

ЛАБОРАТОРНАЯ РАБОТА №3

ТЕМА: Знакомство со средой CLIPS 6.2. Работа с глобальными переменными в среде CLIPS. Создание новых функций в среде CLIPS при помощи конструктора *deffunction*. Работа с родовыми функциями.

ЦЕЛЬ: Научиться общим приемам работы в среде CLIPS. Научиться использовать имеющиеся возможности CLIPS для работы с глобальными переменными. Научиться создавать новые функций в среде CLIPS при помощи конструктора *deffunction*. Научиться использовать конструкцию языка CLIPS такую, как родовые функции; изучить методы их создания, приемы и способы использования; изучить алгоритм родового связывания.

ПЛАН ЗАНЯТИЯ:

1. Работа с глобальными переменными.
2. Приемы работы с внешними функциями.
3. Создание родовой функции.
4. Визуальные инструменты для работы с родовыми функциями.
5. Контрольные вопросы.

1. РАБОТА С ГЛОБАЛЬНЫМИ ПЕРЕМЕННЫМИ

Помимо фактов, CLIPS представляет еще один способ представления данных – глобальные переменные. В отличие от переменных, связанных со своим значением в левой части правила, глобальная переменная (*globals*) доступна везде после своего создания (а не только в правиле, в котором она получила свое значение). Глобальные переменные CLIPS подобны глобальным переменным в процедурных языках программирования, таких как C или ADA. Однако, в отличие от переменных большинства процедурных языков программирования, глобальные переменные в CLIPS слабо типизированы. Фактически переменная может принимать значение любого примитивного типа CLIPS при каждом новом присваивании значения.

1.1. Конструктор *defglobal* и функции для работы с глобальными переменными

В среде CLIPS могут быть объявлены глобальные переменные и присвоены их начальные значения с помощью конструктора *defglobal*. В дальнейшем значения созданных таким образом переменных будут доступны в любых конструкциях CLIPS. Глобальные переменные могут использоваться в процессе сопоставления образцов, но их изменения не запускает этот процесс.

Определение 1. Синтаксис конструктора *defglobal*

(defglobal [⟨имя-модуля⟩ <определение-переменной>)]*

<определение-переменной> ::= <имя-переменной> = <выражение>
<имя-переменной> ::= ?<значение-типа-symbol>**

В одном конструкторе может быть объявлено произвольное количество переменных. CLIPS позволяет использовать произвольное количество конструкторов *defglobal*. Необязательный параметр *<имя-модуля>* определяет модуль, в котором должны быть определены конструируемые переменные. Если имя модуля не задано, то переменные будут помещены в текущий модуль. В случае если создаваемая переменная уже была определена, то старое определение будет заменено новым. Если при выполнении конструктора *defglobal* возникает ошибка, то CLIPS произведет добавление всех переменных, заданных до ошибочного определения.

Команды, использующие глобальные переменные, например, такие как *ppdefglobal* или *undefglobal*, применяют значение типа *symbol*, являющееся именем переменной без символов ? и * (например, *max* для переменной, определенной как *?*max**).

Глобальные переменные могут быть использованы в любом месте, где могут быть использованы переменные, созданные в левой части правил с некоторыми исключениями. Во-первых, глобальные переменные не могут использоваться как параметры в конструкторах *deffunction*, *defmethod* или обработчиках сообщений. Во-вторых, глобальные переменные не могут использоваться для получения новых значений в левой части правил. Например, правило из примера 1 недопустимо.

Пример 1. Неверное использование глобальной переменной

(defrule example
*(fact ?*x*) =>)*

А применение глобальной переменной так, как представлено в примере 2, вполне возможно.

Пример 2. Допустимое применение глобальной переменной

```
(defrule example
  (fact ?y & :(> ?y ?*x*))
  =>)
```

Изменение глобальной переменной не приводит к запуску процесса сопоставления образцов. Например, если в базу знаний системы был добавлен факт (*fact 3*) и переменной *?*x** было присвоено значение 4, правило не будет активировано из-за того, что в системе отсутствует набор данных, удовлетворяющих правилу. Если после этого переменной *?*x** присвоить значение 2, несмотря на то, что текущий набор данных удовлетворяет всем условиям правила, оно все равно не будет активировано, т.к. изменение глобальной переменной не привело к запуску процесса сопоставления образцов

Рассмотрим пример использования конструктора *defglobal*.

Пример 3. Использование конструктора *defglobal*

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

После выполнения данного конструктора в CLIPS появятся 4 глобальные переменные: *x*, *y*, *z* и *q*. Переменной *x* присваивается целое значение 3. Переменной *y* — значение, сохраненное в глобальной переменной *x* (т. е. 3). Переменной *z* — сумма значений *x* и *y* (т. е. 6). Переменной *q* присваивается значение, равное составному полю, содержащему 3 значения типа *symbol* (*a*, *b* и *c*), созданному с помощью функции *create\$*. В случае если в конструкторе *defglobal* не было допущено синтаксических ошибок, то *defglobal* не возвращает никаких значений. Если ошибки имели место, то пользователь получит соответствующее сообщение. Обратите внимание, что переменная *y* не является указателем на переменную *x*, просто их значения в данный момент совпадают. Если изменить значение *x*, значения переменных *y* и *z*, несмотря ни на что, останутся равными 3 и 6 соответственно.

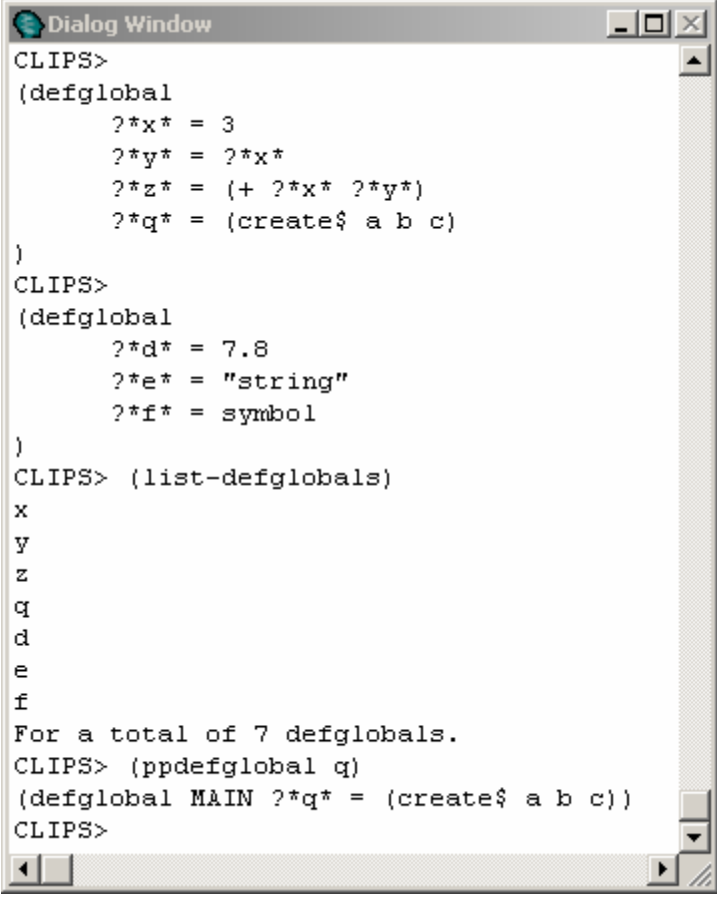
Чтобы увидеть результат работы конструктора *defglobal*, можно воспользоваться командой *list-defglobals*, для вывода на экран списка всех

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

Пример 4. Глобальные переменные различных типов

```
(defglobal
  ?*d*=7.8
  ?*e*="string"
  ?*f* = symbol
)
```

