x $?y)
  =>
  (printout t "Found data (data ? blue " ?x " "
    ?y ") "
    crlf)
)
```

Выполните команды *reset* и *run*. Если правило было введено в систему без ошибок, то на экране появится следующий результат:

Рис 6. Результат работы правила **Find-data**

Образцу, заданному в правиле, удовлетворяют четыре факта с индексами 1, 3, 4, 5. В результате активации правило выводит на экран свойства фактов, активировавших правило. Значение переменной, содержащей значение из составного поля, выводится в скобках. Кроме первого случая (факта с индексом 5), переменная содержит пустое

значение. Переменную составного поля не обязательно использовать в качестве последнего ограничения.

Рассмотрим следующее правило:

Пример 11. Модифицированное правило Find-data

```
(defrule Find-data
  (data ?x $?y ?z)
=>
  (printout t "x=" ?x " y=" ?y " z=" ?z crlf))
```

Заданному образцу удовлетворяют все факты *data*, но обратите внимание, каким образом связываются значения с переменной *y* в разных случаях:

Пример 12. Результат работы модифицированного правила Find-data

x=1.0	y=(blue)	z=red
x=1	y=()	z=blue
x=1	y=(blue)	z=red
x=1	y=(blue)	z=RED
x=1	y=(blue red)	z=6.9

После того как произошло связывание переменной со значением, все ссылки на эту переменную возвращают значение, с которым переменная была связана. Это действительно как для переменных, связанных с составными полями, так и для переменных, связанных с простыми полями. Кроме того, допустимы ссылки между образцами в одном правиле.

Пример 13. Правило Find-2-Coeval-Person

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

Приведенное выше правило *Find-2-Coeval-Person* выведет на экран всевозможные пары имен людей (все перестановки) одинакового возраста. Как научить это правило не выводить эквивалентные по смыслу или бессмысленные пары одинаковых имен (*Bob-Bob*), мы рассмотрим в следующих лабораторных работах.

2.1.4. Связывающие ограничения

CLIPS предоставляет 3 связывающих ограничения, предназначенных для объединения отдельных ограничений и переменных в единое целое:

- & (логическое И),
- | (логическое ИЛИ),
- ~ (логическое НЕ).

Ограничение &, удовлетворяется, если два соседних ограничения удовлетворяются.

Ограничение | удовлетворяется, если любое из двух соседних ограничений удовлетворяется.

Ограничение ~ удовлетворяется, если следующее за ним ограничение не удовлетворяется.

Связывающие ограничения могут комбинироваться почти произвольным образом и в любом количестве. Ограничение ~ имеет наивысший приоритет, далее следуют & и |. В случае одинакового приоритета ограничение вычисляется слева направо. Существует одно исключение из правил приоритета, которое применяется при связывании переменных. Если первое ограничение — это переменная и за ней следует &, то переменная является отдельным ограничением. Ограничение ?X&RED|BLUE вычисляется как ?X& (RED|BLUE), в то время как по правилам приоритета оно должно было вычисляться как (?X&RED)|BLUE.

Связанные ограничения имеют следующий синтаксис:

Определение 9. Синтаксис связывающих ограничений

<элемент-1> & <элемент-2> ... & <элемент -n>
<элемент-1> | <элемент-2> ... | <элемент -n>
~ <элемент>

Здесь <элемент> должен быть переменной, связанной с простым или составным полем, ограничением или связанным ограничением.

Таким образом, определения ограничений, приведенные ранее можно расширить так:

Определение 10. Синтаксис ограничений

<ограничение> ::= ? |
\$? |
<связанное-ограничение>
<связанное-ограничение> ::= <простое-ограничение> |
<простое-ограничение>&<связанное-ограничение>|

```

<простое-ограничение> | <связанное-ограничение> |
<простое-ограничение> ::= <элемент> | ~ <элемент>
<элемент> ::= <константа> |
<простая-переменная> |
<составная-переменная>

```

Ограничение **&** обычно служит только для объединения с другими ограничениями или связывания переменных. Заметьте, что связывающие ограничения могут использовать связанные переменные и в то же время сами производить связывание переменной со значением некоторого поля. Если имя переменной встретилось в первый раз, то для ограничения будут использоваться остальные члены условного элемента, а переменная будет связана с соответствующим значением поля. Если переменная уже была связана, то ее значение работает как дополнительное ограничение для данного поля.

В качестве примера приведем улучшенный вариант правила *Find-2-Coeval-Person* см. пример 18.

Пример 14. Улучшенное правило Find-2-Coeval-Person

```

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

```

Ограничение **?Y&~?X** запрещает выводить бессмысленные пары одинаковых имен (*Bob-Bob*). Однако данное правило все еще выводит эквивалентные по смыслу пары имен (например, *Bob-Sue* и *Sue-Bob*).

2.1.5. Предикатные ограничения

Иногда необходимо ограничить поле, основываясь на истинности некоторого логического выражения. CLIPS позволяет использовать предикатные ограничения. Предикатные ограничения позволяют вызывать предикатные функции (функции, которые возвращают значение FALSE при не соответствии условиям и не-FALSE, если значение удовлетворяет условиям) в течение процесса сопоставления образцов. Если предикатная функция возвращает значение не-FALSE, ограничение удовлетворяется. Если предикатная функция возвращает значение FALSE, то ограничение не удовлетворяется. Предикатные ограничения записываются с помощью

двоеточия и следующего за ним вызова соответствующей предикатной функции. Обычно предикатные ограничения используются совместно со связывающими ограничениями и при связывании переменных (т. е. если вы имеете переменную, которую нужно связать с некоторым полем и хотите одновременно ее протестировать, объедините ее с предикатным ограничением).

Определение 11. Синтаксис предикатного ограничения

:<вызов-функции>

Таким образом, определение понятия "элемент", приведенное ранее (см. определение 10), можно расширить следующим образом:

Определение 12. Синтаксис понятия "элемент"

**<элемент> ::= <константа> |
<простая-переменная> |
<составная-переменная> |
:<вызов-функции>**

CLIPS предоставляет несколько готовых предикатных функций. Кроме этого, пользователь также может создавать свои собственные предикатные функции.

Пример 15. Еще один вариант правила Find-data

```
(defrule Find-data
  (data ?x&:(floatp ?x)&:(> ?x 0) $?y
   ?z&:(stringp ?z))
  =>
  (printout t "x=" ?x " y=" ?y " z=" ?z crlf)
)
```

Выше приведен еще один вариант правила *Find-data*. В данном случае ищется факт неявно созданного шаблона *data*, первое поле которого — вещественное число больше нуля, а последнее — строка. В нашем списке фактов такому правилу удовлетворяет только факт с индексом 1 — *(data 1.0 blue "red")*.

2.1.6. Ограничения, возвращающие значения

В ограничениях возможно использование значений, возвращенных некоторыми функциями (в том числе и внешними). Вызов функции записывается с помощью знака = и указанной за ним функцией.

Замечание: Функция сравнения также использует знак = Разница между ними может быть определена по контексту.

Возвращаемое значение должно быть одним из простых типов данных CLIPS. Это значение, возвращенное функцией, объединяется с образцом так, как если бы оно было символьным ограничением. Заметьте, что функция вычисляется при каждом сопоставлении образцов, а не один раз при определении правила.

Определение 13. Синтаксис ограничения, возвращающего значение

=<вызов-функции>

Приведенные определения понятия "элемент", примут такой вид:

Определение 14. Синтаксис понятия "элемент"

**<элемент> ::= <константа> |
 <простая-переменная> |
 <составная-переменная> |
 :<вызов-функции>
 =<вызов-функции>**

Следующее правило выводит на экран такие факты *data*, в которых значение второго поля в два раза больше, чем значение первого. В нашем случае это факты (*data 1 2*) и (*data 2 4*).

Пример 16. Использование ограничения, возвращающего значение

