Министерство науки и высшего образования Российской Федерации

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

# Общие функции и система ввода-вывода в среде CLIPS

Методические указания по выполнению лабораторной работы

Калуга, 2024

**Содержание**

## Цель и задачи работы, требования к результатам её выполнения

Целью выполнения лабораторной работы является формирование практических навыков работы с функциями в среде CLIPS.

Основными задачами выполнения лабораторной работы являются:

1. изучить основные математические функции;
2. получить навыки работы с функциями системы ввода-вывода;
3. научиться работать со списками и строками;
4. получить навыки работы по созданию собственных функций.

Результатами работы являются:

- созданные в среде CLIPS функции;
- сохранённые в файлах скрипты, тестовые входные данные и полученные выходные данные;
- подготовленный отчёт.

**Краткая характеристика объекта изучения, исследования**

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. Функции в CLIPS являются последовательностью действий с заданным именем, возвращающей некоторое значение или выполняющей различные полезные действия (например, вывод информации). 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 исполняемые модули к CLIPS. Для создания новых функций в CLIPS используется конструктор (`deffunction`), описанный далее.

**Создание функций**

Конструктор deffunction позволяет пользователю создавать новые функции собственно в среде CLIPS. Способ вызова функций, определенных пользователем, эквивалентен способу вызова внутренних функций CLIPS.

Вызов функции осуществляется по имени, заданному пользователем. 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`) [5, с. 93, раздел 7] функции исполняется интерпретатором CLIPS (в отличие от функций, созданных на других языка программирования, которые должны иметь уже готовый исполняемый 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.

Синтаксис конструктора `deffunction`

```
(deffunction <name> [<comment>]
  (<regular-parameter>* [<wildcard-parameter>])
  <action>*
)

<regular-parameter> ::= <single-field-variable>

<wildcard-parameter> ::= <multifield-variable>
```

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. Приведём пример создания функции, вычисляющей длину гипотенузы прямоугольника, а также правила, содержащего в себе вызов данной функции.

Пример использования конструктора **deffunction**

```
CLIPS> (deffunction hypotenuse (?a ?b)
  (sqrt (+ (* ?a ?a) (* ?b ?b)))
)
CLIPS> (defrule calculate-hypotenuse
  (dimensions ?base ?height)
  =>
  (println "Hypotenuse = " (hypotenuse ?base ?height))
)
CLIPS> (assert (dimensions 3 4))
<Fact-1>
CLIPS> (run)
Hypotenuse = 5.0
CLIPS>
```

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, которые должны быть переданы функции при её вызове. В действиях функции можно ссылаться на какие-то 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. 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.

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, например функция, вычисляющая факториал (в примере используется конструкция (**if then else**)):

```
(deffunction factorial (?a)
  (if
    (or
      (not (integerp ?a))
      (< ?a 0)
    )
  then
    (println "Factorial Error!")
  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. Для демонстрации взаимной рекурсии приведём следующий пример:

```
(deffunction a())
(deffunction b()
  (a)
)
(deffunction a()
  (println "nothing")
)
```

**Работа с функциями**

Для просмотра (тела) функции, находящейся в памяти, используют команду (`ppdeffunction`) [5, с. 324, раздел 13.9.1]. Она которая не возвращает никакого значения, а лишь выводит текст функции на экран.

Синтаксис функции `ppdeffunction`

```
(ppdeffunction <deffunction-name> [<logical-name>])
```

Для просмотра списка функций, находящихся в памяти, используют команду (`list-deffunctions`) [5, с. 324, раздел 13.9.2]. Она не возвращает никакого значения, а лишь выводит на экран имена всех функций, находящихся в памяти.

Синтаксис функции `list-deffunction`

```
(list-deffunctions)
```

Для удаления функции из памяти используют команду (`undeffunction`) [5, с. 325, раздел 13.9.3].

Синтаксис функции `undeffunction`

```
(undeffunction <deffunction-name>)
```

Если вместо параметра `<deffunction-name>` использовать символ «*», то из памяти удалятся все функции, находившиеся там к моменту удаления (конечно, если в рабочей памяти

нет функции с именем «*»). Данная команда не возвращает никакого значения, и ею могут быть удалены любые функции, находящиеся в данный момент в рабочей памяти.

**Методы**

Для определения функций с большими возможностями используют (**defgeneric**) и (**defmethod**) [5, с. 97–102, раздел 8.4]. Функции, определённые таким образом, называют методами. (**defgeneric**) и (**defmethod**) имеют ряд преимуществ по сравнению с (**deffunction**):

- возможность перегрузки функций (в том числе стандартных);
- явное задание типов аргументов и возможность наложения на них дополнительных ограничений.

Но помимо преимуществ имеют место и следующие недостатки:

- снижение производительности при перегрузке функций на 15-20%;
- сложный синтаксис.

Синтаксис конструктора **defgeneric**

```
(defgeneric <name> [<comment>])
```

Синтаксис конструктора **defmethod**

```
(defmethod <name> [<index>] [<comment>]
  (<parameter-restriction>* [<wildcard-parameter-restriction>])
  <action>*
)

<parameter-restriction> ::=
  <single-field-variable> |
  (<single-field-variable> <type>* [<query>])

<wildcard-parameter-restriction> ::=
  <multifield-variable> |
  (<multifield-variable> <type>* [<query>])

<type> ::= <class-name>

<query> ::= <global-variable> | <function-call>
```

Название прототипа функции должно соответствовать названию методов. Прототип функции необходимо вводить только при опережающем описании и перегрузке. В других случаях в этом нет необходимости (неявно прототип будет введен в систему автоматически). При прямом рекурсивном вызове в явном введении прототипа также нет необходимости.

Индекс нового метода будет автоматически увеличиваться при перегрузке (по умолчанию равен 1).

Пример использования функции **defmethod**

```
(defmethod > ((?a STRING) (?b STRING))
  (> (str-compare ?a ?b) 0)
)
```

Если два и более перегруженных метода претендуют на обработку функционального вызова, предпочтение отдается методу, предоставляющему большие ограничения на аргументы в порядке их перечисления. Продемонстрируем данную ситуацию на примере:

```
; Системный оператор '+' является начальным методом
; Его системное описание таково:
; #1
(defmethod + ((?a NUMBER) (?b NUMBER) ($?rest NUMBER)))
; #2
(defmethod + ((?a NUMBER) (?b INTEGER)))
; #3
(defmethod + ((?a INTEGER) (?b INTEGER)))
; #4
(defmethod + ((?a INTEGER) (?b NUMBER)))
; #5
(defmethod + ((?a NUMBER) (?b NUMBER) ($?rest NUMBER SYMBOL)))
; #6
(defmethod + ((?a NUMBER) (?b INTEGER (> ?b 2))))
; #7
(defmethod + ((?a INTEGER (> ?a 2)) (?b INTEGER (> ?b 3))))
; #8
(defmethod + ((?a INTEGER (> ?a 2)) (?b NUMBER)))
```

Приоритет методов при вызове функции «+» будет следующим (в порядке убывания): #7; #8; #3; #4; #6; #2; #1; #5.

Для вызова другого перегруженного метода можно пользоваться командой (**call-next-method**) [5, с. 271–272, раздел 12.15.6].

Синтаксис функции **call-next-method**