Выполните после этого команду (*list-defglobals*), а также команду (*ppdefglobal q*), которая выведет на экран определение конкретной переменной. Результат описанных действий должен соответствовать рис. 1.



```
Dialog Window
CLIPS>
(defglobal
  ?*x* = 3
  ?*y* = ?*x*
  ?*z* = (+ ?*x* ?*y*)
  ?*q* = (create$ a b c)
)
CLIPS>
(defglobal
  ?*d* = 7.8
  ?*e* = "string"
  ?*f* = symbol
)
CLIPS> (list-defglobals)
x
y
z
q
d
e
f
For a total of 7 defglobals.
CLIPS> (ppdefglobal q)
(defglobal MAIN ?*q* = (create$ a b c))
CLIPS>
```

Рис 1. Результат выполнения команд **list-defglobals** и **ppdefglobal**

1.2. Визуальные инструменты для работы с глобальными переменными

Помимо приведенных выше команд, Windows-версия содержит два визуальных инструмента для контроля количества и состояния созданных глобальных переменных. Первый из этих инструментов — **Globals Window** (окно глобальных переменных), изображен на рис. 2. Для того чтобы сделать окно глобальных переменных видимым, используйте пункт **Globals Window** меню **Window**. Этот инструмент позволяет следить за изменением списка глобальных переменных, определенных в системе, например при трассировке или отладки программы.

Другой инструмент, предназначенный для работы с глобальными переменными, называется **Defglobal Manager** (Менеджер глобальных переменных). Этот инструмент доступен в меню **Browse**, пункт **Defglobal Manager**. Его внешний вид представлен на рис. 3.

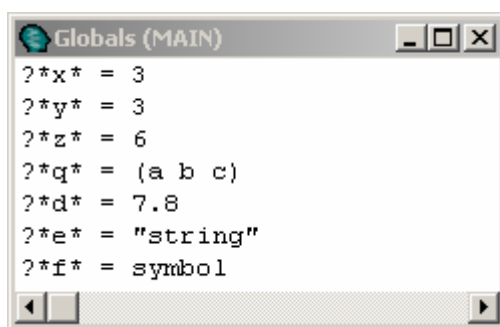


Рис 2. Окно глобальных переменных

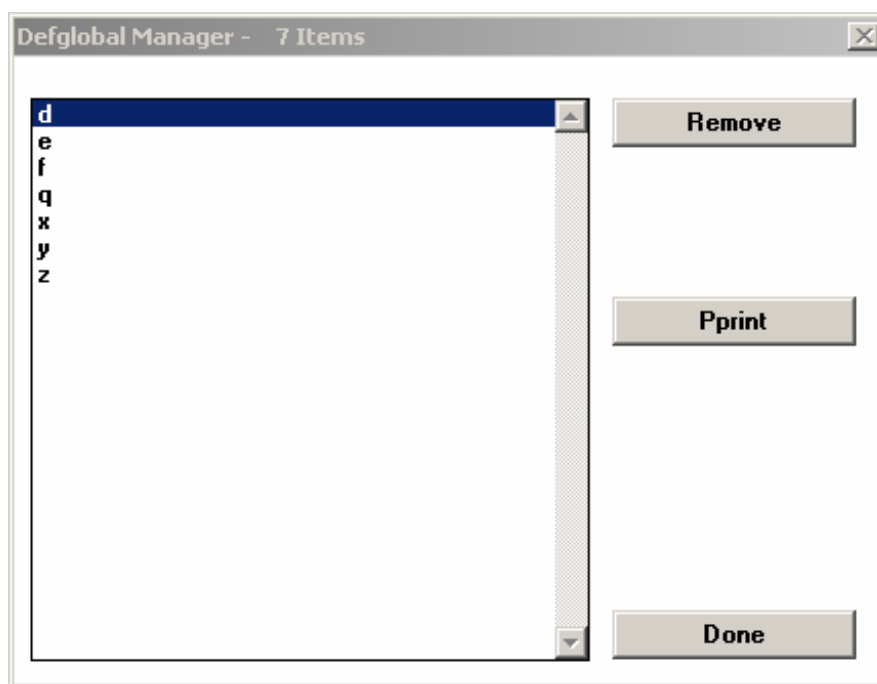


Рис 3. Окно менеджера глобальных переменных

Обратите внимание, что он выводит список глобальных переменных в алфавитном порядке. Общее количество переменных отображается у заголовке окна **Defglobal Manager – 7 Items**. С помощью этого инструмента можно просматривать определение глобальной переменной или удалять ее из системы. Если вы не хотите использовать менеджер глобальных переменных, то удалить созданные ранее глобальные переменные можно с помощью команды *undefglobal*.

При выполнении команды *reset* все глобальные переменные получают начальные значения, определенные в конструкторе. Такое поведение системы можно изменить, для этого в диалоговом окне **Execution Options** сбросьте флажок **Reset Global Variables**.

Вы можете установить режим просмотра изменений значений глобальных переменных. Для этого установите флажок **Globals** в диалоговом окне **Watch Options**, как показано на рис. 4.

В этом случае, например при выполнении команды *reset*, вы увидите результат, приведенный на рис. 5.

Для полноценной работы с глобальными переменными необходимо рассмотреть еще одну важную функцию — *bind*. Эта функция позволяет устанавливать переменным новые значения:

Определение 2. Синтаксис функции *bind*

(bind <имя-переменной> <выражение>*)



Рис 4. Установка режима просмотра изменения глобальных переменных

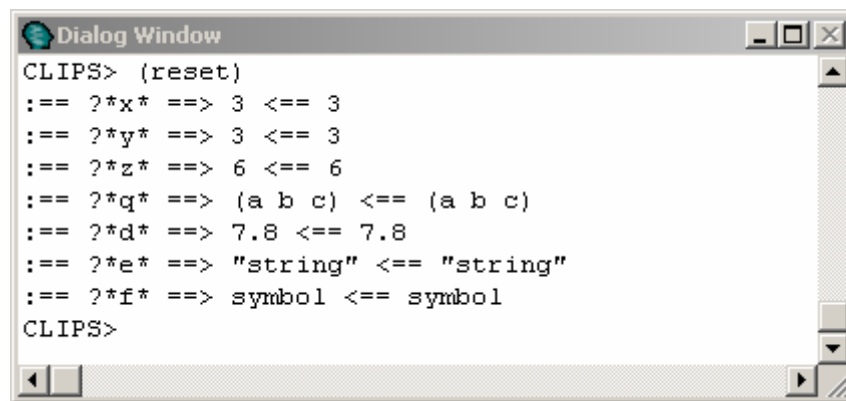


Рис 5. Режим просмотра изменения глобальных переменных

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

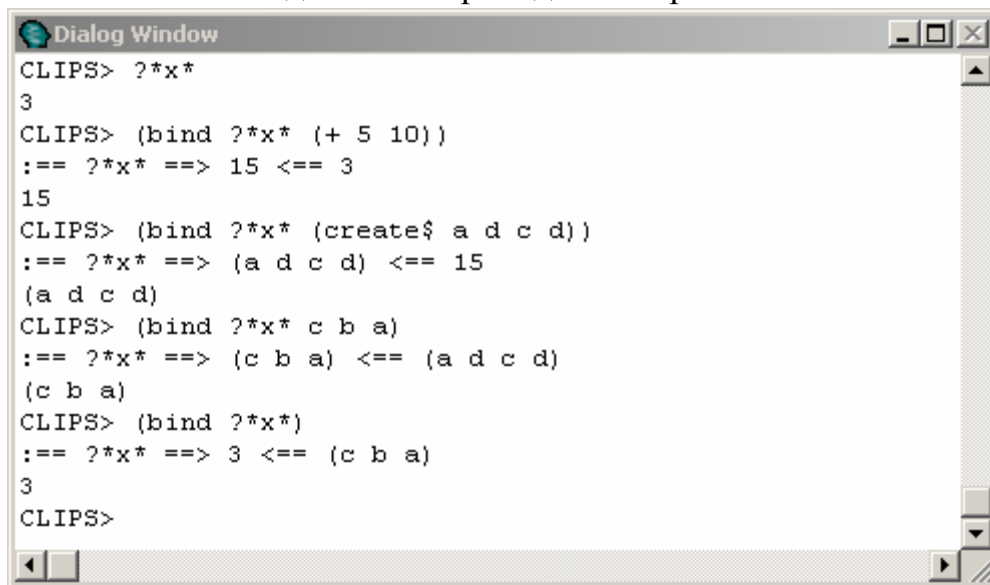
1.3. Функция *bind*

Функция *bind* возвращает значение FALSE в случае, если переменной по какой-то причине не было присвоено никакого значения. В

противном случае Функция возвращает значение, присвоенное переменной.