```
(assert      (data 1 2)
              (data 2 3)
              (data 2 4)
)
(defrule     Find-data
  (data ?x ?y&(* 2 ?x))
=>
  (printout t "x=" ?x " y=" ?y crlf)
)
```

2.1.7. Сопоставление образцов с объектами

Во всех приведенных выше примерах образцы сопоставлялись с фактами из списка фактов. Кроме этого, образцы можно сопоставлять с экземплярами объектов — экземпляров, определенных пользователем классов на языке COOL. Такие образцы называются образцами объектов. Образцы могут сопоставляться с объектами, спецификация которых определена до создания образца и которые находятся в границах видимости текущего модуля. Любой класс, который имеет объекты,

соответствующие образцу, не может быть удален или изменен, пока не будет удален образец. Даже если правило удалено с помощью действий, выполняемых в собственной правой части, класс, связанный с образцом, не может быть изменен до тех пор, пока правая часть правила не закончит работу.

При создании или удалении объекта все образцы, подходящие этому объекту, обновляются. Однако в случае изменения слота объекта обновляются только те образцы, которые явно сопоставляются по этому слоту. Таким образом можно использовать логические зависимости для обработки изменений некоторых слотов.

Изменение неактивных слотов или объектов неактивных классов не оказывает никакого воздействия на правила.

Определение 15. Синтаксис образцов объектов

```
<образец объекта> ::= (object <атрибуты-ограничения>)  
<атрибуты-ограничения> ::= (is-a <ограничение>) |  
                             (name <ограничение>) |  
                             (slot <ограничение>)
```

Ограничение *is-a* (является) используется для определения ограничений класса, таких как "Является ли этот объект экземпляром заданного класса?". Ограничение *is-a* также определяет, является ли объект экземпляром класса, который является наследником класса, заданного в ограничении, в случае если это не будет явно запрещено образцом.

Ограничение *name* используется для определения конкретного объекта с заданным именем. Имя, задаваемое в данном ограничении, должно быть значением типа *instance-name*, а не значением типа *symbol*, как обычно. Ограничения для составных полей (такие как \$?) не могут использоваться с ограничениями *is-a* и *name*. Эти ограничения применяются в работе со слотами объектов так же, как и при работе со слотами шаблонов. Как и в случае образцов для шаблонов, имена слотов для образца объекта должны быть значениями типа *symbol*.

Пример 17. Использование образцов объектов

```
(defrule example-1  
  (object (is-a MyObj1 | MyObj2))  
  => )  
(defrule example-2  
  (object (is-a ?x)) (object (is-a ~?x))  
  => )  
(defrule example-3
```

```
(object (width ?x&: (> ?x 20)))
=> )
(defrule example-4
  (object (width ?x) (height ?x))
=> )
```

Первое правило удовлетворяет любой объект класса *Myobj1* или *MyObj2*. Второе правило активируется любой парой объектов, принадлежащей разным классам. Третье правило выполняется в случае, если будет найден объект активного класса, содержащий активный слот *width*, значение которого больше 20. Последний приведенный пример удовлетворяется любым объектом активного класса, содержащим активные слоты *width* и *height*, значения которых должны быть равны.

2.1.8. Адрес образца

Некоторые действия в правой части правил, такие как *retract* и *unmake-instance*, оперируют с фактами или объектами, участвующими в левой части. Для того чтобы определить, какой факт или объект будет изменяться, необходимо присвоить переменной адрес конкретного факта или объекта. Присваивание адресов происходит в левой части правила и полученное значение называется адресом образца (*pattern-address*).

Определение 16. Синтаксис адреса образца

<адрес-образца> ::= ?<имя-переменной> <- <образец>

Стрелка влево (<-) – необходимая часть синтаксиса. Переменная, связанная с адресом факта или объекта, может сравниваться с другой переменной или использоваться внешней функцией. Переменная, связанная с адресом факта или объекта, может быть также использована для последующего ограничения полей в образце условного выражения. Однако нельзя связывать переменную в условном выражении *not*.

В качестве примера приведем простое правило, которое удаляет все факты *data*.

Пример 18. Правило del-data-facts

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

На этом рассмотрение синтаксиса и способов использования условного элемента образец (*pattern CE*) можно считать завершенным. Это

довольно сложная конструкция языка CLIPS. Утешением может послужить то, что остальные условные элементы (*test*, *and*, *or*, *not*, *exists*, *forall* и *logical*) гораздо проще используют образцы в качестве основы.

2.2. Условные элементы (*test*, *and*, *or*, *not*, *exists*, *forall* и *logical*)

2.2.1. Условный элемент *test*

Условный элемент *test* предоставляет возможность наложения дополнительных ограничений на слоты фактов или объектов. Элемент *test* удовлетворяется, если вызванная в нем функция возвращает значение не-*FALSE*. Как и в случае предикатных ограничений образца в условном элементе *test*, можно использовать переменные, уже связанные со своими значениями. Внутри элемента *test* могут быть выполнены различные логические операции, например сравнения переменных.

Определение 17. Синтаксис условного элемента *test*

<условный-элемент-test > ::= (test <вызов-функции>)

Выражение *test* вычисляется каждый раз при удовлетворении других условных элементов. Это означает, что условный элемент *test* будет вычислен больше одного раза, если обрабатываемое выражение может быть удовлетворено более чем одной группой данных. Использование условного элемента *test* может стать причиной автоматического добавления правилу некоторых условных выражений. Кроме того, CLIPS может автоматически переупорядочивать условные элементы *test*.

Приведенное ниже правило находит пару фактов *data*, причем разница между значениями первых полей этих фактов должна быть больше или равной 3.

Пример 19. Применение условного элемента *test*

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

Условный элемент *test* может привести к автоматическому добавлению образцов *initial-fact* или *initial-object* в левую часть правила. Поэтому не забывайте использовать команду *reset* (которая создает *initial-*

fact и *initial-object*), чтобы быть уверенным в корректной работе условного элемента *test*.

2.2.2. Условный элемент *or*

Условный элемент *or* позволяет активировать правило любым из нескольких заданных условных элементов. Если какой-нибудь из условных элементов, объединенных с помощью *or*, удовлетворен, то и все выражение *or* считается удовлетворенным. В этом случае, если все остальные условные элементы, входящие в левую часть правила (но не входящие в *or*), также удовлетворены, правило будет активировано. Условный элемент *or* может объединять любое количество элементов.

Замечание: Правило будет активировано для каждого выражения в условном элементе *or*, которое было удовлетворено. Таким образом, условный элемент *or* производит эффект, идентичный написанию нескольких правил с похожими посылками и следствиями.

Определение 18. Синтаксис условного элемента *or*

<условный-элемент-ог > ::= (or <условный-элемент>+)

Пример 20. Применение условного элемента *or*