```
(call-next-method)
```

Для просмотра прототипа, находящегося в памяти, используют команду (**ppdefgeneric**) [5, с. 325, раздел 13.10.1].

Синтаксис функции **ppdefgeneric**

```
(ppdefgeneric <generic-function-name> [<logical-name>])
```

Для просмотра списка прототипов, находящихся в памяти, используют команду (**list-defgenerics**) [5, с. 326, раздел 13.10.3].

Синтаксис функции **list-defgenerics**

```
(list-defgenerics [<module-name>])
```

Для удаления прототипа из памяти используют команду (**undefgeneric**) [5, с. 326, раздел 13.10.5].

Синтаксис функции **undefgeneric**

```
(undefgeneric <generic-function-name> | *)
```

Для просмотра списка методов, подходящих для выполнения данного функционального вызова, можно использовать функцию (**preview-generic**) [5, с. 327–328, раздел 13.10.7].

Синтаксис функции **preview-generic**

```
(preview-generic <generic-function-name> <expression>*)
```

Пример использования функции `preview-generic`

```
(preview-generic (> "duck1" "duck2"))
```

Для просмотра (тела) метода, находящейся в памяти, используют команду (`ppdefmethod`) [5, с. 325, раздел 13.10.2].

Синтаксис функции `ppdefmethod`

```
(ppdefmethod <generic-function-name> <index> [<logical-name>])
```

Для просмотра списка методов, находящихся в памяти, используют команду (`list-defmethods`) [5, с. 325, раздел 13.10.4].

Синтаксис функции `list-defmethods`

```
(list-defmethods [<generic-function-name>])
```

Для удаления метода из памяти можно использовать команду (`undefmethod`) [5, с. 326–327, раздел 13.10.6].

Синтаксис функции `undefmethod`

```
(undefmethod <generic-function-name> <index>)
```

В остальном команда (`defmethod`) сходна с (`deffunction`).

### Система ввода-вывода информации

Система ввода-вывода CLIPS базируется на концепции логических имен (сходной с потоками), присваиваемых устройствам (физическим и логическим). Логические имена могут быть символьного, строкового или числового типа. Ряд имен зарезервирован (табл. 1).

Таблица 1. Список зарезервированных имён

| Имя | Описание |
|---|---|
| `stdin` | Имя, используемое по умолчанию в операциях ввода (*read*, *readln*). Аналогом является указание символа «t» |
| `stdout` | Имя, используемое в операциях вывода. Аналогом является указание символа «t» |
| `wclips` | Имя, используемое командной строкой CLIPS |
| `wdialog` | Имя для работы с сообщениями |
| `wdisplay` | Имя, используемое при отображении информации CLIPS (факты, правила) |
| `werror` | Имя, используемое при обработке ошибок |
| `wwarning` | Имя, используемое при обработке предупреждений |
| `wtrace` | Имя, используемое при просмотре отладочной информации (`watch` и др.) |

**Функция `open`**

Эта функция позволяет открывать файл (связывать файловую переменную с файлом) [5, с. 202–203, раздел 12.4.1].

Синтаксис функции `open`

```
(open <file-name> <logical-name> [<mode>])
```

`<file-name>` (имя файла) должно быть строковым или символьным литералом, содержащим относительный или абсолютный путь к файлу. Функция возвращает `TRUE` в случае успешного открытия файла, иначе `FALSE`. При использовании `open` в правилах, она может находиться только в правой части. В табл. 2 приведен список режимов и их значений.

Таблица 2. Список режимов открытия файла

| Режим | Значение (тип доступа к файлу) |
|---|---|
| `"r"` | Только для чтения (значение по умолчанию) |
| `"w"` | Только для записи |
| `"r+"` | Для чтения и записи |
| `"a"` | Только для добавления |
| `"wb"` | Бинарный доступ на запись |

Пример использования функции `open`

```
CLIPS> (open "myfile.clp" writeFile "w")
TRUE
CLIPS>
```

**Функция `close`**

Функция `close` закрывает файл (поток), открытый при помощи `open`. В случае вызова функции без параметров будут закрыты все файлы. Функция возвращает `TRUE` в случае успешного закрытия файла, иначе `FALSE` [5, с. 203–204, раздел 12.4.2].

Синтаксис функции `close`

```
(close [<logical-name>])
```

Пример использования функции `close`

```
CLIPS> (open "myfile.clp" writeFile "w")
TRUE
CLIPS> (open "path/to/file.dp" readFile)
TRUE
CLIPS> (close writeFile)
TRUE
CLIPS> (close writeFile)
FALSE
CLIPS> (close)
TRUE
CLIPS> (close)
FALSE
CLIPS>
```

**Функции `print`, `println`, `printout`**

**`printout`** —- функция, позволяющая выводить информацию на устройство, ассоциированное с логическим именем (например, может производиться вывод информации на экран или запись в файл). Если в качестве логического имени выбрано `nil`, информация никуда не будет выведена. `t` (или `stdout`) в качестве `<logical-name>` отправит вывод на экран (стандартный вывод) [5, с. 204–205, раздел 12.4.3].

Функции **`print`** и **`println`** являются вариантами функции **`printout`**. Обе функции всегда направляют вывод на стандартный вывод (stdout), а функция **`println`** добавляет возврат каретки/перевод строки после печати всех своих аргументов.

Синтаксис функции функций **`print`**, **`println`**, **`printout`**

```
(printout <logical-name> <expression>*)
(print <expression>*)
(println <expression>*)
```

В качестве `<expression>` могут быть использованы, помимо строк, символов, чисел и адресов (а также переменных, содержащих их), специальные символы, приведенные в табл. 3.

Таблица 3. Специальные символьные константы

| Символьная константа | Действие |
|---|---|
| **`crlf`** | перенос каретки на начало новой строки |
| **`tab`** | табуляция горизонтальная |
| **`vtab`** | табуляция вертикальная |

Пример использования функций **`print`**, **`println`**, **`printout`**

```
CLIPS> (printout t "Hello World!" crlf)
Hello World!
CLIPS> (println "Hello World!")
Hello World!
CLIPS> (print "Hello World!" crlf)
Hello World!
CLIPS> (open "data.txt" data "w")
TRUE
CLIPS> (printout data "red green")
CLIPS> (close)
TRUE
CLIPS>
```

**Функция `read`**

Функция, позволяющая считывать информацию по одному полю (признаком, разделяющим поля, является пробел, табуляция или перенос строки; в строковых полях должны быть двойные кавычки) [5, с. 205–206, раздел 12.4.4]. Если при чтении будет обнаружен конец файла, **`read`** вернет символ `EOF`, функция **`get-error`** [5, с. 240, раздел 12.7.14] (если она вызывается) тоже вернет символ `EOF`. Если при чтении будут обнаружены ошибки, то **`read`** вернет символ `FALSE`, а функция **`get-error`** (если она вызвана) вернет код ошибки (отличный от символа `FALSE`).

Синтаксис функции **read**