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

В качестве примера попробуйте присвоить переменной *x* следующие значения: *(+ 5 10)*, *(create\$ a b c d)*, три отдельных выражения *(c)*, *(b)* и *(a)*, а так же не присваивать переменной вообще никакого выражения. Результаты описанных действий приведены на рис. 6.



```
Dialog Window
CLIPS> ?*x*
3
CLIPS> (bind ?*x* (+ 5 10))
:== ?*x* ==> 15 <== 3
15
CLIPS> (bind ?*x* (create$ a d c d))
:== ?*x* ==> (a d c d) <== 15
(a d c d)
CLIPS> (bind ?*x* c b a)
:== ?*x* ==> (c b a) <== (a d c d)
(c b a)
CLIPS> (bind ?*x*)
:== ?*x* ==> 3 <== (c b a)
3
CLIPS>
```

Рис 6. Изменение типа глобальной переменной

Обратите внимание на то, что глобальная переменная *x* в нашем примере постоянно меняла тип своего значения.

2. ПРИЕМЫ РАБОТЫ С ВНЕШНИМИ ФУНКЦИЯМИ

CLIPS поддерживает не только эвристическую парадигму представления знаний (в виде правил), но и процедурную парадигму, используемую в большинстве языков программирования, таких, например, как Pascal или C. Для представления знаний в процедурной парадигме CLIPS предоставляет такие механизмы, как функции.

Функции в CLIPS являются последовательностью действий с заданным именем, возвращающей некоторое значение или выполняющей различные полезные действия (например, вывод информации на экран). В CLIPS существуют *внутренние* и *внешние* функции.

Внутренние функции реализованы средой CLIPS, поэтому их можно использовать в любой момент.

Внешние функции — это функции, написанные пользователем. Внешние функции можно создавать как с помощью среды CLIPS, так и на любых других языках программирования, а затем подключать готовые, откомпилированные исполнимые модули к CLIPS. Для создания новых функций в CLIPS используется конструктор *deffunction*.

2.1. Конструктор *deffunction*

Конструктор *deffunction* позволяет пользователю создавать новые функции непосредственно в среде CLIPS. Способ вызова функций, определенных пользователем, эквивалентен способу вызова внутренних функций CLIPS. Вызов функции осуществляется по имени, заданному пользователю. За именем функции следует список необходимых аргументов, отделенный одним или большим числом пробелов. Вызов функции вместе со списком аргументов должен заключаться в скобки. Последовательность действий определенной с помощью конструктора *deffunction* функции исполняется интерпретатором CLIPS (в отличие от функций, созданных на других языках программирования, которые должны иметь уже готовый исполнимый код).

Синтаксис конструктора *deffunction* включает в себя 5 элементов:

- ☐ имя функции;
- ☐ необязательные комментарии;
- ☐ список из нуля или более параметров;
- ☐ необязательный символ групповых параметров для указания того, что функция может иметь переменное число аргументов;
- ☐ последовательность действий или выражений, которые будут выполнены (вычислены) по порядку в момент вызова функции.

Определение 3. Синтаксис конструктора *deffunction*

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

Функция, создаваемая с помощью конструктора *deffunction*, должна иметь уникальное имя, не совпадающее с именами других внешних и внутренних функций. Функция, созданная с помощью *deffunction*, не может быть перегружена. Конструктор *deffunction* должен быть объявлен до первого использования создаваемой им функции. Исключения составляют только рекурсивные функции.

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

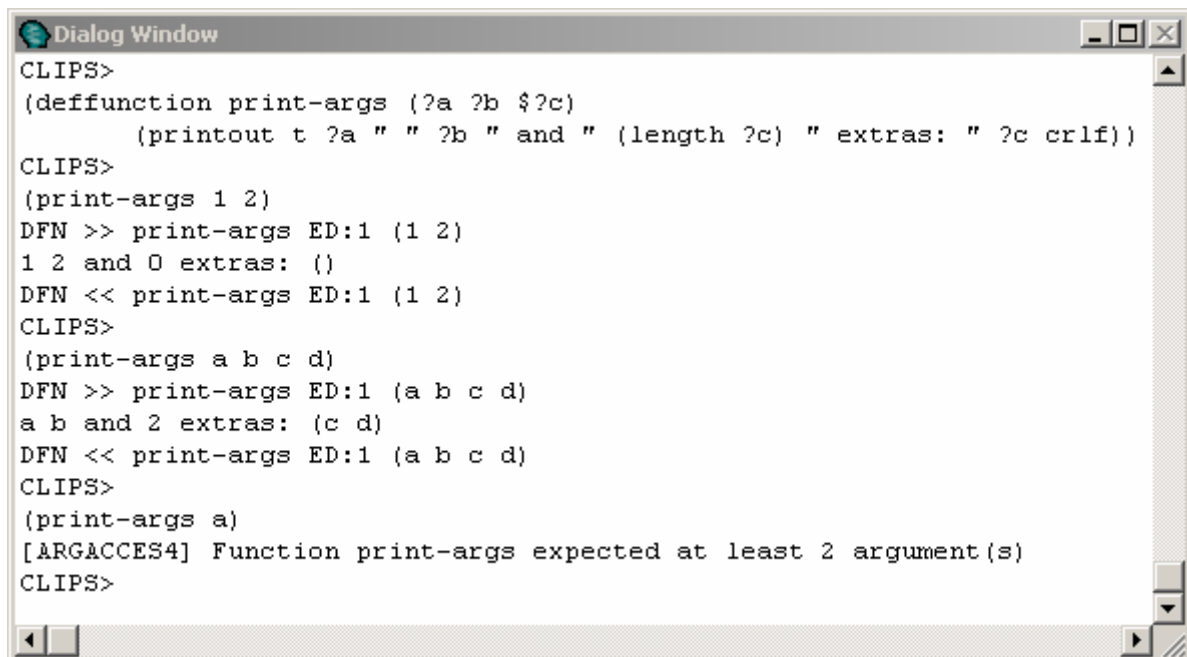
Приведем пример, демонстрирующий описанные выше возможности работы с групповыми параметрами.

Пример 5. Использование группового параметра

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

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

параметре и его содержимое. Результат выполнения данного примера приведен на рис. 7.



```

Dialog Window
CLIPS>
(deffunction print-args (?a ?b $?c)
  (printout t ?a " " ?b " and " (length ?c) " extras: " ?c crlf))
CLIPS>
(print-args 1 2)
DFN >> print-args ED:1 (1 2)
1 2 and 0 extras: ()
DFN << print-args ED:1 (1 2)
CLIPS>
(print-args a b c d)
DFN >> print-args ED:1 (a b c d)
a b and 2 extras: (c d)
DFN << print-args ED:1 (a b c d)
CLIPS>
(print-args a)
[ARGACCES4] Function print-args expected at least 2 argument(s)
CLIPS>

```

Рис 7. Результат работы функции **print-args**

Обратите внимание, что вызов функции с числом параметров, меньшим минимального, приводит к сообщению об ошибке.