```
(defrule system-fault
  (error-status unknown)
  (or (temp high)
      (valve broken)
      (pump off))
  =>
  (printout t "The system has a fault." crlf))
```

Данное правило сообщит о поломке системы, если в списке фактов будет присутствовать факт *error-status unknown* и один из фактов *temp high*, *valve broken* или *pump off*. В случае если будут присутствовать два из этих трех фактов, например *temp high* и *pump off*, то сообщение будет выведено два раза. Заметьте, что приведенный пример — точный эквивалент следующих трех отдельных правил:

Пример 21. Эквивалент правилу *system-fault*

```
(defrule system-fault-1
  (error-status unknown)
  {pump off}
  =>
  (printout t "The system has a fault." crlf))
(defrule system-fault-2
```

```

        (error-status unknown)
        (valve broken)
    =>
        (printout t "The system has a fault." crlf))
(defrule system-fault-3
    (error-status unknown)
    (temp high)
    =>
        (printout t "The system has a fault." crlf))

```

2.2.3. Условный элемент *and*

Все условные элементы в левой части правил CLIPS объединены неявным условным элементом *and*. Это означает, что все условные элементы, заданные в левой части, должны удовлетвориться, для того чтобы правило было активировано. С помощью явного применения условного элемента *and* можно смешивать различные условия *and* и *or* и группировать элементы так, как этого требует логика правил. Условие *and* удовлетворяется, только если все условия внутри явного *and* удовлетворены. В случае, если остальные условия в левой части правила также истинны, правило будет активировано. Элемент *and* может объединять любое число условных элементов.

Определение 19. Синтаксис условного элемента *and*

<условный-элемент-and> ::= (and <условный-элемент>+)

Пример 22. Применение условного элемента *and*

```

(defrule system-flow
    (error-status confirmed)
    (or (and (temp high)
             (valve closed))
        (and (temp low)
             (valve open)))
    =>
        (printout t "The system is having a flow problem."
crlf))

```

Если условный элемент *and* содержит условные элементы *test* или *not* в качестве первого элемента, то перед ними автоматически добавляется образец *initial-fact* или *initial-object*. Помните, что левая часть любого правила содержит неявный элемент *and*, поэтому приведенное в примере 24 правило будет автоматически преобразовано (см. пример 25).

Пример 23. Правило nothing-to-schedule

```
(defrule nothing-to-schedule
  (not (schedule ?))
  =>
  (printout t "Nothing to schedule." crlf))
```

Пример 24. Преобразованное правило nothing-to-schedule

```
(defrule nothing-to-schedule
  (and (initial-fact)
       (not (schedule ?)))
  (printout t "Nothing to schedule." crlf))
```

2.2.4. Условный элемент *not*

Иногда важнее отсутствие информации, а не ее присутствие, т. е. возникают ситуации, когда необходимо запустить правило, если образец или другой условный элемент не удовлетворяется (например, факт не существует). Условный элемент ***not*** предоставляет эту возможность. Элемент ***not*** удовлетворяется, только если условный элемент, который он содержит, не удовлетворяется.

Определение 20. Синтаксис условного элемента *not*

<условный-элемент-not> ::= (not <условный-элемент>)

Условный элемент ***not*** может отрицать только одно выражение. Несколько условных элементов нужно отрицать с помощью нескольких элементов ***not***. Тщательно следите за комбинациями ***not*** с ***or*** или ***and***, результат не всегда очевиден!

Пример 25. Применение условного элемента *not*

```
(defrule high-flow-rate
  (temp high)
  (valve open)
  (not (error-status confirmed))
  =>
  (printout t "Recommend closing of valve due to high
              temp"
  crlf))
```

В логическом элементе ***not*** можно использовать связанные переменные, так же как и в других условных элементах:

Пример 26. Правило check-value

```
(defrule check-valve
  (check-status ?valve)
```

```

(not (valve-broken ?valve))
=>
(printout t "Device " ?valve " is OK" crlf)}

```

С помощью условного элемента ***not*** можно, наконец, довести до совершенства наше правило *Find-2-coeval-person* (пример 15). Если вы помните, это правило выводит всевозможные пары персон одинакового возраста. Чтобы данное правило не выводило эквивалентные по смыслу пары имен (например, Bob-Sue и Sue-Bob), преобразуем нашу программу следующим образом:

Пример 27. Улучшенное правило Find-2-Coeval-Person

```

(deftemplate person
  (slot name)
  (slot age))
(deftemplate person-pair
  (slot name1)
  (slot name2)
  (slot age))
(deffacts people
  (person (name Joe) (age 20))
  (person (name Bob) (age 20))
  (person (name Joe) (age 34))
  (person (name Sue) (age 34))
  (person (name Sue) (age 20)))
(defrule Find-2-Coeval-Person
  (person (name ?x) (age ?z))
  (person (name ?y&~?x) (age ?z))
  (not (person-pair (name1 ?x) (name2 ?y) (age ?z)))
  (not (person-pair (name1 ?y) (name2 ?x) (age ?z)))
  =>
  (printout t "name=" ?x " name=" ?y " age=" ?z crlf)
  (assert (person-pair (name1 ?x) (name2 ?y) (age ?z))))

```

Обратите внимание на произведенные изменения. Во-первых, с помощью конструктора *deftemplate* был добавлен дополнительный шаблон *person-pair*. В фактах, соответствующих данному шаблону, будет храниться информация об уже найденных парах ровесников. Кроме того, было сильно изменено и само правило. В его левой части было добавлено два условия:

```

(not (person-pair (name1 ?x) (name2 ?y) (age ?z)))
(not (person-pair (name1 ?y) (name2 ?x) (age ?z)))

```

Эти условные элементы проверяют наличие фактов типа *person-pair* и, тем самым отслеживают, была ли уже обработана данная пара или ее перестановка. Если эти факты отсутствуют, то это означает, что обработка еще не была выполнена. В этом случае правило активируется, и выполняются действия, описанные в правой части правила. А именно выводится на экран сообщение о найденной паре ровесников и добавляется соответствующий факт *person-pair*, утверждающий, что данная пара уже была обработана. Для запуска программы выполните команды **reset** и **run**. Программа выведет на экран следующую информацию:

Пример 28. Результат работы правила Find-2-Coeval-Person

name=Sue	name=Bob	age=20
name=Sue	name=Joe	age=20
name=Sue	name=Joe	age=34
name=Bob	name=Joe	age=20

Если вы внимательно посмотрите на полученный результат и исходные данные, то обнаружите, что это именно то, что нам было нужно. Это список всевозможных ровесников без повторений и с исключением того факта, что все люди являются ровесниками сами себе. Теперь наше правило достигло полного совершенства! Обратите внимание на тот факт, что если вы повторно попытаете выполнить команду **run**, то ничего не увидите. Это происходит потому, что и списке фактов содержится информация обо всех обработанных парах, оставшаяся после первого запуска. Для того чтобы повторно запускать данный пример, выполняйте команду **reset** перед каждой командой **run**.

Условный элемент **not**, так же как и **test**, может привести к автоматическому добавлению образцов *initial-fact* или *initial-object* в левой части правил. Поэтому не забывайте использовать команду **reset** (которая создает *initial-fact* и *initial-object*), чтобы быть уверенным в корректной работе условного элемента **not**.

В условный элемент **not**, содержащий элемент **test**, автоматически преобразуется в элемент **not**, содержащий **and** с *initial-fact* и исходным элементом **test**. Например, следующий условный элемент из примера 34 преобразуется в элемент из примера 35.

Пример 29. Условный элемент not, содержащий элемент test

(not (test (> ?time-1 ?time-2)))

Пример 30. Преобразованный условный элемент not, содержащий элемент test

(not and (initial-fact)