```
(read [<logical-name>])
```

Пример использования функции **read**

```
CLIPS> (open "data.txt" data "w")
TRUE
CLIPS> (printout data "red green")
CLIPS> (close)
TRUE
CLIPS> (open "data.txt" data)
TRUE
CLIPS> (read data)
red
CLIPS> (get-error)
FALSE
CLIPS> (read data)
green
CLIPS> (read data)
EOF
CLIPS> (get-error)
EOF
CLIPS> (close)
TRUE
CLIPS>
```

**Функция `readline`**

Функция, позволяющая считывать информацию построчно (признаком окончания строки является перенос строки или символ `EOF`) [5, с. 206–207, раздел 12.4.5]. Если при чтении будет обнаружен конец файла, **read** вернет символ `EOF`. Если при чтении будут обнаружены ошибки, то **read** вернет символ `FALSE`.

Синтаксис функции **readline**

```
(readline [<logical-name>])
```

Пример использования функции **readline**

```
CLIPS> (open "data.txt" data "w")
TRUE
CLIPS> (printout data "red green")
CLIPS> (close)
TRUE
CLIPS> (open "data.txt" data)
TRUE
CLIPS> (readline data)
"red green"
CLIPS> (readline data)
EOF
CLIPS> (close)
TRUE
CLIPS>
```

**Функция `format`**

Функция, позволяющая выводить форматированную информацию на устройство, ассоциированное с логическим именем [5, с. 207–210, раздел 12.4.6]. Функция всегда возвращает строку форматированного текста. Если в качестве логического имени выбрано `nil`, форматированный текст не будет записан ни на какое устройство. Функция является аналогом

`printf` языка C и дополняет рассмотренную ранее простую функцию **`printout`**.

Синтаксис функции **`format`**

```
(format <logical-name> <string-expression> <expression>*)
```

Флаги (табл. 4) определяют стиль вывода каждого параметра в `<string-expression>`. Флаги имеют следующий общий формат:

```
%-M.Nx
```

`M.N` — необязательный параметр, определяющий длину целой и дробной частей (соответственно) числа, а знак «-» — тип заполнения.

Таблица 4. Флаги форматирования

| Флаг форматирования | Назначение |
|---|---|
| c | вывод одного символа |
| d | вывод длинного целого числа |
| f | вывод вещественного числа |
| e | вывод вещественного числа в экспоненциальной форме |
| g | вывод в кратчайшей форме, доступной для данного типа |
| o | вывод беззнакового восьмеричного числа |
| x | вывод беззнакового шестнадцатеричного числа |
| s | вывод строки |
| n | переход на новую строку |
| r | перевод коретки на начало строки |
| % | вывод символа «%» |

Пример использования функции **`format`**

```
CLIPS> (format t "Hello World!%n")
Hello World!
"Hello World!
"
CLIPS> (format nil "Integer:     |%d|" 12)
"Integer:     |12|"
CLIPS> (format nil "Integer:     |%4d|" 12)
"Integer:     |  12|"
CLIPS> (format nil "Integer:     |%-04d|" 12)
"Integer:     |12  |"
CLIPS> (format nil "Integer:     |%6.4d|" 12)
"Integer:     |  0012|"
CLIPS> (format nil "Float:       |%f|" 12.01)
"Float:       |12.010000|"
CLIPS> (format nil "Float:       |%7.2f|" 12.01)
"Float:       |  12.01| "
CLIPS> (format nil "Test:        |%e|" 12.01)
"Test:        |1.201000e+01|"
CLIPS> (format nil "Test:        |%7.2e|" 12.01)
"Test:        |1.20e+01|"
CLIPS> (format nil "General:     |%g|" 1234567890)
"General:     |1.23457e+09|"
CLIPS> (format nil "General:     |%6.3g|" 1234567890)
"General:     |1.23e+09|"
```

```
CLIPS> (format nil "Hexadecimal: |%x|" 12)
"Hexadecimal: |c|"
CLIPS> (format nil "Octal:       |%o|" 12)
"Octal:       |14|"
CLIPS> (format nil "Symbols:     |%s| |%s|" value-a1 capacity)
"Symbols:     |value-a1| |capacity|"
CLIPS>
```

**Функция `rename`**

Переименовывает файл. Возвращает `TRUE` в случае удачного переименования и `FALSE` — в противном случае [5, с. 210, раздел 12.4.7].

Синтаксис функции **`remove`**

```
(remove <file-name>)
```

Пример использования функции **`rename`**

```
CLIPS> (rename "data.txt" "new_data.txt")
TRUE
CLIPS>
```

**Функция `remove`**

Удаляет файл. Возвращает `TRUE` в случае удачного удаления и `FALSE` — в противном случае [5, с. 210–211, раздел 12.4.8].

Синтаксис функции **`remove`**

```
(remove <file-name>)
```

Пример использования функции **`remove`**

```
CLIPS> (remove "new_data.txt")
TRUE
CLIPS>
```

## Задачи и порядок выполнения работы

### Общие функции. Операции над списками и строками

Функция, создающая список, — **`create$`** [5, с. 188–189, раздел 12.2.1].

Синтаксис функции **`create$`**

```
(create$ <expression>*)
```

Элементы списка (`<expression>*`) разделяются пробелом и могут быть произвольного типа.

Пример использования функции **create$**

```
CLIPS> (create$ hammer drill saw screw pliers wrench)
(hammer drill saw screw pliers wrench)
CLIPS> (create$ (+ 3 4) (* 2 3) (/ 8 4))
(7 6 2.0)
CLIPS> (create$)
()
CLIPS>
```

Функция, возвращающая элемент списка по номеру, — **nth$** [5, с. 189, раздел 12.2.2].

Синтаксис функции **nth$**

```
(nth$ <integer-expression> <multifield-expression>)
```

`<integer-expression>` (номер элемента) — целое число, большее или равное `1`. Если `<integer-expression>` превышает длину списка, функция возвращает `nil`.

Пример использования функции **nth$**

```
CLIPS> (nth$ 3 (create$ a b c d e f g))
c
CLIPS> (nth$ 12 (create$ a b c d e f g))
nil
CLIPS>
```

Функция, проверяющая вхождение элемента в список, — **member$** [5, с. 189–190, раздел 12.2.3].

Синтаксис функции **member$**

```
(member$ <expression> <multifield-expression>)
```

В случае вхождения элемента (`<expression>`) функция возвратит его номер в списке (`<multifield-expression>`). Если элемент является списком и является подмножеством искомого списка, функция возвратит номера первого и последнего элементов исходного списка, соответствующие заданному списку. Иначе возвратит `FALSE`.

Пример использования функции **member$**