При вызове функции интерпретатор CLIPS последовательно выполняет действия в порядке, заданном конструктором. Функция возвращает значение, равное значению, которое вернуло последнее действие или вычисленное выражение. Если последнее действие не вернуло никакого результата, то выполняемая функция также не вернет результата (как в приведенном выше примере). Если функция не выполняет никаких действий, то возвращенное значение равно FALSE. В случае возникновения ошибки при выполнении очередного действия выполнение функции будет прервано и возвращенным значением также будет FALSE.

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

Пример 6. Использование рекурсии для вычисления факториала
(deffunction factorial (?a)
 (if (or (not {integerp ?a}) (< ?a 0)) then

```

        (printout t "Factorial Error!" crlf)
    else
        (if (= ?a 0) then
            1
        else
            (* ?a (factorial (- ?a 1)))))

```

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

Пример 7. Создание взаимно рекурсивных функций

```

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

```

Внимательно следите за рекурсивными вызовами функций, слишком большой уровень рекурсии может привести к переполнению стека памяти. Например приведенный выше пример с функциями *bar* и *foo* приводит к аварийному завершению CLIPS.

Обратите внимание, что на рис. 7 в главном окне CLIPS выводится информация о запуске каждой функции. Для установки этого режима воспользуйтесь диалоговым окном **Watch Options**. Для этого откройте диалоговое окно, выбрав пункт **Watch** из меню **Execution**, и установите флажок **Deffunctions**, как показано на рис. 8.



Рис 8. Установка режима просмотра вызова функций

Этот режим также позволяет просматривать аргументы, которые использовались при каждом конкретном вызове функции.

Так же как и для правил, предопределенных фактов, глобальных переменных Windows-версия CLIPS предоставляет специальный инструмент для работы с функциями — **Deffunction Manager** (Менеджер функций). Для запуска этого инструмента воспользуйтесь пунктом **Deffunction Manager** из меню **Browse**. В случае если в CLIPS не определена ни одна функция, данный пункт меню недоступен. Менеджер функций, отображающий функции, созданные нами, изображен на рис. 9.

Общее количество внешних функций отображается в заголовке окна менеджера — **Deffunction Manager — 4 Items**. С его помощью можно распечатать определение функции, удалить ее, а также установить режим просмотра вызова для отдельно выбранной функции.

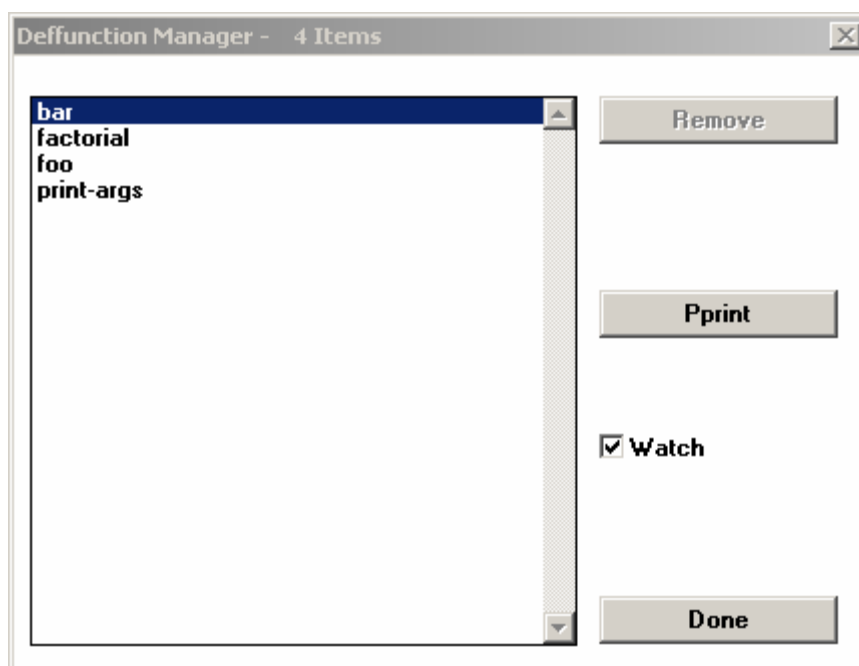


Рис 9. Окно менеджера функций

3. СОЗДАНИЕ РОДОВОЙ ФУНКЦИИ

Родовые функции подобны функциям, созданным с помощью конструктора *deffunction*. Они также могут использоваться для определения нового процедурного кода в CLIPS и могут быть вызваны как любые другие функции. Однако, в отличие от простых, родовые функции являются более мощным средством обработки данных, т.к. они способны выполнять различные наборы действий в зависимости от числа и типа полученных в данный момент аргументов. Родовые функции подобны перегруженным операторам или функциям языка C++. Например, функция + может выполнять как операцию конкатенации строк, так и простое арифметическое сложение чисел.

Родовые функции обычно состоят из нескольких компонентов, называемых *методами*. Каждый метод определяет последовательность действий, выполняющих обработку различных наборов аргументов. Родовая функция, которая имеет более одного метода, называется *перегруженной*. Родовые функции могут содержать как системные методы, так и методы, определенные пользователем. Например, перегруженная функция + состоит из двух методов:

1. неявный метод, являющийся системной функцией, которая обрабатывает арифметическое сложение;

2. явный (определенный пользователем) обработчик сложения строк.

CLIPS не позволяет использовать функции, созданные с помощью конструктора *deffunction*, в качестве методов родовых функций. Конструкторы *deffunction* предоставляют возможность добавления в CLIPS новых функций без использования концепции перегрузки. Родовая функция, имеющая только один метод, по своему поведению идентична функции, созданной с помощью конструктора *deffunction*.

В большинстве случаев методы родовых функций не вызываются напрямую, несмотря на то, что CLIPS предоставляет такую возможность с помощью функции *call-specific-method*.

Процесс самостоятельного распознавания вызова родовой функции с использованием аргументов для поиска и запуска соответствующего метода в CLIPS называется *родовым связыванием*.

3.1. Конструкторы *defgeneric* и *defmethod*

Родовые функции определяются с помощью конструкторов *defgeneric* и *defmethod*.

Родовая функция состоит из заголовка (подобного предварительному объявлению функции) и нескольких методов (число которых теоретически может быть равным нулю). Заголовок родовой функции может быть либо явно определен пользователем, либо не явно объявлен определением метода. Объявление метода состоит из 6 элементов:

- ☐ имя (которое отображает, к какой основной функции относится метод);
- ☐ необязательный индекс;
- ☐ необязательные комментарии;
- ☐ набор ограничений для параметров;
- ☐ необязательный групповой параметр для обработки переменного числа аргументов;
- ☐ последовательность действий или выражений, которые будут выполнены в заданном порядке в момент вызова метода.

Ограничения параметров используются в процессе родового связывания для определения применимости метода к некоторому набору аргументов. Для создания заголовка родовой функции служит конструктор *defgeneric*, а для создания каждого нового метода родовой функции — конструктор *defmethod*.