```
(test (> ?time-1 ?time-2)))
```

Замечание: Заметьте, что наиболее простым и правильным способом записи данного выражения будет: `(test (not (> ?time-1 ?time-2)))`.

2.2.5. Условный элемент *exists*

Условный элемент *exists* позволяет определить, существует ли хотя бы один набор данных (фактов или объектов), которые удовлетворяют условным элементам, заданным внутри элемента *exists*.

Определение 21. Синтаксис условного элемента *exists*

```
<условный-элемент-exists> ::= (exists <условный-элемент>+)
```

CLIPS автоматически заменяет *exists* двумя последовательными условными элементами *not*. Например, следующее правило (пример 32) будет преобразовано в правило из примера 33.

Пример 31. Правило example

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

Пример 32. Преобразованное правило example

```
(defrule example
  (not (not (and (a ?x) (b ?x))))
=>)
```

Так как внутренний способ реализации *exists* использует условный элемент *not*, то для *exists* справедливы все замечания и ограничения описанные ранее.

Пример 33. Использование условного элемента *exists*

```
(deftemplate hero
  (multislot name)
  (slot status (default unoccupied)))
(deffacts goal-and-heroes
  (goal save-the-world)
  (hero (name Death Defying Man))
  (hero (name Stupendous Man))
  (hero (name Incredible Man)))
(defrule save-the-world
  (goal save-the-world)
  (exists (hero (status unoccupied)))
=>)
```

```
(printout t "The day is saved." crlf))
```

Данная программа определяет шаблон — героя, имеющего составное поле с именем героя и простое поле, содержащее статус "не занят" по умолчанию. Конструктор *deffacts* определяет трех ничем не занятых героев и текущую цель — спасение мира. Правило проверяет, есть в данный момент эта цель, и в случае положительного ответа проверяет, если ли какой-нибудь еще не занятый герой. Если все условные элементы правила удовлетворены, оно сообщает, что мир спасен. Обратите внимание: несмотря на то, что у нас все три героя не заняты, правило будет активировано только один раз.

Так как способ реализации *exists* использует условный элемент *not*, то условный элемент *exists* может привести к автоматическому добавлению образцов *initial-fact* или *initial-object* в левую часть правила. Поэтому не забывайте использовать команду *reset* (которая создает *initial-fact* и *initial-object*), чтобы быть уверенным в корректной работе условного элемента *exists*.

2.2.6. Условный элемент *forall*

Условный элемент *forall* позволяет определить, что некоторое заданное условие выполняется для всех заданных условных элементов.

Определение 22. Синтаксис условного элемента *forall*

```
<условный-элемент-forall> ::= (forall <условный-элемент>  
                                <условный-элемент>+)
```

CLIPS автоматически заменяет *forall* комбинацией условных элементов *not* и *and*. Например, следующее правило (пример 35) будет преобразовано так, как показано в примере 36.

Пример 34. Правило example

```
(defrule example  
  (forall (a ?x) (b ?x) (c ?x))=>)
```

Пример 35. Преобразованное правило example

```
(defrule example  
  (not (and (a ?x)  
            (not (and (b ?x)  
                      (c ?x)))) =>)
```

Рассмотрим следующий пример. Правило *all-students-passed* определяет, прошли ли все студенты чтение, письмо и арифметику, используя условие *forall*:

Пример 36. Правило all-students-passed


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

Заметьте, что данное правило удовлетворяется, пока нет ни одного студента. При добавлении факта (*student Bob*) правило перестает удовлетворяться, т. к. нет фактов, подтверждающих, что *Bob* прошел все необходимые предметы. Правило не начнет удовлетворяться и после добавления фактов (*reading Bob*) и (*writing Bob*). А вот после добавления факта (*arithmetic Bob*) правило будет активировано и сможет вывести на экран соответствующую запись. Если добавить факт (*student John*), правило опять перестанет удовлетворяться, т. к. один из студентов (*John*) не прошел все необходимые предметы. Используя условный элемент *exists*, вы без труда сможете изменить это правило так, чтобы оно не выполнялось в случае отсутствия студентов.

Так как реализация *forall* использует условный элемент *not*, то *forall*, так же как и *not*, *test* и *exists*, может привести к автоматическому добавлению образцов *initial-fact* или *initial-object* в левую часть правила. Не забывайте использовать команду *reset* для корректной работы этого условного элемента.

2.2.7. Условный элемент *logical*

Условный элемент *logical* предоставляет механизм поддержки достоверности для созданных правилом данных (фактов или объектов), удовлетворяющих образцам. Данные, созданные в правой части правила, могут иметь логическую зависимость от данных, удовлетворивших образцы в левой части правила. Такая зависимость называется *логической поддержкой*. Данные могут зависеть от группы данных или нескольких групп данных, удовлетворивших одно или несколько правил. Если удаляются данные, которые поддерживают некоторые другие данные, то зависимые данные также автоматически удаляются.

Если некоторые данные созданы без логической поддержки (например, с помощью конструкторов *deffacts*, *definstance* или команды *assert*, введенной пользователем или вызванной в правой части правила), то считается, что они имеют безусловную поддержку. Безусловная

поддержка удаляет все присутствующие в данный момент условные поддержки этих данных (но не удаляет сами данные). Дальнейшая логическая поддержка для данных с безусловной поддержкой игнорируется. Удаление правила, которое вызвало логическую поддержку для данных, удаляет логическую поддержку, сгенерированную этим правилом (но не удаляет данные, если у них еще есть логическая поддержка, сгенерированная другим правилом).

Определение 23. Синтаксис условного элемента *logical*

<условный-элемент-logical> ::= (logical <условный-элемент>+)

Условный элемент *logical* группирует образцы, так же как это делает *and*. Данное свойство можно использовать при объединении элементов *and*, *or* и *not*. Однако только первые *n* образцов правила могут использоваться в условном элементе *logical*. Например, следующее правило записано верно:

Пример 37. Правильный вариант использования условного элемента *logical*

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

А такое объявление правил недопустимо:

Пример 38. Неправильные варианты использования условного элемента *logical*

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

(defrule not-ok-2
  (a)
  (logical (b))
  (logical (c))
=>
  (assert (d)))

(defrule not-ok-3
  (or (a)
```

```

        (logical (b)))
    (logical (c))
=>
    (assert (d)))

```

Рассмотрим следующий пример. Включите просмотр списка фактов с помощью пункта **Facts Window** меню **Windows**, это поможет следить за тем, что происходит в момент выполнения программы.

Пример 39. Использование условного элемента *logical*

```

(clear)
(reset)
(defrule example
    (logical (a))
    (b)
=>
    (assert (c)))
(assert (a) (b))
(run)
(retract 2)
(retract 1)

```

По команде **run** правило *example*, активированное фактами *a* и *b*, добавляет новый факт *c*, который имеет логическую поддержку (зависит) от факта *a*.

После удаления факта *b* с помощью команды (**retract 2**) ничего особенного не происходит, но если мы удалим факт *a*, то увидим, что это тут же приведет к удалению связанного с ним факта *c*.