```
CLIPS> (member$ blue (create$ red 3 "text" 8.7 blue))
5
CLIPS> (member$ 4 (create$ red 3 "text" 8.7 blue))
FALSE
CLIPS> (member$ (create$ b c) (create$ a b c d))
(2 3)
CLIPS>
```

Функция (предикат), проверяющая вхождение одного списка в другой, — **subsetp** [5, с. 190, раздел 12.2.4].

Синтаксис функции **subsetp**

```
(subsetp <multifield-expression> <multifield-expression>)
```

В случае вхождения всех элементов списка 1 (первый `<multifield-expression>`) в список 2 (второй `<multifield-expression>`) функция возвратит `TRUE`, иначе — `FALSE`.

Пример использования функции **subsetp**

```
CLIPS> (subsetp
  (create$ hammer saw drill)
  (create$ hammer drill wrench pliers saw)
)
TRUE
CLIPS> (subsetp
  (create$ wrench crowbar)
  (create$ hammer drill wrench pliers saw)
)
FALSE
CLIPS> (subsetp
  (create$)
  (create$ hammer drill wrench pliers saw)
)
TRUE
CLIPS> (subsetp
  (create$ wrench crowbar)
  (create$)
)
FALSE
CLIPS>
```

Функция, удаляющая элементы из списка (заключенные между индексированными начальным и конечным элементами), — **delete$** [5, с. 190–191, раздел 12.2.5]. Данная функция возвращает измененный список. Для того чтобы удалить один элемент из списка, необходимо чтобы начальный и конечный индексы совпадали.

Синтаксис функции **delete$**

```
(delete$
  <multifield-expression>
  <begin-integer-expression>
  <end-integer-expression>
)
```

Пример использования функции **delete$**

```
CLIPS> (delete$ (create$ hammer drill saw pliers wrench) 3 4)
(hammer drill wrench)
CLIPS> (delete$ (create$ computer printer hard-disk) 1 1)
(printer hard-disk)
CLIPS>
```

Функция, создающая список из строки, — explode [5, с. 191, раздел 12.2.6].

Синтаксис функции **explode$**

```
(explode$ <string-expression>)
```

Если `<string-expression>` не содержит ни одного значения, то будет создан список нулевой длины. Числовые значения в строковом выражении будут преобразованы в строку.

Пример использования функции **explode$**

```
CLIPS> (explode$ "hammer drill saw screw")
(hammer drill saw screw)
CLIPS> (explode$ "1 2 abc 3 4 \"abc\" \"def\"")
(1 2 abc 3 4 "abc" "def")
CLIPS> (explode$ "?x ~ )")
(?x ~ ))
CLIPS>
```

Функция **implode$**, обратная функции **explode$**, создаёт строку из списка [5, с. 191–192, раздел 12.2.7].

Синтаксис функции **implode$**

```
(implode$ <multifield-expression>)
```

Пример использования функции **implode$**

```
CLIPS> (implode$ (create$ hammer drill screwdriver))
"hammer drill screwdriver"
CLIPS> (implode$ (create$ 1 "abc" def "ghi" 2))
"1 "abc" def "ghi" 2"
CLIPS> (implode$ (create$ "abc       def         ghi"))
""abc       def         ghi""
CLIPS>
```

Функция, возвращающая подсписок списка (заключенный между индексированными начальным и конечным элементами), — **subseq$** [5, с. 192, раздел 12.2.8].

Синтаксис функции **subseq$**

```
(subseq$
  <multifield-value>
  <begin-integer-expression>
  <end-integer-expression>
)
```

Пример использования функции **subseq$**

```
CLIPS> (subseq$ (create$ hammer drill wrench pliers) 3 4)
(wrench pliers)
CLIPS> (subseq$ (create$ 1 "abc" def "ghi" 2) 1 1)
(1)
CLIPS>
```

Функция, заменяющая подсписок списка (заключенный между индексированными начальным и конечным элементами) на список (или некоторое поле), — **replace$** [5, с. 192–193, раздел 12.2.9].

Синтаксис функции **replace$**

```
(replace$
  <multifield-expression>
  <begin-integer-expression>
  <end-integer-expression>
  <single-or-multi-field-expression>+
)
```

Пример использования функции **replace$**

```
CLIPS> (replace$ (create$ drill wrench pliers) 3 3 machete)
(drill wrench machete)
CLIPS> (replace$ (create$ a b c d) 2 3 x y (create$ q r s))
(a x y q r s d)
CLIPS>
```

Функция, вставляющая поле или список в заданное место списка (определенное индексом) и возвращающая результирующий список, — **insert$** [5, с. 193, раздел 12.2.10].

Синтаксис функции **insert$**

```
(insert$
  <multifield-expression>
  <integer-expression>
  <single-or-multi-field-expression>+
)
```

Пример использования функции **insert$**

```
CLIPS> (insert$ (create$ a b c d) 1 x)
(x a b c d)
CLIPS> (insert$ (create$ a b c d) 4 y z)
(a b c y z d)
CLIPS> (insert$ (create$ a b c d) 5 (create$ q r))
(a b c d q r)
CLIPS>
```

Функция, возвращающая первый элемент (голову) списка, — **first$** [5, с. 194, раздел 12.2.11].

Синтаксис функции **first$**

```
(first$ <multifield-expression>)
```

Пример использования функции **first$**

```
CLIPS> (first$ (create$ a b c))
(a)
CLIPS> (first$ (create$))
()
CLIPS>
```

Функция, возвращающая все элементы списка, кроме первого (хвост), — **rest$** [5, с. 194, раздел 12.2.12].

Синтаксис функции **rest$**

```
(rest$ <multifield-expression>)
```

Пример использования функции **rest$**

```
CLIPS> (rest$ (create$ a b c))
(b c)
CLIPS> (rest$ (create$))
()
CLIPS>
```

Функция, возвращающая длину (количество элементов, полей) списка, — **length$** [5, с. 194, раздел 12.2.13].

Синтаксис функции **length$**

```
(length$ <multifield-expression>)
```

Пример использования функции **length$**

```
CLIPS> (length$ (create$ a b c d e f g))
7
CLIPS>
```

Функция, удаляющая все элементы одного списка (<expression>+) из другого (<multifield-expression>), — **delete-member$** [5, с. 195, раздел 12.2.14]. Возвращает результирующий список.

Синтаксис функции **delete-member$**

```
(delete-member$ <multifield-expression> <expression>+)
```

Пример использования функции **delete-member$**

```
CLIPS> (delete-member$ (create$ a b a c) b a)
(c)
CLIPS> (delete-member$ (create$ a b c c b a) (create$ b a))
(a b c c)
CLIPS>
```

Функция, заменяющая выбранные элементы (<search-expression>+) одного списка (<multifield-expression>) элементами другого (<substitute-expression>), — **replace-member$** [5, с. 195, раздел 12.2.15].

Синтаксис функции **replace-member$**

```
(replace-member$
  <multifield-expression>
  <substitute-expression>
  <search-expression>+
)
```

Пример использования функции **replace-member$**