Определение 4. Синтаксис конструктора *defgeneric*

(defgeneric <имя-функции>

[комментарии])

Определение 5. Синтаксис конструктора *defmethod*

```
(defmethod <имя-функции>
  [<индекс>]
  [<комментарии>]
  (<ограничения-параметра>*)
  [<групповой-параметр>] )
<действие>*)

<ограничения-параметров> ::= <простая-переменная> |
                              (<простая-переменная>
                               <ограничение-по-типу>*
                               [<ограничение-по-запросу>])

<групповой-параметр>      ::= <составная-переменная> |
                              (<составная-переменная>
                               <ограничение-по-типу>*
                               [<ограничение-по-запросу>])

<ограничение-по-типу>     ::= <имя-класса>

<ограничение-по-запросу> ::= <глобальная-переменная> |
                              <вызов-функции>
```

Родовая функция должна быть либо явно объявлена конструктором *defgeneric*, либо одним из своих методов, до того как она будет вызвана из другой функции, правила или обработчика сообщения. Исключения составляют рекурсивные родовые функции.

3.2. Ограничения параметров метода

Каждый параметр метода может быть определен с некоторыми произвольными комплексными ограничениями или без них. Ограничения параметров применяются к аргументам родовой функции во время работы программы для определения того, какой именно метод должен принимать эти аргументы. Параметр может иметь два типа ограничений:

- ☐ ограничение типа
- ☐ ограничение запросом.

Ограничение типа содержит классы аргументов, которые может принимать параметр.

Ограничение запросом является определенным пользователем условным выражением, которое должно удовлетвориться для аргументов в момент вызова функции.

Ограничение типа позволяет пользователю определить список типов (классов), один из которых должен соответствовать (или являться суперклассом) аргументу родовой функции. В качестве ограничения типа будут доступны следующие типы (классы): OBJECT, PRIMITIVE, LEXEME, SYMBOL, STRING, NUMBER, INTEGER, FLOAT, MULTIFIELD, FACT-ADDRESS И EXTERNAL-ADDRESS, INSTANCE, INSTANCE-ADDRESS, INSTANCE-NAME, USER, INITIAL-OBJECT, а также любой определенный пользователем класс. Классы, заданные в ограничении типа, должны быть определены до определения приоритета метода. CLIPS не поддерживает избыточность в списке ограничений типов аргументов методов. Например, для представленного ниже метода ограничения типов аргументов избыточны, т. к. класс INTEGER — подкласс NUMBER.

Пример 8. Избыточные ограничения типов

```
(defmethod foo ((?a INTEGER NUMBER)) )
```

Если ограничение типа удовлетворяется для некоторого аргумента, то к нему будет применено ограничение запросом (если оно задано). Ограничение запросом должно быть либо глобальной переменной, либо вызовом функции. CLIPS вычисляет заданное выражение, и если полученный результат не равен FALSE — ограничение полагается удовлетворенным.

Так как ограничения запросом вычисляются каждый раз при поиске соответствующего метода, они не могут использоваться для произведения какого-нибудь побочного действия, потому что вычисляемое ограничение может принадлежать методу, неподходящему к данной конкретной ситуации.

Поскольку все ограничения просматриваются слева направо, запрос с несколькими параметрами должен быть записан после ограничений типов всех используемых параметров. Этим правилом обеспечивается условие удовлетворения ограничений типов всех необходимых параметров. Например, метод из примера 9 не вычисляет ограничение запросом до тех пор, пока не удовлетворятся два соответствующих ограничения типа.

Пример 9. Использование ограничения запросом с двумя параметрами

```
(defmethod foo ((?a INTEGER) (?b INTEGER (> ?a ?b))))
```

Если аргумент удовлетворяет всем своим ограничениям, то считается, что он применим для данного метода. Если все аргументы

родовой функции применимы к ограничениям метода, метод полагается применимым для данного набора аргументов. В случае если существует более одного метода, применимого для некоторого набора аргументов, процесс родовой связывания определяет некоторый упорядоченный список этих методов и использует первый метод из этого списка. Для создания списка служит приоритет методов.

Пример 10. Перегрузка системной функции +
(defmethod + ((?a STRING) (?b STRING))
 (str-cat ?a ?b))

(+ 1 2)
(+ "foo" "bar")
(+ "foo" "bar" "woz")

В примере 10 первое обращение к родовой функции + вызовет выполнение системной функции + — неявный метод, выполняющий арифметическое сложение. Второй вызов приведет к выполнению явного метода родовой функции, осуществляющего конкатенацию строк, т.к. оба аргумента являются строками. Третий вызов сгенерирует ошибку, поскольку явный метод для конкатенации строк принимает только два аргумента, а неявный метод для арифметического сложения не принимает строковые аргументы вообще (см. рис.10).

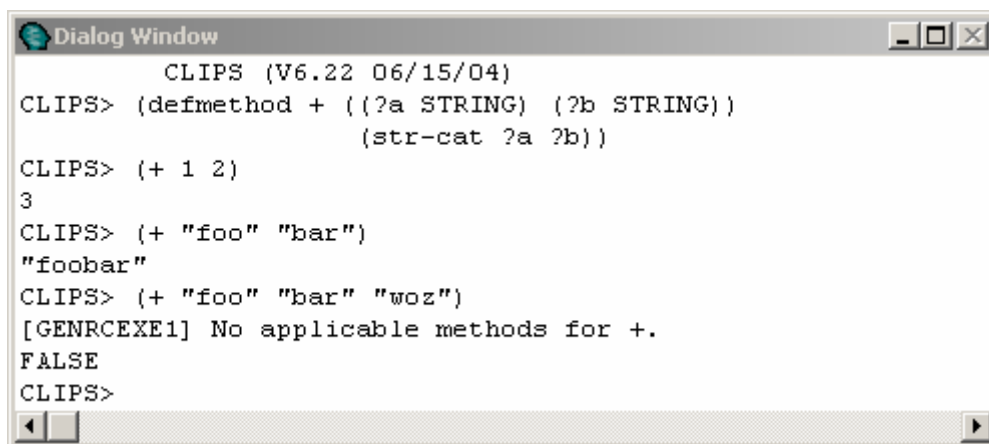


Рис 10. Перегрузка системной функции +

3.3. Групповой параметр

В зависимости от того, задан ли групповой параметр, метод может принимать точное число параметров или число параметров не меньше, чем некоторое заданное. Для работы с групповым параметром могут

использоваться стандартные функции CLIPS, предназначенные для работы с составными полями, такие как *length* и *nth*. Определение метода может содержать только один групповой параметр.

Ограничения типом и запросом могут применяться к аргументам, сгруппированным в групповом параметре, аналогично тому, как они употребляются с основными параметрами метода. Такие ограничения задаются для каждого отдельного поля результирующего составного значения (а не для всего значения). Выражение, содержащее групповой символ, может быть применено в запросе.

Дополнительно в запросе может быть использована специальная переменная *?current-argument* для ссылки на отдельные аргументы, объединенные групповым символом. Это переменная существует только в ограничении запросом и не имеет значения в теле метода. Метод из примера 11 иллюстрирует версию функции *+*, которая находит полусумму любого количества четных целых чисел.

Пример 11. Еще один вариант функции *+*

```
(defmethod +  
  (($?any INTEGER (evenp ?current-argument)))  
  (div (call-next-method) 2))
```

Ограничения по типу и запросу для группового параметра применяются к каждому аргументу, сгруппированному групповым символом, даже если для проверки используются функции для работы с составным значением. Таким образом, в примере 12 функции *>* и *length\$* вызываются 3 раза для каждого из трех аргументов.

Пример 12. Ограничение запросом для группового поля

```
(defmethod foo  
  (($?any (> (length$ ?any) 2)))  
  TRUE)  
(foo 1 red 3)
```

Кроме того, если у метода нет обязательных параметров (как в предыдущем примере) и функция вызвана без аргументов, заданные ограничения группового параметра не рассматриваются. Например, метод из примера 12 можно применить к следующему вызову родовой функции (метод успешно вызовется и вернет значение TRUE): *(foo)*.