Итак, условный элемент **logical** может быть использован для создания данных, которые будут логически связаны с изменениями некоторых отдельных слотов объекта, а не от всего объекта целиком. Данную возможность можно использовать только при работе с объектом. При работе с шаблонами фактов данную возможность использовать нельзя, т. к. изменения слота факта, фактически приводит к удалению старого факта и добавления нового с измененными слотами и индексом. В отличие от фактов изменения слотов объекта выполняются без удаления объекта. Это поведение иллюстрируется приведенным ниже примером:

Пример 40. Использование условного элемента *logical* с объектами

```

(clear)
(defclass A (is-a USER)
    (role concrete)

```

```
(pattern-match reactive)
(slot foo (create-accessor write))
(slot bar (create-accessor write))
(defrule match-A
  (logical (object (is-a A) (foo ?)))
=>
  (assert (new-fact)))
(make-instance a of A)
(run)
(send [a] put-foo 100)
```

После выполнения команды **run** правило *match-A* добавляет факт *new-fact*, логически связанный с конкретным значением слота *foo* объекта *a*. При изменении значения данного слота факт *new-fact* автоматически удаляется из списка фактов.

3. КОМАНДЫ И ФУНКЦИИ ДЛЯ РАБОТЫ С ПРАВИЛАМИ

3.1. Просмотр и удаление существующих правил

Рассмотрим функции и команды, предоставляемые CLIPS для работы с правилами. После создания правил с помощью конструктора **defrule** вполне естественно возникает желание сделать что-нибудь с уже существующим правилом. CLIPS поддерживает множество различных команд, оперирующих с правилами. Рассмотрим наиболее часто используемые команды: **ppdefrule**, **list-defrules** и **undefrule**.

1. С помощью команды **ppdefrule** можно просмотреть определение правила в том виде, в котором оно было создано с помощью конструктора **defrule**.

Определение 24. Синтаксис команды *ppdefrule*

```
(ppdefrule <имя-правила>)
```

2. Для того чтобы получить полный список правил, присутствующих в CLIPS в данный момент, используется команда **list-defrules**.

Определение 25. Синтаксис команды *list-defrules*

```
(list-defrules <имя-модуля>)
```

Полный синтаксис этой команды содержит необязательный аргумент *<имя-модуля>*. Если данный аргумент не задан, то будет выведен список правил, определенных в текущем модуле. В случае явного задания модуля будет список правил, принадлежащих конкретному модулю. Данный

аргумент может принимать значение *. В этом случае на экран будет выведен список всех правил из всех модулей.

3. Для удаления правила используется команда *undefrule*.

Определение 26. Синтаксис команды undefrule

(undefrule <имя-правила>)

В качестве параметра команда *undefrule* принимает имя правила, которое нужно удалить. Если в качестве имени правила был задан символ *, то будут удалены все правила.

Для демонстрации работы команд, будем использовать следующие правила:

Пример 41. Необходимые для дальнейшей работы правила

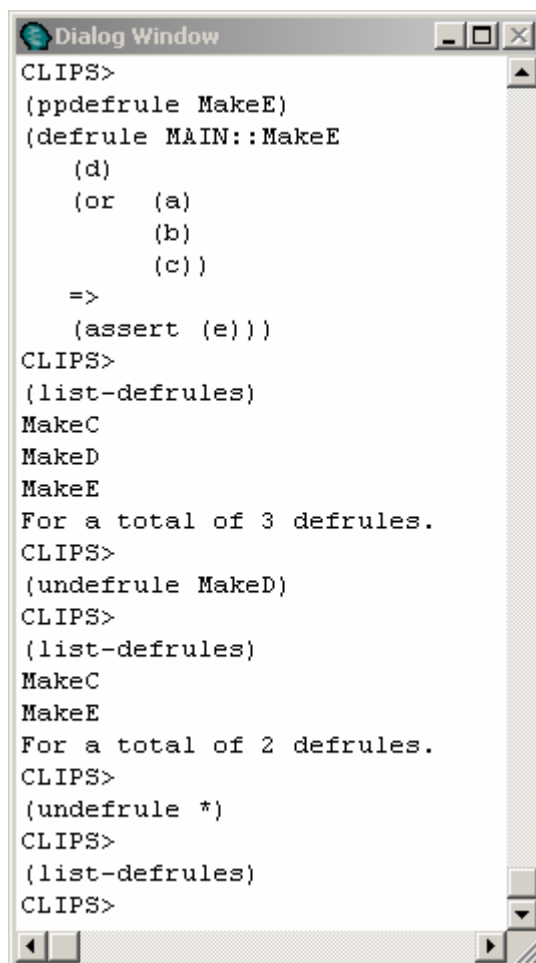
```
(defrule MakeC
  (a)
  (b)
  =>
  (assert (c)))
(defrule MakeD
  (c)
  (or (a)
      (b))
  =>
  (assert (d)))
(defrule MakeE
  (d)
  (or (a)
      (b)
      (c))
  =>
  (assert (e)))
```

Введите эти правила в среду CLIPS, а затем выполните следующую последовательность команд:

Пример 42. Использование команд *ppdefrule*, *list-defrules* и *undefrule*

```
(ppdefrule MakeE)
(list-defrules)
(undefrule MakeD)
(list-defrules)
(undefrule *)
(list-defrules)
```

Если приведенные выше действия были выполнены правильно, то полученный результат должен соответствовать рис. 7.



```
Dialog Window
CLIPS>
(ppdefrule MakeE)
(defrule MAIN::MakeE
  (d)
  (or (a)
      (b)
      (c))
  =>
  (assert (e)))
CLIPS>
(list-defrules)
MakeC
MakeD
MakeE
For a total of 3 defrules.
CLIPS>
(undefrule MakeD)
CLIPS>
(list-defrules)
MakeC
MakeE
For a total of 2 defrules.
CLIPS>
(undefrule *)
CLIPS>
(list-defrules)
CLIPS>
```

Рис 7. Результат применения команд **ppdefrule**, **list-defrules** и **undefrule**

В Windows-версии CLIPS доступен инструмент под названием **Defrule Manager** (Менеджер правил). Если в данный момент в среде CLIPS отсутствуют правила, то пункт **Defrule Manager** меню **Browse** не будет доступен. Если вы повторно заведете приведенные выше правила и откроете менеджер правил, то должны будете увидеть результат, приведенный на рис. 8.

Менеджер отображает список всех правил, доступных в данный момент. Общее количество правил отображается в заголовке окна менеджера, в данный момент это **Defrule Manager - 3 Items**. С помощью кнопок **Remove** и **Pprint** можно удалять и выводить определение

выбранного правила соответственно. Вся информация, получаемая от менеджера правил, отображается непосредственно в главном окне CLIPS.