```
CLIPS> (replace-member$ (create$ a b a b) (create$ a b a) a b)
(a b a a b a a b a a b a)
CLIPS> (replace-member$ (create$ a b a b) (create$ a b a) (create$ a b))
(a b a a b a)
CLIPS>
```

Функция, осуществляющая конкатенацию строк, — **str-cat** [5, с. 196, раздел 12.3.1].

Синтаксис функции **str-cat**

```
(str-cat <expression>*)
```

Пример использования функции **str-cat**

```
CLIPS> (str-cat 2023 "-" 5 "-" 29)
"2023-5-29"
CLIPS>
```

Функция, осуществляющая конкатенацию символов, — **sym-cat** [5, с. 196, раздел 12.3.2].

Синтаксис функции **sym-cat**

```
(sym-cat <expression>*)
```

Пример использования функции **sym-cat**

```
CLIPS> (clear)
CLIPS> (sym-cat super star)
superstar
CLIPS>
```

Функция, возвращающая часть строки (заключенной между начальным и конечными индексами), — **sub-string** [5, с. 196–197, раздел 12.3.3].

Синтаксис функции **sub-string**

```
(sub-string
  <integer-expression>
  <integer-expression>
  <string-expression>
)
```

Пример использования функции **sub-string**

```
CLIPS> (sub-string 3 8 "abcdefghijkl")
"cdefgh"
CLIPS>
```

Функция, возвращающая индекс начала первой строки во второй, — **str-index** [5, с. 197, раздел 12.3.4]. Если вторая строка не содержит первую, функция возвращает `FALSE`.

Синтаксис функции **str-index**

```
(str-index <lexeme-expression> <lexeme-expression>)
```

Пример использования функции **str-index**

```
CLIPS> (str-index "def" "abcdefghi")
4
CLIPS> (str-index "qwerty" "qwertypoiuyt")
1
CLIPS> (str-index "qwerty" "poiuytqwer")
FALSE
CLIPS>
```

Функция, осуществляющая сравнение строк, — **str-compare** [5, с. 199, раздел 12.3.9]. Функция возвращает **1**, **-1** или **0** в случае, если первая строка соответственно больше, меньше или равна второй строке.

Синтаксис функции **str-compare**

```
(str-compare
  <string-or-symbol-expression>
  <string-or-symbol-expression>
)
```

Пример использования функции **str-compare**

```
CLIPS> (str-compare "string" "string")
0
CLIPS> (str-compare "string1" "string2")
-1
CLIPS> (str-compare "string2" "string1")
1
CLIPS>
```

Функция, возвращающая длину строки или символьного литерала, — **str-length** [5, с. 200, раздел 12.3.10].

Синтаксис функции **str-length**

```
(str-length <string-or-symbol-expression>)
```

Пример использования функции **str-length**

```
CLIPS> (str-length "abcd")
4
CLIPS> (str-length xyz)
3
CLIPS>
```

Функция, преобразующая первое поле строки или символьного литерала в соответствующий тип данных, — **string-to-field** [5, с. 201, раздел 12.3.12].

Синтаксис функции **string-to-field**

```
(string-to-field <string-or-symbol-expression>)
```

Пример использования функции **string-to-field**

```
CLIPS> (string-to-field "3.4")
3.4
CLIPS> (string-to-field "a b")
a
CLIPS>
```

Функция, преобразующая строку в последовательность команд и выполняющая эту последовательность, — **eval** [5, с. 197–198, раздел 12.3.5]. Возможности функции **eval** ограничены: так, запрещено использовать в выражениях переменные и объявлять структуры данных (функции, объединения, правила, шаблоны и др.).

Синтаксис функции **eval**

```
(eval <string-or-symbol-expression>)
```

Пример использования функции **eval**

```
CLIPS> (bind ?y 3)
3
CLIPS> (defglobal ?*x* = 4)
CLIPS> (eval "(+ 3 4)")
7
CLIPS> (eval "(+ ?*x* ?y)")
7
CLIPS> (eval "?*x*")
4
CLIPS> (eval "?y")
3
CLIPS> (eval "3")
3
CLIPS>
```

Аналог **eval**, позволяющий использовать переменные и создавать структуры, — функция **build** [5, с. 198, раздел 12.3.6].

Синтаксис функции **build**

```
(build <string-or-symbol-expression>)
```

Пример использования функции **build**

```
CLIPS> (clear)
CLIPS> (build "(defrule hello ⇒ (println \"Hello\"))")
TRUE
CLIPS> (rules)
hello
For a total of 1 defrule.
CLIPS> (run)
Hello
CLIPS>
```

Функции, позволяющие преобразовывать регистр строки (в верхний и нижний соответственно), — upcase [5, с. 198–199, раздел 12.3.7] и lowcase [5, с. 199, раздел 12.3.8].

Синтаксис функции **upcase**

```
(upcase <string-or-symbol-expression>)
```

Синтаксис функции **lowcase**

```
(lowcase <string-or-symbol-expression>)
```

Пример использования функции **upcase**

```
CLIPS> (upcase "This is a test of upcase")
"THIS IS A TEST OF UPCASE"
CLIPS> (upcase A_Word_Test_for_Upcase)
A_WORD_TEST_FOR_UPCASE
CLIPS>
```

Пример использования функции **lowcase**

```
CLIPS> (lowcase "This is a test of lowcase")
"this is a test of lowcase"
CLIPS> (lowcase A_Word_Test_for_Lowcase)
a_word_test_for_lowcase
CLIPS>
```

### Математические функции

Сумма — **+** (аргументы — числа) [5, с. 217, раздел 12.5.1].

Синтаксис функции **+**

```
(+ <numeric-expression> <numeric-expression>+)
```

Пример использования функции **+**

```
CLIPS> (+ 2 3 4)
9
CLIPS> (+ 2 3.0 5)
10.0
CLIPS> (+ 3.1 4.7)
7.8
CLIPS>
```

Разность — **-** [5, с. 217, раздел 12.5.2].

Синтаксис функции **-**

```
(- <numeric-expression> <numeric-expression>+)
```

Пример использования функции `-`

```
CLIPS> (- 12 3 4)
5
CLIPS> (- 12 3.0 5)
4.0
CLIPS> (- 4.7 3.1)
1.6
CLIPS>
```

Умножение — `*` [5, с. 218, раздел 12.5.3].

Синтаксис функции функции `*`

```
(* <numeric-expression> <numeric-expression>+)
```

Пример использования функции `*`

```
CLIPS> (* 2 3 4)
24
CLIPS> (* 2 3.0 5)
30.0
CLIPS> (* 3.1 4.7)
14.57
CLIPS>
```

Деление — `/` [5, с. 218, раздел 12.5.4].

Синтаксис функции функции `/`

```
(/ <numeric-expression> <numeric-expression>+)
```

Пример использования функции `/`