Как правило, ограничения запросом применяют ко всему групповому параметру для проверки *мощности* (числа аргументов, переданных методу). В таких случаях первое поле группового аргумента выносится в

обязательный параметр (если это возможно). Приведенный выше пример можно усовершенствовать.

Пример 13. Улучшенная версия функции *foo*

```
(defmethod foo
  ((?arg (> (length$ ?any) 1) ) $?any)
  TRUE)
```

Теперь попытка вызова функции (*foo*) без параметров закончится ошибкой.

3.4. Родовое связывание

В момент вызова родовой функции CLIPS выбирает метод с наивысшим приоритетом, для которого удовлетворяются все ограничения параметров. Этот метод выполняется, и его значение возвращается как значение родовой функции. Такой процесс называется *родовым связыванием*.

3.4.1. Применимость методов

Явный (определенный пользователем) метод применим к вызову родовой функции при следующих трех условиях:

- ☐ имя совпадает с именем родовой функции;
- ☐ метод принимает не меньше аргументов, чем родовая функция;
- ☐ каждый аргумент родовой функции удовлетворяет соответствующим ограничениям параметров метода.

Ограничения метода рассматриваются слева направо. Как только будет найдено одно ограничение, не удовлетворяющее некоторому параметру, метод забраковывается, и оставшиеся ограничения не рассматриваются.

При перегрузке стандартной системной функции CLIPS создает неявный метод с определением соответствующей системной функции. Этот неявный метод получает ограничения аргументов благодаря вызову внутренней системной функции **DefineFunction2**. Строка с соответствующими ограничениями также может быть получена с помощью функции *get-function-restriction*.

Определение неявного метода можно просмотреть функциями *list-defmethods* или *get-method-restrictions*.

3.4.2. Приоритет методов

Когда два или более метода применимы к некоторому вызову родовой функции, CLIPS выполняет метод с наивысшим приоритетом. Приоритет метода определяется в момент его создания. Для того чтобы просмотреть приоритеты существующих методов, можно воспользоваться функцией *list-defmethods*.

Приоритет определяется сравнением ограничений параметров для пар методов. Метод с большим числом заданных ограничений параметров имеет больший приоритет. Кроме того, CLIPS учитывает диапазон значений, задаваемых приоритетом. Например, метод, который требует наличие типа INTEGER, для некоторого аргумента имеет больший приоритет, чем метод, который требует для этого аргумента тип NUMBER. Ниже приведены правила, используемые CLIPS для определения приоритета между двумя методами.

1. Последовательно, слева направо сравниваются ограничения параметров обоих методов, (первое ограничение параметра первого метода сравнивается с первым ограничением параметра второго метода и т.д.). Сравнение между этими парами ограничений параметров двух методов определяет приоритет между двумя методами. Сравнение прекращается, как только будет найдена первая пара ограничений, однозначно определяющая метод с более высоким приоритетом. Для сравнения пар ограничений параметров применяются следующие правила в указанном порядке:
 - ☐ обязательные параметры имеют более высокий приоритет, чем групповой параметр;
 - ☐ более строгие ограничения типа имеют более высокий приоритет. Например, класс имеет больший приоритет, чем его суперкласс;
 - ☐ параметр с ограничением запроса имеет приоритет выше, чем параметр, который его не имеет.
2. Метод с большим числом постоянных параметров имеет больший приоритет.
3. Метод без групповых параметров имеет более высокий приоритет, чем метод с групповыми параметрами.
4. Если метод определен раньше другого, то первый метод имеет более высокий приоритет.

Рассмотрим процесс определения приоритета методов на нескольких примерах.

Пример 14. Перегрузка функции +

```

; Системная функция '+' является неявным методом ;#1
; Данный метод имеет следующее определение:
; (defmethod + ((?a NUMBER) (?b NUMBER) (S?rest NUMBER)))
(defmethod + ( (?a NUMBER) (?b INTEGER))) ;#2
(defmethod + ((?a INTEGER) (?b INTEGER))) ;#3
(defmethod + ' ((?a INTEGER) (?b NUMBER))) ;#4
(defmethod + ((?a NUMBER) (?b NUMBER)
              ($?rest PRIMITIVE))) ;#5
(defmethod + ((?a NUMBER) (?b INTEGER (> ?b 2))))
;#6
(defmethod + ((?a INTEGER (> ?a 2))
              (?b INTEGER (> ?b 3)))) ;#7
(defmethod + ((?a INTEGER (> ?a 2)) (?b NUMBER))) ;#8

```

Приоритет методов, в приведенном выше примере, будет следующим: #7, #8, #3, #4, #6, #2, #1, #5. В данной ситуации для определения приоритета методы могут быть сразу разбиты на три группы по уменьшению приоритета с помощью ограничения, заданного для их первого параметра:

- методы, которые имеют ограничение и типом и запросом для класса INTEGER (#7, #8);
- методы, которые имеют ограничение типа для класса INTEGER (#3, #4);
- методы, которые имеют ограничение типа для класса NUMBER (#1, #2, #5, #6).

Методы первой группы имеют более высокий приоритет, чем методы второй группы, т. к. ограничение параметров типом и запросом имеет более высокий приоритет, чем ограничение только типом. Вторая группа имеет более высокий приоритет, чем третья группа, поскольку INTEGER является подклассом NUMBER. Поэтому порядок методов можно представить так: (#7, #8) (#3, #4) (#1, #2, #5, #6).

Следующим шагом будет определение приоритета между методами в выделенных нами группах, учитывая ограничение по второму параметру. Метод #7 имеет больший приоритет, чем #8, т.к. INTEGER является подклассом NUMBER. Метод #3 имеет больший приоритет по сравнению с #4 по той же причине. Методы #6 и #2 имеют приоритет больший, чем методы #1 и #5, т.к. INTEGER является подклассом NUMBER. Метод #6 имеет больший приоритет по сравнению с #2, т. к. #6 имеет ограничение запросом, а #2 нет. Таким образом, порядок методов примет следующий вид: #7, #8, #3, #4, #6 #2, (#1, #5)

Ограничение по групповому аргументу дает методу #1 (системной функции + – неявному методу родовой функции +) больший приоритет по сравнению с методом #5, т.к. NUMBER является подклассом PRIMITIVE. Итак, окончательным порядком расстановки методов по приоритету будет такой: #7, #8, #3, #4, #6, #2, #1, #5.

Пример 15. Определение приоритета

```
(defmethod foo ((?a NUMBER STRING)) ; #1  
(defmethod foo ((?a INTEGER LEXEME)) ; #2
```

Несмотря на то, что класс STRING является подклассом класса LEXEME, порядок методов будет #2, #1, т.к. INTEGER является подклассом, а пара NUMBER/INTEGER находится левее пары STRING/LEXEME в списке классов.

Пример 16. Определение приоритета

```
(defmethod foo ((?a MULTIFIELD STRING)) ; #1  
(defmethod foo ((?a LEXEME)) ; #2
```

Порядок методов из примера 16 будет следующим: #2, #1. Такой порядок определяется тем фактом, что классы первой пары ограничений типов — MULTIFIELD/LEXEME — не связаны, а метод #2 имеет более короткий список классов.

Пример 17. Определение приоритета

```
(defmethod foo ((?a INTEGER LEXEME)) ; #1  
(defmethod foo ((?a STRING NUMBER)) ; #2
```

В данном примере обе пары классов (INTEGER/STRING и LEXEME/NUMBER) несвязаны. Кроме того, списки классов ограничений имеют одинаковую длину. Таким образом, приоритет будет установлен по порядку создания следующих методов: #1, #2.

3.4.3. Скрытые методы

Если один из методов родовой функции вызывается другим, то такой метод называется *скрытым*. Обычно, только один метод должен быть применим к конкретному вызову родовой функции. Если для данного вызова существует больше одного применимого метода, родовое связывание выполнит метод с наивысшим приоритетом. Такой подход называется *декларативным методом родового связывания*.