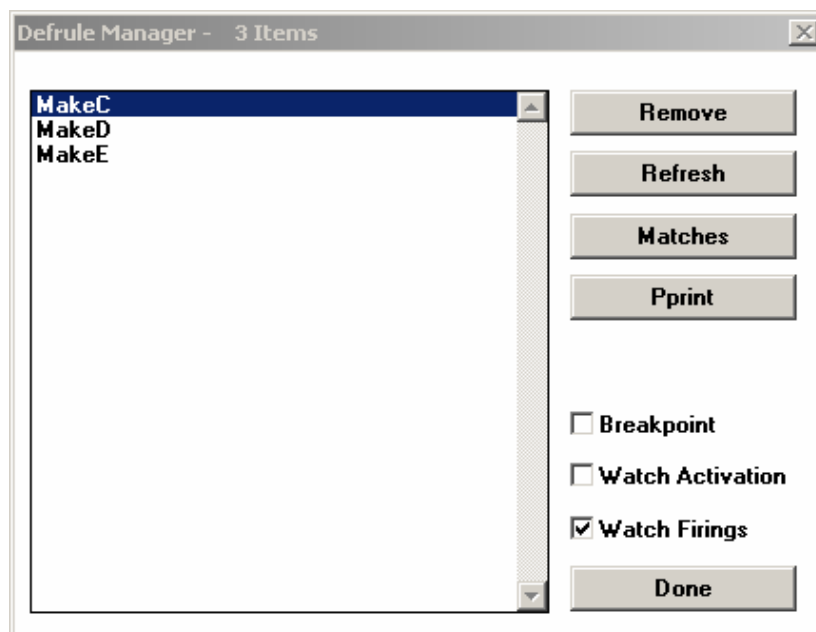


Рис 8. Просмотр списка правил с помощью менеджера правил

CLIPS не содержит специальных команд для изменения существующих правил. Чтобы изменить существующее правило, пользователю необходимо заново определить такое правило с помощью конструктора *defrule*. При этом существующее определение правила будет автоматически удалено из системы, даже если новый конструктор содержал ошибки, и новое правило добавлено не было.

3.2. Сохранение и загрузка правил

Создавать правила конструктором *defrule* каждый раз, по мере необходимости используя для этого среду CLIPS, довольно неудобно. Для облегчения участи пользователя CLIPS позволяет загружать конструкторы правил (как, впрочем, и все остальные конструкторы) из текстового файла. Для этого используется следующая команда:

Определение 27. Синтаксис команды *load*

(load <имя-файла>)

Имя файла должно быть строкой, т. е. заключаться в кавычки. Имя файла может содержать полный путь к файлу. В противном случае система будет искать файл в текущем каталоге. Для создания файла в принципе можно использовать любой ASCII-редактор, но лучше применять

встроенный редактор, предоставляемый средой CLIPS. Встроенный редактор поддерживает несколько дополнительных функций, чрезвычайно полезных при разработке программ.

- ❑ Во-первых, он способен проверять синтаксис функций баланс открывающих и закрывающих скобок, помогает в расстановке и удалении комментариев и т. д. Если вы будете использовать встроенный редактор для создания серьезной экспертной системы, вы по достоинству оцените эти возможности.
- ❑ Во-вторых, встроенный редактор позволяет быстро загружать в среду отдельные конструкторы и команды. Эта возможность помогает проверять и тестировать большую экспертную систему.
- ❑ И, наконец, в-третьих, редактор предоставляет помощь по среде и языку, которая бывает чрезвычайно полезной, даже при наличии большого опыта работы в CLIPS.

По умолчанию файлы, созданные во встроенном редакторе CLIPS, получают расширение ***.clp**.

Для начала работы с редактором просто выберите пункт **New** меню **File**.

Создайте в CLIPS файл *example1.CLP* с тремя приведенными выше правилами. После чего очистите CLIPS с помощью команды **clear** и выполните команду (**load "example1.CLP"**). Полученный результат должен соответствовать рис. 9.

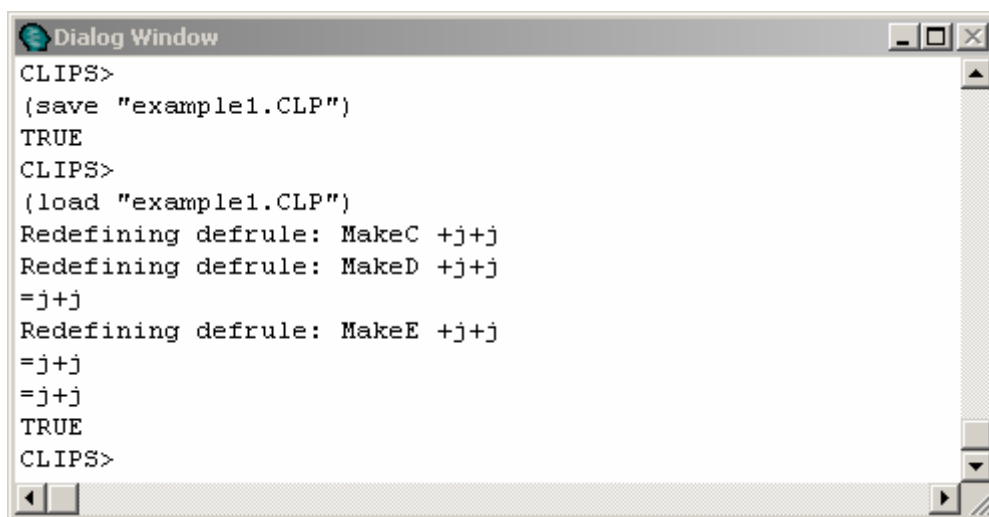


Рис 9. Результат загрузки файла **example1.CLP**

Команда **load** отображает процесс загрузки каждого конструктора. В случае успешной загрузки всех определенных в файле конструкторов

команда возвращает значение TRUE, в противном случае — информацию об ошибке. В случае если была найдена ошибка, процесс загрузки файла прекращается.

CLIPS поддерживает также команду *load**. Эта команда полностью идентична *load* за исключением того, что она не отображает процесса загрузки конструкторов.

Определение 28. Синтаксис команды *load**

(load* <имя-файла>)

CLIPS предоставляет также команду *save*, которая позволяет сохранять в текстовый файл все конструкторы, определенные в данный момент в системе. Синтаксис этой команды идентичен синтаксису команд *load* и *load**.

Определение 29. Синтаксис команды *save*

(save <имя-файла>)

Текстовый формат не единственный способ хранения конструкторов CLIPS. Команды *bsave* и *bload* позволяют сохранять и загружать конструкторы в двоичном виде. Двоичные файлы загружаются гораздо быстрее, чем текстовые, но занимают больше места (т. к. кроме конструкторов они хранят полную информацию о текущем состоянии среды). Еще одним неудобством использования двоичных файлов является то, что создавать их можно только непосредственно в среде CLIPS.

Большинство описанных выше команд для работы с файлами (а именно *load*, *save*, *bsave* и *bload*) доступны в меню **File** Windows-версии среды CLIPS. Это команды **Load**, **Save**, **Save Binary** и **Load Binary** соответственно. Все используют стандартные Windows-диалоги для выбора файлов.

3.3. Запуск и остановка программы

Как было замечено, для запуска CLIPS-программ используется команда *run*.

Определение 30. Синтаксис команды *run*

(run <целочисленное-выражение>)

Целочисленное выражение является необязательным аргументом команды *run*. В простейшем случае в качестве этого аргумента можно использовать любую целую константу. Если данный аргумент задан и он положителен, то CLIPS запустит на выполнение заданное число правил из плана решения задачи. Если данное число больше числа правил в плане решения задачи, то будут запущены все правила. В случае если аргумент

не задан или является отрицательным, план решения задачи также будет выполнен полностью.

В Windows-версии CLIPS в меню **Execution** доступны две версии команды *run* — **Run** и **Step**. Первая команда использует версию команды *run* без аргументов и запускает все правила из плана решения задачи. Программа **Step** позволяет трассировать программу и выполнять заданное число правил. По умолчанию это число равно 1, но эту установку среды можно изменить с помощью диалогового окна **Preferences**. Для запуска этого диалогового окна выберите пункт **Preferences** меню **Execution**. Общий вид диалогового окна показан на рис. 10. Количество правил, запущенных за один шаг трассировки, отображается в поле **Step Rule Firing Increment**.