```
CLIPS> (/ 4 2)
2.0
CLIPS> (/ 4.0 2.0)
2.0
CLIPS> (/ 24 3 4)
2.0
CLIPS>
```

Целочисленное деление — `div` [5, с. 218–219, раздел 12.5.5].

Синтаксис функции `div`

```
(div <numeric-expression> <numeric-expression>+)
```

Пример использования функции `div`

```
CLIPS> (div 4 2)
2
CLIPS> (div 5 2)
2
CLIPS> (div 33 2 3 5)
1
CLIPS>
```

Максимум — `max` [5, с. 219, раздел 12.5.6].

Синтаксис функции `max`

```
(max <numeric-expression>+)
```

Пример использования функции `max`

```
CLIPS> (max 3.0 4 2.0)
4
CLIPS>
```

Минимум — `min` [5, с. 219–220, раздел 12.5.7]

Синтаксис функции `min`

```
(min <numeric-expression>+)
```

Пример использования функции `min`

```
CLIPS> (min 4 0.1 -2.3)
-2.3
CLIPS>
```

Абсолютное значение, т.е. модуль — `abs` [5, с. 220, раздел 12.5.8].

Синтаксис функции `abs`

```
(abs <numeric-expression>)
```

Пример использования функции `abs`

```
CLIPS> (abs 4.0)
4.0
CLIPS> (abs -2)
2
CLIPS>
```

Перевод числа в вещественный формат — `float` [5, с. 220, раздел 12.5.9].

Синтаксис функции `float`

```
(float <numeric-expression>)
```

Пример использования функции `float`

```
CLIPS> (float 4.0)
4.0
CLIPS> (float -2)
-2.0
CLIPS>
```

Перевод числа в целочисленный формат — `integer` [5, с. 220–221, раздел 12.5.10].

Синтаксис функции `integer`

```
(integer <numeric-expression>)
```

Пример использования функции `integer`

```
CLIPS> (integer 4.0)
4
CLIPS> (integer -2)
-2
CLIPS>
```

Тригонометрические функции (исчисление в радианах) приведены в табл. 5 [5, с. 221, раздел 12.5.11].

Конвертация различных единиц измерения угла между собой:

- градусы в градианы — **deg-grad** [5, с. 222, раздел 12.5.12];
- градусы в радианы — **deg-rad** [5, с. 222–223, раздел 12.5.13];
- градианы в градусы — **grad-deg** [5, с. 223, раздел 12.5.14];
- радианы в градусы — **rad-deg** [5, с. 223, раздел 12.5.15].

Синтаксис функции **deg-grad**

```
(deg-grad <numeric-expression>)
```

Пример использования функции **deg-grad**

```
CLIPS> (deg-grad 90)
100.0
CLIPS>
```

Синтаксис функции **deg-rad**

```
(deg-rad <numeric-expression>)
```

Пример использования функции **deg-rad**

```
CLIPS> (deg-rad 180)
3.141592653589793
CLIPS>
```

Синтаксис функции **grad-deg**

```
(grad-deg <numeric-expression>)
```

Пример использования функции **grad-deg**

```
CLIPS> (grad-deg 100)
90.0
CLIPS>
```

Синтаксис функции **rad-deg**

```
(rad-deg <numeric-expression>)
```

Пример использования функции **rad-deg**

```
CLIPS> (rad-deg 3.141592653589793)
180.0
CLIPS>
```

Число π (пи) — **pi** [5, с. 223–224, раздел 12.5.16].

Синтаксис функции **pi**

```
(pi)
```

Пример использования функции **pi**

```
CLIPS> (pi)
3.141592653589793
CLIPS>
```

Таблица 5. Тригонометрические функции CLIPS

| Функция | Возвращает | Функция | Возвращает |
|---|---|---|---|
| **acos** | арккосинус | **acosh** | гиперболический арккосинус |
| **acot** | арккотангенс | **acoth** | гиперболический арккотангенс |
| **acsc** | арккосеканс | **acsch** | гиперболический арккосеканс |
| **asec** | арксеканс | **asech** | гиперболический арксеканс |
| **asin** | арксинус | **asinh** | гиперболический арксинус |
| **atan** | арктангенс | **atanh** | гиперболический арктангенс |
| **cos** | косинус | **cosh** | гиперболический косинус |
| **cot** | котангенс | **coth** | гиперболический котангенс |
| **csc** | косеканс | **csch** | гиперболический косеканс |
| **sec** | секанс | **sech** | гиперболический секанс |
| **sin** | синус | **sinh** | гиперболический синус |
| **tan** | тангенс | **tanh** | гиперболический тангенс |

Квадратный корень из числа — **sqrt** [5, с. 224, раздел 12.5.17].

Синтаксис функции **sqrt**

```
(sqrt <numeric-expression>)
```

Пример использования функции **sqrt**

```
CLIPS> (sqrt 9)
3.0
CLIPS>
```

Возведение числа (первый аргумент) в степень (второй аргумент) — ****** [5, с. 224, раздел 12.5.18].

Синтаксис функции функции ******

```
(** <numeric-expression> <numeric-expression>)
```

Пример использования функции ******

```
CLIPS> (** 3 2)
9.0
CLIPS>
```

Возведение экспоненты (эйлерова константа) в степень — **exp** [5, с. 224–225, раздел 12.5.19].

Синтаксис функции **exp**

```
(exp <numeric-expression>)
```

Пример использования функции **exp**

```
CLIPS> (exp 1)
2.718281828459045
CLIPS>
```

Натуральный логарифм — **log** [5, с. 225, раздел 12.5.20].

Синтаксис функции **log**

```
(log <numeric-expression>)
```

Пример использования функции **log**

```
CLIPS> (log 2.718281828459045)
1.0
CLIPS>
```

Десятичный логарифм — clipsflog10 [5, с. 225, раздел 12.5.21].

Синтаксис функции **log10**

```
(log10 <numeric-expression>)
```

Пример использования функции **log10**

```
CLIPS> (log10 100)
2.0
CLIPS>
```

Округление до целого — **round** [5, с. 226, раздел 12.5.22].

Синтаксис функции **round**

```
(round <numeric-expression>)
```

Пример использования функции **round**

```
CLIPS> (round 3.6)
4
CLIPS>
```

Остаток от деления одного числа (первый аргумент) на другое (второй аргумент) — **mod** [5, с. 226, раздел 12.5.23].

Синтаксис функции **mod**

```
(mod <numeric-expression> <numeric-expression>)
```

Пример использования функции **mod**

```
CLIPS> (mod 5 2)
1
CLIPS> (mod 3.7 1.2)
0.1
CLIPS>
```

### Дополнительные функции

Функция, создающая новую или модифицирующая существующую переменную, — `bind` [5, с. 226–228, раздел 12.6.1].

Синтаксис функции `bind`