Однако с помощью функций *call-next-method* и *override-next-method* метод родовой функции может вызвать некоторый другой метод данной родовой функции (скрыть вызов). Такой подход называется *императивным*

методом (после вызова некоторого метода он играет роль родового связывания).

Не рекомендуется использовать данный подход без крайней необходимости. В большинстве случаев обработку вызова с заданным набором аргументов должен осуществлять только один метод.

Помимо функций *call-next-method* и *override-next-method* для реализации императивного подхода можно использовать функцию *call-specific-method* для перегрузки установленного приоритета метода.

4. ВИЗУАЛЬНЫЕ ИНСТРУМЕНТЫ ДЛЯ РАБОТЫ С РОДОВЫМИ ФУНКЦИЯМИ

Рассмотрим визуальные инструменты, которые предоставляет CLIPS для работы с родовыми функциями. Для работы с родовыми функциями Windows-версия среды CLIPS предоставляет инструмент — **Defgeneric Manager** (Менеджер родовых функций). Для его запуска выберите пункт **Defgeneric Manager** в меню **Browse**. Соответствующий пункт в меню недоступен, если в данный момент в среде не определена ни одна родовая функция. Общий вид менеджера представлен на рис. 11.

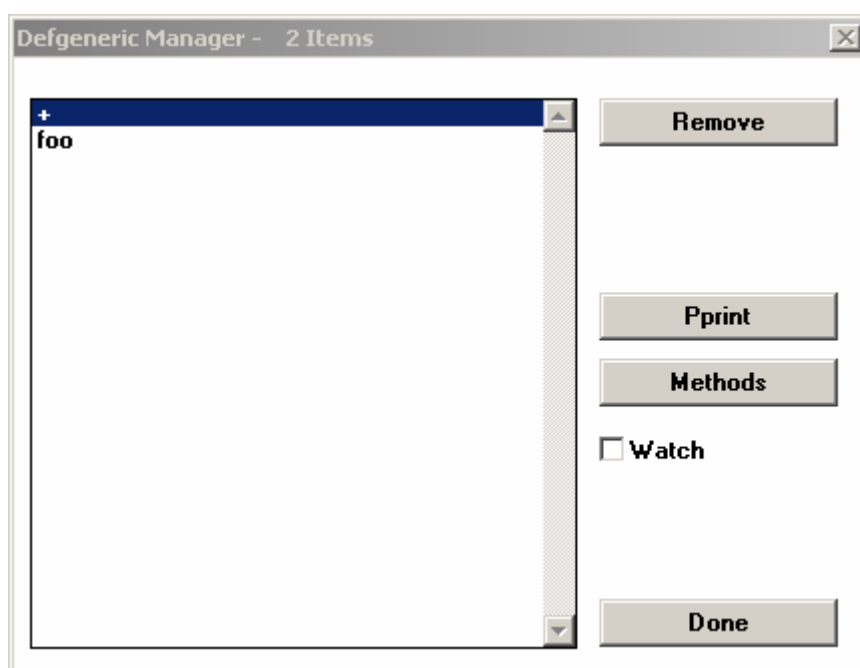


Рис 11. Окно менеджера родовых функций

Общее количество родовых функций отображается в заголовке окна менеджера — **Defgeneric Manager – 2 Items**. С помощью этого

инструмента вы можете удалить родовую функцию из системы (кнопка **Remove**), вывести на экран ее определение (кнопка **Pprint**), установить режим просмотра вызова отдельной функции и вызвать менеджер методов для заданной функции (кнопка **Methods**).

Учтите, что удаление родовой функции приводит к удалению всех ее методов.

Для тренировки использования родовых функций и менеджера родовых функций очистите CLIPS и добавьте в него методы, приведенные в примере 8.

Пример 18. Перегрузка функции +

```
(defmethod + ((?a INTEGER (> ?a 0)) (?b INTEGER (> ?b 0)))  
  (call-next-method))  
(defmethod + ((?a INTEGER) (?b FLOAT))  
  (call-next-method))  
(defmethod + ((?a FLOAT) (?b FLOAT))  
  (call-next-method))  
(defmethod + ((?a STRING) (?b STRING))  
  (str-cat ?a ?b))
```

Обратите внимание на реализацию методов для сложения чисел. После проверки своих аргументов они просто вызывают системную функцию +. Если бы мы вместо вызова (*call-next-method*) использовали системную функцию + напрямую (+ ?a ?b), то получили бы бесконечную рекурсию, которая привела бы к переполнению стека и аварийному завершению программы.

Попробуйте несколько раз вызвать функцию + с различными аргументами:

Пример 19. Тестирование родовой функции +

```
(+ "Hello " "World")  
(+ 1 3)  
(+ 1 3.5)  
(+ 1.5 3)  
(+ 1 -3)  
(+ 1.5 3.0)  
(+ 1.5 3.0 5.0)  
(+ "Hello" "World" "!!!")
```

Полученный результат должен соответствовать приведенному на рис. 12.

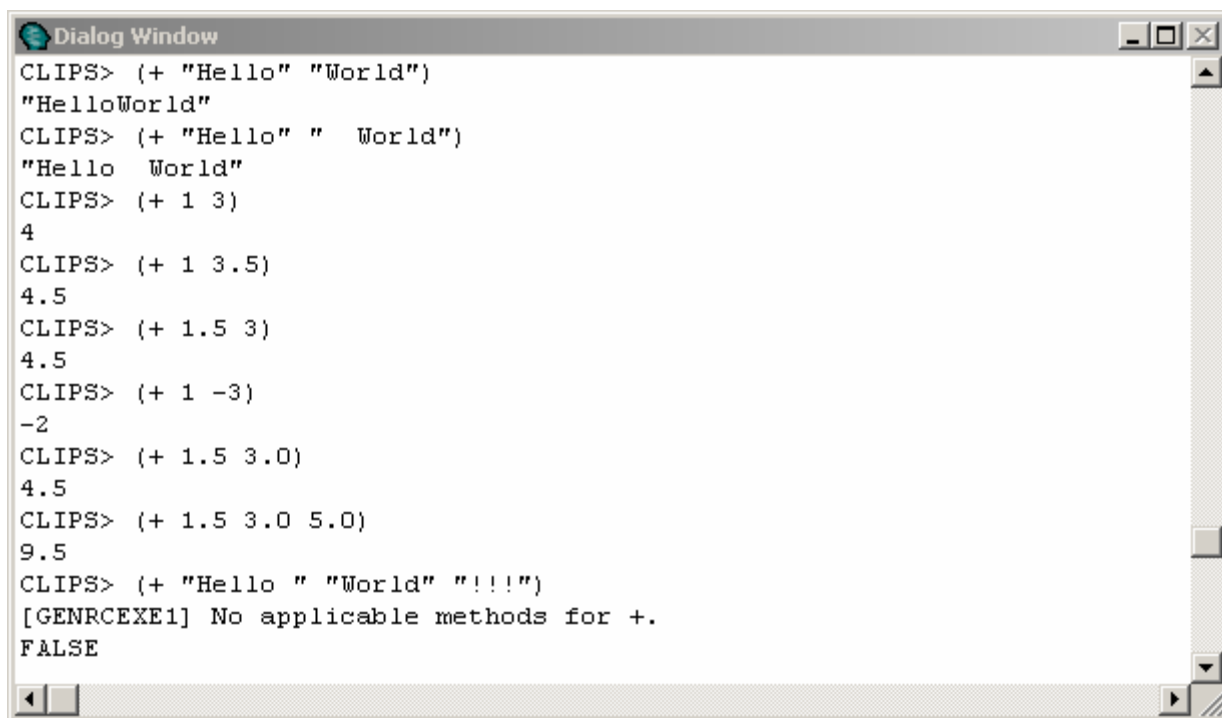


Рис 12. Результаты тестирования родовой функции

Обратите внимание, что для вызовов (+ 1.5 3), (+ 1 -3), (+ 1.5 3.0 5.0), (+ 1 3.5 4) применяется вызов системной функции +, т. к. мы не определили методов, способных принять такие аргументы, но, тем не менее, мы получили корректные ответы. Родовое связывание не смогло подобрать метод, применимый к вызову (+ "Hello " "world" "!!!") (наша функция для конкатенации строк принимает строго два аргумента), поэтому мы получили соответствующее сообщение об ошибке.