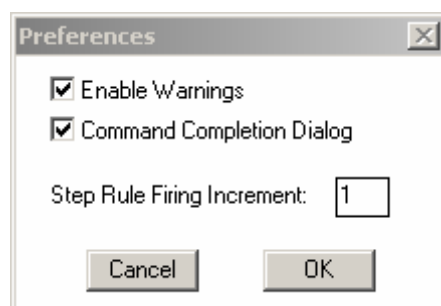


Рис 10. Диалоговое окно **Preferences**

Кроме выполнения программы по шагам, CLIPS позволяет установку точек останова (*breakpoints*) на отдельных правилах.

Определение 31. Синтаксис команды *set-break*

(*set-break* <имя-правила>)

Если точка останова определена для заданного правила, то выполнение программы прекратится перед запуском этого правила. Точка останова не останавливает правило, если это первое правило в плане решения задачи.

Удалить точки останова можно с помощью команды *remove-break*.

Определение 32. Синтаксис команды *remove-break*

(*remove-break* <имя-правила>)

В случае выполнения команды *remove-break* без параметров CLIPS удалит все определенные ранее точки останова.

Устанавливать и снимать точки останова также можно с помощью менеджера правил, внешний вид которого представлен на рис. 7. Для этого выберите правило и установите флажок **Breakpoint**.

В случае если выполнение программы необходимо остановить, используйте команду *halt* без аргументов.

Определение 33. Синтаксис команды *halt* (*halt*)

Диалоговое окно **Watch Options** (пункт **Watch** меню **Execution**) позволяет установить флажок **Statistics**, как показано на рис. 11. В этом случае после выполнения каждой команды *run* CLIPS будет выводить статистическую информацию о количестве запущенных правил, полном и среднем времени выполнения правил, количестве добавленных фактов и т. д.



Рис 11. Установка режима с выводом статистической информации

Если вы установите флажок **Statistics**, загрузите файл *example1.CLP*, добавьте факты (a) и (b) и запустите программу, то увидите результаты, представленные на рис. 12.

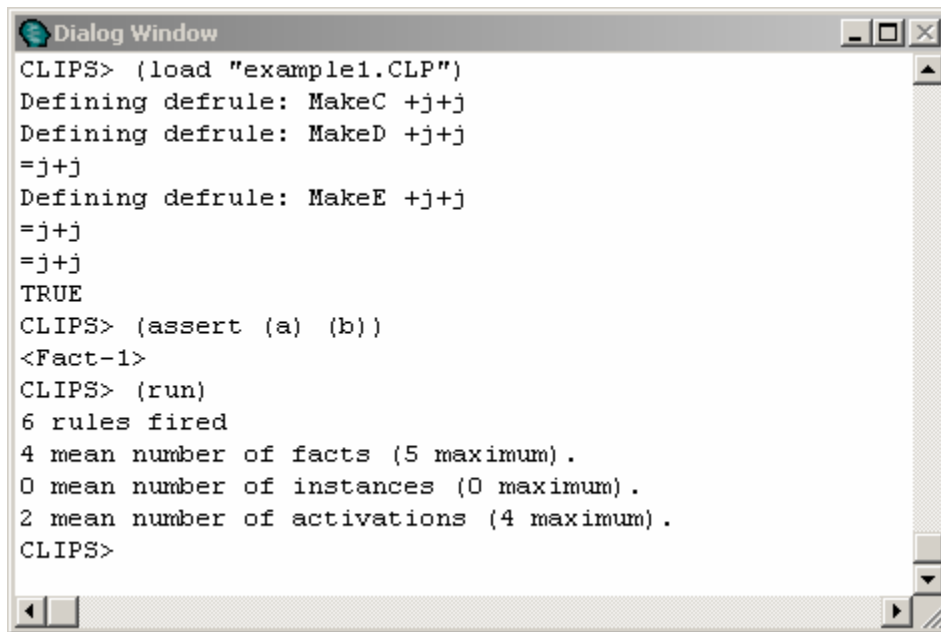
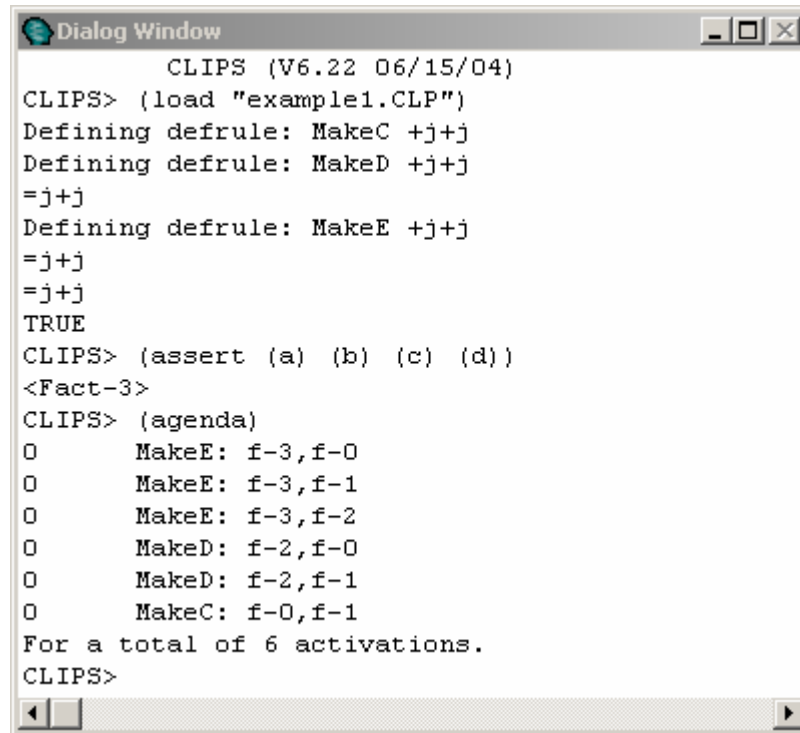


Рис 12. Получение статистической информации

3.4. Просмотр плана решения задачи

План решения задачи (*agenda*) можно просматривать различными способами. Самый простой из них — команда ***agenda***, набранная в главном окне CLIPS. Очистите CLIPS, загрузите файл *example1.CLP*, добавьте факты *a*, *b*, *c* и *d* и вызовите команду ***agenda***. Полученный результат должен соответствовать приведенному на рис. 13.



```
CLIPS (V6.22 06/15/04)
CLIPS> (load "example1.CLP")
Defining defrule: MakeC +j+j
Defining defrule: MakeD +j+j
=j+j
Defining defrule: MakeE +j+j
=j+j
=j+j
TRUE
CLIPS> (assert (a) (b) (c) (d))
<Fact-3>
CLIPS> (agenda)
0      MakeE: f-3,f-0
0      MakeE: f-3,f-1
0      MakeE: f-3,f-2
0      MakeD: f-2,f-0
0      MakeD: f-2,f-1
0      MakeC: f-0,f-1
For a total of 6 activations.
CLIPS>
```

Рис 13. Просмотр плана решения задачи

План решения задачи содержит 6 активаций правил. По команде **agenda** все эти активации будут выведены на экран вместе с приоритетом правил (слева от имени правила) и списком данных, активировавших правило (справа от имени правила). Порядок правил в плане решения задачи сильно зависит от выбранной стратегии разрешения конфликтов и приоритета правил.