```
(bind <variable> <expression>*)
```

Пример использования функции `bind`

```
CLIPS> (defglobal ?*x* = 3.4)
CLIPS> ?*x*
3.4
CLIPS> (bind ?*x* (+ 8 9))
17
CLIPS> ?*x*
17
CLIPS> (bind ?*x* (create$ a b c d))
(a b c d)
CLIPS> ?*x*
(a b c d)
CLIPS> (bind ?*x* d e f)
(d e f)
CLIPS> ?*x*
(d e f)
CLIPS> (bind ?*x*)
3.4
CLIPS> ?*x*
3.4
CLIPS> (bind ?x 32)
32
CLIPS> ?x
32
CLIPS> (reset)
CLIPS> ?x
[EVALUATN1] Variable x is unbound
FALSE
CLIPS>
```

Условная конструкция — (`if then else`) [5, с. 228–229, раздел 12.6.2]. Все действия в данной конструкции выполняются последовательно и конструкция возвращает результат последнего действия.

Синтаксис функции `if`

```
(if
  <expression>
then
  <action>*
[else
  <action>*]
)
```

Пример использования функции `if`

```
CLIPS> (clear)
CLIPS> (deftemplate person
  (slot name)
  (slot age)
)
CLIPS> (defrule print-age
  (person (name ?name) (age ?age))
  =>
  (if (= ?age 1)
```

```
  then
    (println ?name " is 1 year old")
  else
    (println ?name " is " ?age " years old")
  )
)
CLIPS> (assert (person (name "Sam Jones") (age 1)))
<Fact-1>
CLIPS> (assert (person (name "Jill Smith") (age 13)))
<Fact-2>
CLIPS> (run)
Jill Smith is 13 years old
Sam Jones is 1 year old
CLIPS>
```

Конструкция для организации циклов с предусловием — **while** [5, с. 229–230, раздел 12.6.3].

Синтаксис функции **while**

```
(while <expression> [do]
  <action>*
)
```

Пример использования функции **while**

```
CLIPS> (deffunction countdown (?count)
  (while (> ?count 0)
    (println ?count)
    (bind ?count (- ?count 1))
  )
  (return (void))
)
CLIPS> (countdown 3)
3
2
1
CLIPS>
```

Функция, прерывающая выполнение циклов — **break** [5, с. 232, раздел 12.6.8].

Синтаксис функции **break**

```
(break)
```

Пример использования функции **break**

```
CLIPS> (deffunction iterate (?num)
  (bind ?i 0)
  (while TRUE do
    (if (≥ ?i ?num)
    then (break)
    )
    (print ?i " ")
    (bind ?i (+ ?i 1))
  )
  (println)
)
CLIPS> (iterate 1)
0
CLIPS> (iterate 10)
0 1 2 3 4 5 6 7 8 9
CLIPS>
```

Конструкция для организации циклов со счетчиком — **loop-for-count** [5, с. 230, раздел

12.6.4].

Синтаксис функции `loop-for-count`

```
(loop-for-count <range-spec> [do] <action>*)

<range-spec> ::= <end-index> |
                 (<loop-variable> <start-index> <end-index>) |
                 (<loop-variable> <end-index>)
<start-index> ::= <integer-expression>
<end-index> ::= <integer-expression>
```

Пример использования функции `loop-for-count`

```
CLIPS> (loop-for-count 2 (println "Hello World!"))
Hello World!
Hello World!
FALSE
CLIPS> (loop-for-count (?cnt1 2 4) do
  (loop-for-count (?cnt2 1 3) do
    (println ?cnt1 " " ?cnt2)
  )
)
2 1
2 2
2 3
3 1
3 2
3 3
4 1
4 2
4 3
FALSE
CLIPS>
```

Конструкция множественного выбора — `witch` [5, с. 233, раздел 12.6.9].

Синтаксис функции `switch`

```
(switch <test-expression>
  <case-statement>*
  [<default-statement>]
)

<case-statement> ::= (case <comparison-expression> then <action>*)
<default-statement> ::= (default <action>*)
```

Пример использования функции `switch`

```
CLIPS> (deffunction complement (?color)
  (switch ?color
    (case red then cyan)
    (case cyan then red)
    (case green then magenta)
    (case magenta then green)
    (case blue then yellow)
    (case yellow then blue)
    (default FALSE)
  )
)
CLIPS> (complement green)
magenta
CLIPS> (complement black)
FALSE
CLIPS>
```

Функция `return` позволяет прервать выполнение правила (правой части), функции или

метода и вернуть при этом (если необходимо) какое-либо значение [5, с. 231–232, раздел 12.6.7].

Синтаксис функции `return`

```
(return [<expression>])
```

Пример использования функции `return`

```
CLIPS> (deffunction sign (?num)
  (if (> ?num 0) then (return 1))
  (if (< ?num 0) then (return -1))
  0
)
CLIPS> (sign 5)
1
CLIPS> (sign -10)
-1
CLIPS> (sign 0)
0
CLIPS>
```

Функция **progn**, вычисляющая последовательность операндов (в списке) и возвращающая результат последнего [5, с. 231, раздел 12.6.5].

Синтаксис функции `progn`

```
(progn <expression>*)
```

Пример использования функции `progn`

```
CLIPS> (progn (setgen 5) (gensym))
gen5
CLIPS>
```

Функция **gensym**, генерирующая уникальный идентификатор (при каждом вызове возвращается новый идентификатор) [5, с. 234–235, раздел 12.7.1].

Синтаксис функции `gensym`

```
(gensym)
```

Пример использования функции `gensym`

```
CLIPS> (deftemplate order
  (slot id (default-dynamic (gensym)))
  (slot item)
  (slot quantity)
)
CLIPS> (assert (order (item C3) (quantity 3)))
<Fact-1>
CLIPS> (assert (order (item B1) (quantity 1)))
<Fact-2>
CLIPS> (assert (order (item C3) (quantity 3)))
<Fact-3>
CLIPS> (facts)
f-1     (order (id gen1) (item C3) (quantity 3))
f-2     (order (id gen2) (item B1) (quantity 1))
f-3     (order (id gen3) (item C3) (quantity 3))
For a total of 3 facts.
CLIPS>
```

Функция **setgen**, присваивающая номер, начиная с которого будут генерироваться уникальные идентификаторы посредством **gensym** [5, с. 235–236, раздел 12.7.3].

Синтаксис функции **setgen**

```
(setgen <integer-expression>)
```

Пример использования функции **setgen**

```
CLIPS> (setgen 32)
32
CLIPS> (gensym)
gen32
CLIPS>
```

Функция **random**, возвращающая число из заданного интервала (интервал указывается опционально, значения могут быть только целочисленными) [5, с. 236, раздел 12.7.4].

Синтаксис функции **random**

```
(random [<start-integer-expression> <end-integer-expression>])
```

Пример использования функции **random**

```
CLIPS> (clear)
CLIPS> (defrule roll-the-dice
  ?f <- (roll-the-dice)
  =>
  (retract ?f)
  (bind ?roll1 (random 1 6))
  (bind ?roll2 (random 1 6))
  (println "Your roll is: " ?roll1 " " ?roll2)
)
CLIPS> (assert (roll-the-dice))
<Fact-1>
CLIPS> (run)
Your roll is: 5 6
CLIPS>
```

Функция **seed**, инициализирующая генератор случайных чисел [5, с. 236–237, раздел 12.7.5].

Синтаксис функции **seed**