Установите режим отображения вызова родовой функции с помощью менеджера и попробуйте еще раз повторить вызовы, приведенные выше. Обратите внимание на сообщения о вызовах родовой функции. на сообщения о вызовах родовой функции.

Defmethod-Handler Manager (Менеджер методов родовой функции) — один инструмент, предоставляемый CLIPS. Внешний вид этого инструмента представлен на рис. 12. Этот инструмент выводит на экран список методов родовой функции, указанной менеджером родовых функций. Список методов сортируется по приоритету, установленному для этих методов. Общее количество методов заданной родовой функции отображается в заголовке окна менеджера — **Defmethod-Handler Manager** — **5 Items** (in precedence order).

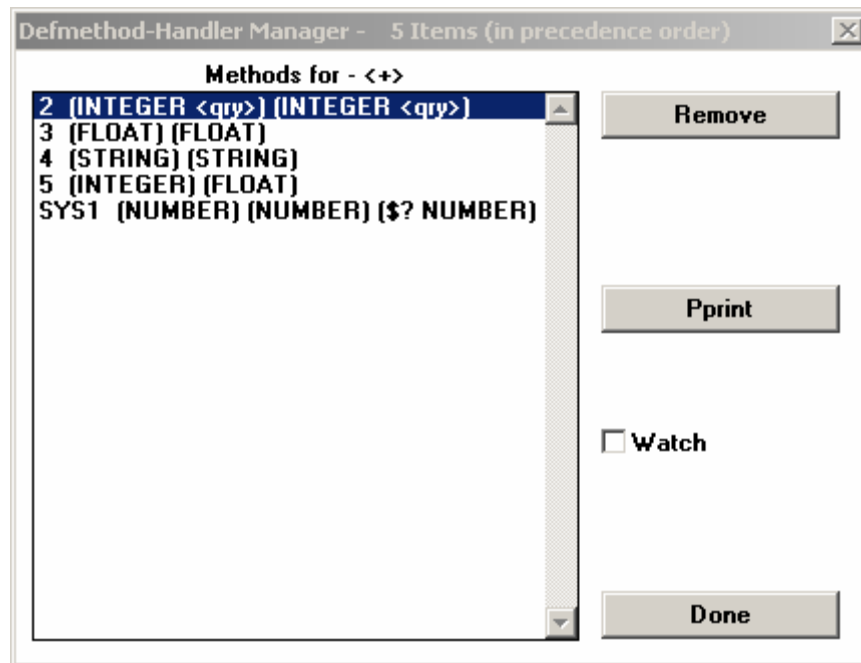


Рис 13. Окно менеджера методов родовой функции

С помощью менеджера методов вы можете удалить некоторый метод (кнопка **Remove**), вывести на экран его определение (кнопка **Pprint**) или установить режим просмотра вызовов отдельного метода. Обратите внимание, что метод, неявно определенный системой, например метод, представляющий системную функцию +, не может быть удален.

Снимите установку вывода сообщений о вызове родовой функции + и установите вывод сообщений о вызове методов с помощью менеджера методов. Выполните следующие вызовы:

Пример 20. Тестирование родовой функции +

(+ "Hello" "World")

(+ 1 3)

(+ 1 3.5)

(+ 1 -3)

Результат этих действий представлен на рис. 14.

Обратите внимание, что, в случае получения сообщений о вызове конкретного метода родовой функции, мы можем получить информацию о том, какой именно метод обработал полученный вызов.

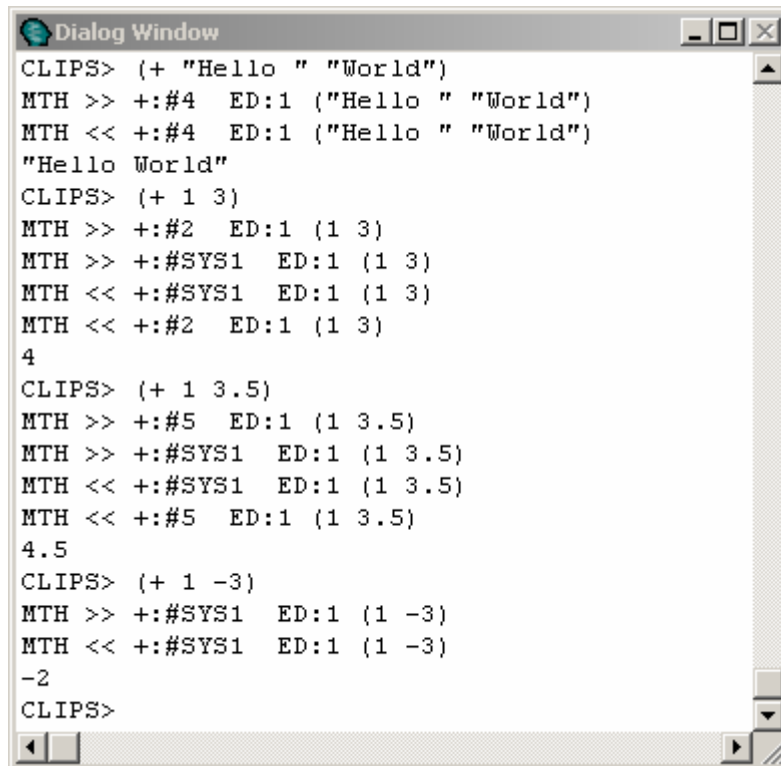


Рис 14. Результаты тестирования родовой функции +

В случае если вы хотите установить режим просмотра вызовов всех методов или всех родовых функций, воспользуйтесь диалоговым окном **Watch Options** из меню **Execution**. Установите флажки в полях **Generic Functions** или/и **Methods**, как показано на рис. 15.



Рис 15. Установка режима отображения вызовов родовых функций и методов

Примечание: Вызов родовой функции может быть на 15—20% медленнее вызова обычной функции. Поэтому не используйте родовые функции в операциях, для которых время критично. Например, не вызывайте родовую функцию в цикле, если это возможно.

ЗАДАНИЕ

Используя конструкторы *deffunction*, *defgeneric* и *defmethod* создайте пользовательскую функцию, родовую пользовательскую функцию и пять различных методов к ней.

5. КОНТРОЛЬНЫЕ ВОПРОСЫ:

1. Какие механизмы используются CLIPS для представления знаний в процедурной парадигме?
2. Как с помощью конструктора *defglobal* в среде CLIPS могут быть объявлены глобальные переменные?
3. Что позволяет функция *bind* ?
4. Какие стандартные функции CLIPS, предназначенные для работы с составными полями?
5. Особенности использования внутренних и внешних функции в CLIPS?
6. Как создать новые функции в CLIPS, используя конструктор *deffunction*?
7. Приведите синтаксис конструктора *deffunction*.
8. Само- и взаимно рекурсивные функции.
9. При помощи каких конструкторов определяются родовые функции?
10. Дайте понятие метода при использовании родовых функций?
11. Дайте понятие перегруженной функции. Из каких методов состоит перегруженная функция +?
12. Что представляет собой родовое связывание?