Кроме этого, Windows-версия CLIPS позволяет выводить план решения задачи в отдельном окне — **Agenda**. Для того чтобы сделать окно видимым, воспользуйтесь пунктом **Agenda Window** меню **Window**. Внешний вид этого окна показан на рис. 14, его содержимое полностью соответствует информации, получаемой с помощью команды **agenda**. Данный инструмент чрезвычайно полезен при отладке программ или для наблюдения за изменением плана решения задачи в процессе выполнения программы.

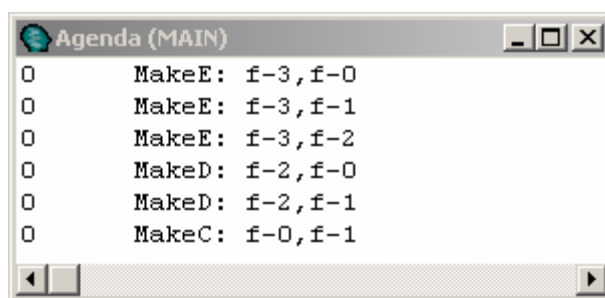


Рис 14. Окно Agenda

Помимо окна **Agenda**, которое позволяет только просмотр, CLIPS предоставляет еще один удобный визуальный инструмент — **Agenda Manager** (Менеджер плана решения задачи), который позволяет в случае необходимости корректировать план решения задачи. Для вызова менеджера плана решения задачи выберите пункт **Agenda Manager** меню **Browse**. Внешний вид этого инструмента приведен на рис. 15. С его помощью можно удалять из плана решения задачи отдельные активации правил или запускать правила в некотором произвольном порядке.

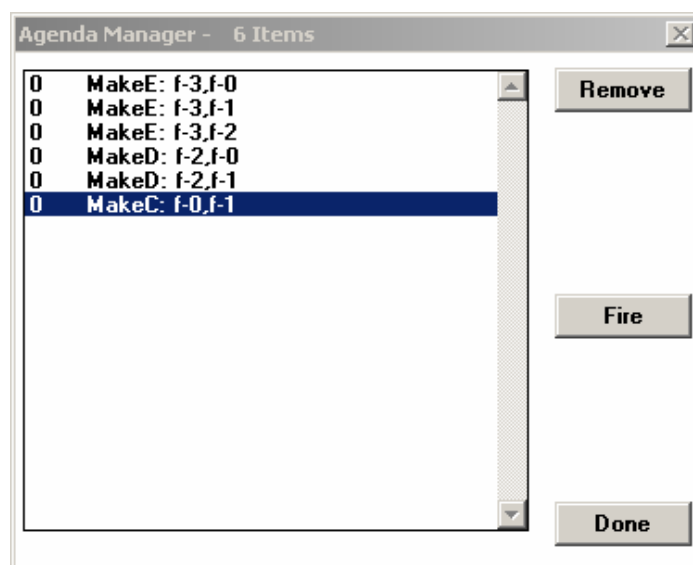


Рис 15. Окно менеджера плана решения задачи

С помощью диалогового окна **Watch Options** (см. рис. 10) или менеджера правил можно задавать режим отображения активаций и/или запуска правил. В этом случае пользователь будет получать соответствующее информационное сообщение при добавлении правила в план решения задачи или при удалении правила из него, а также при каждом запуске правила.

3.5. Просмотр данных, способных активировать правило

CLIPS предоставляет возможность просматривать списки наборов данных (фактов или объектов), способных активировать заданное правило.

Определение 34. Синтаксис команды **matches**

(matches <имя-правила>)

Команда **matches** выводит информацию обо всех возможных наборах данных, способных активировать это правило. Посмотрите на результаты выполнения данной команды для правил *MakeC* и *MakeD* (наличие фактов *a*, *b* и *c* обязательно), приведенные на рис. 16 и 17 соответственно.

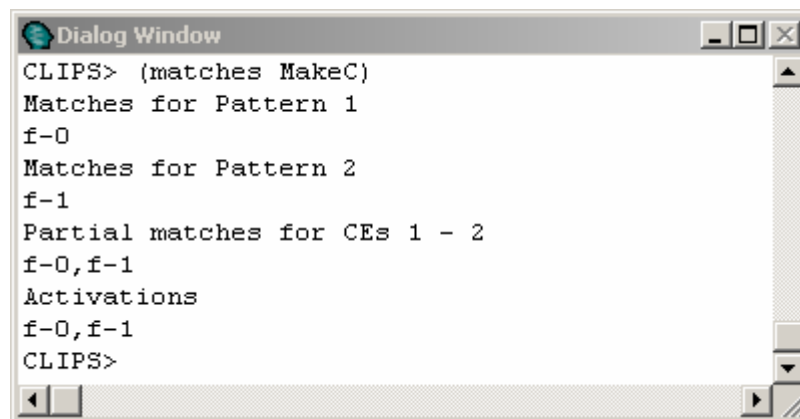


Рис 16. Данные, активирующие правило MakeC

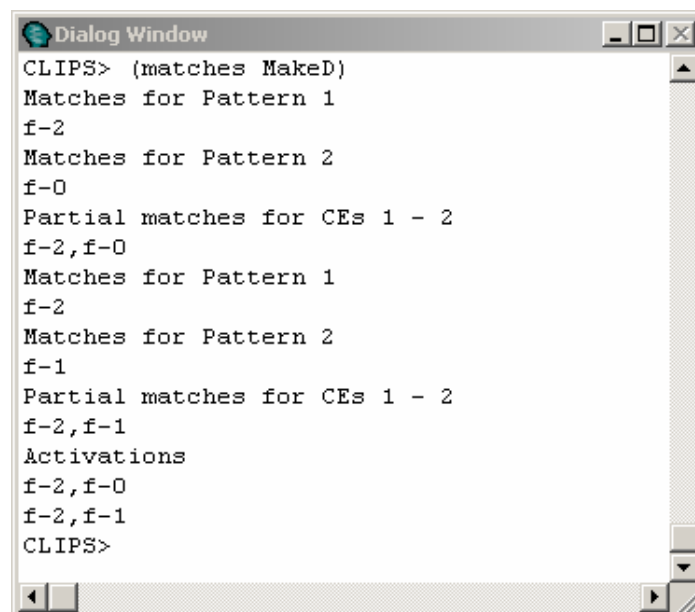


Рис 17. Данные, активирующие правило MakeD

ЗАДАНИЕ

Используя условные элементы *test*, *and*, *or*, *not*, *exist*, *forall*, *logical* составьте 10 правил, описывающих предметную область, заданную преподавателем.

4. КОНТРОЛЬНЫЕ ВОПРОСЫ:

1. Для чего служат правила в CLIPS?
2. Расскажите об основных составляющих правила?
3. Для каких целей используется специальный конструктор *defrule*?
4. Какое ключевое слово используется в CLIPS, для задания свойств правила?
5. Какие основные ограничения, использующиеся в образцах?
6. Свойства правила *salience* и *auto-focus*.
7. Какие стратегии разрешения конфликтов поддерживает CLIPS?
8. Перечислите типы условных элементов, используемых в левой части правил?
9. Какие типы ограничений поддерживает CLIPS, кроме символьных?
10. Какие связывающие ограничения, предназначенные для объединения отдельных ограничений и переменных в единое целое использует CLIPS?
11. С помощью какой команды можно просмотреть данные, способные активизировать правило?
12. Какую задачу выполняют в CLIPS команды: *ppdefrule*, *list-defrules* и *undefrule*?