```
(seed <integer-expression>)
```

Параметр `<integer-expression>` должен быть целого типа. Он определяет значение, с которого будут генерироваться случайные числа.

Пример использования функции **seed**

```
CLIPS> (seed 2357)
CLIPS> (random 1 10)
10
CLIPS> (random 1 10)
4
CLIPS> (random 1 10)
2
CLIPS> (seed 2357)
CLIPS> (random 1 10)
10
CLIPS> (random 1 10)
4
CLIPS> (random 1 10)
2
CLIPS>
```

Функция **time**, возвращающая вещественное представление системного времени [5, с. 237, раздел 12.7.6].

Синтаксис функции **time**

```
(time)
```

Пример использования функции **time**

```
CLIPS> (time)
3124213200.25809
CLIPS>
```

Функция **timer**, возвращающая количество секунд, затраченное интерпретатором на обработку введенного выражения [5, с. 239, раздел 12.7.10].

Синтаксис функции **timer**

```
(timer <expression>*)
```

Пример использования функции **timer**

```
CLIPS> (timer (loop-for-count 10000 (+ 3 4)))
0.00206589698791504
CLIPS>
```

Функция **sort**, возвращающая отсортированный по ключу список [5, с. 238, раздел 12.7.8].

Синтаксис функции **sort**

```
(sort <comparison-function-name> <expression>*)
```

Пример использования функции **sort**

```
CLIPS> (sort > 4 3 5 7 2 7)
(2 3 4 5 7 7)
CLIPS> (deffunction string> (?a ?b)
  (> (str-compare ?a ?b) 0)
)
CLIPS> (sort string> ax aa bk mn ft m)
(aa ax bk ft m mn)
CLIPS>
```

### Задание на лабораторную работу

Решить задачу, указанную в варианте, использую функциональный стиль программирования в среде CLIPS. Разработать алгоритм для решения поставленной задачи в соответствии с вариантом. Реализовать разработанный алгоритм в среде CLIPS, реализовать ввод данных из файла и вывод результата в файл. Протестировать работу алгоритма на всех возможных вариантах наборов входных данных.

### Требования к реализации

Вариант задания назначается преподавателем.

Все функции, входные и выходные данные алгоритма должны быть сохранены в файлы.

### Варианты заданий

1. Написать функцию, получающую в качестве параметра имя файла, в котором в трёхмерных координатах заданы прямая (двумя точками) и некоторый замкнутый пространственный контур (множеством точек, соединённых отрезками, перечисленных в определённом порядке обхода). Определить положение прямой относительно контура (проходит внутри контура, пересекает контур или лежит вне контура). Результат вывести на экран.
2. Написать функцию, получающую в качестве параметра имя файла, содержащего координаты выпуклого $n$-угольника в порядке обхода $(n > 3)$. Разбить его на треугольники диагоналями, так чтобы сумма длин диагоналей была минимальной. Координаты получившихся треугольников записать в файл.
3. Написать функцию, получающую в качестве параметра имя файла, содержащего число $n$, координаты вершин выпуклого $n$-угольника в порядке обхода (в двумерных декартовых координатах), число $m$, координаты выпуклого $m$-угольника. Написать функцию, находящую $k$-угольник, являющийся пересечением $n$- и $m$-угольников.
4. Написать функцию, получающую в качестве параметра имя файла. В котором содержатся размерность и элементы матрицы в виде списка. Написать функцию, вычисляющую обратную матрицу. Результат записать в файл.

   Пример входного файла:

   ```
   3 3
   2 11 4 5 7 3 2 1 1
   ```

5. Написать функцию, получающую в качестве параметра имя файла, содержащего два числа – $n$ и $m$. Найти на промежутке от $n$ до $m$ числа с наибольшим количеством простых делителей. Выходных данные – число простых делителей и для каждого числа само число и список его простых делителей. Результат записать в файл.

   Пример входного файла:

   ```
   2 13
   ```

Результат:

```
2
6 2 3
10 2 5
12 2 3
```

6. Написать функцию, получающую в качестве параметра имя файла, содержащего число в десятичной системе счисления. Перевести число в двоичную и восьмеричную системы. Результат записать в файл.
7. Написать функцию, получающую в качестве параметра имя файла, содержащего карту кораблей и размерность карты. Каждый корабль представляет собой вертикальный или горизонтальный набор подряд идущих закрашенных клеток (1), разные корабли не соприкасаются по сторонам или углам и не накладываются друг на друга. Корабли могут быть более, чем из четырех клеток. Необходимо найти корабль наибольшей площади. Результат записать в файл.

   Пример входного файла:

```
12 12
0 0 0 0 0 0 0 0 0 0 0 1
0 1 1 1 1 1 0 0 0 0 0 1
0 0 0 0 0 0 0 1 0 0 0 1
0 1 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0
0 1 0 1 1 1 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0
0 1 1 0 0 0 0 0 0 0 0 0
```

   Результат:

```
6
```

8. Написать функцию, получающую в качестве параметра имя файла, в котором содержатся: количество дорог, информация о каждой дороге в формате первый город, второй город, стоимость проезда (все дороги двусторонние). Так же в файле указаны города начала и конца пути. Требуется проложить наиболее дешёвую дорогу. Результат записать в файл.
9. Написать функцию, получающую в качестве параметра имя файла, в первой строке которого указана размерность первой матрицы ($m$ $n$), во второй - размерность второй матрицы ($n$ $p$). В третьей строке - матрица $A$ в виде списка элементов, в четвертой – матрица $B$. Необходимо вычислить произведение матриц. Результат вычислений записать в файл.

   Пример входного файла:

```
3 3
3 2
3 2 1 4 2 1 2 2 1
4 2 5 1 2 3
```

10. Написать функцию, получающую в качестве параметра имя файла, содержащего число $n$

и ещё $n^2$ чисел. Необходимо расположить их в матрице $n \cdot n$, так что бы определитель был минимален. Результат записать в файл.

Пример входного файла:

```
2
0 1 2 3
```

Результат:

```
0 3
2 1
```

11. Написать функцию, получающую в качестве параметра имя файла, в котором содержится множество координат точек (в двумерных декартовых координатах) и число $n$. Написать функцию, находящую $n$-угольник с наибольшей площадью с вершинами в данных точках.
12. Написать функцию, получающую в качестве параметра имя файла, содержащего элементы двух матриц размерности $3 \cdot 3$ в виде списков. Написать функцию, вычисляющую произведение матриц. Результат записать в файл.

    Пример входного файла:

```
2 11 4 5 7 3 2 1 1
4 5 2 7 1 1 2 1 3
```

13. Написать функцию, получающую в качестве параметра имя файла, в котором содержится левая часть уравнения, заданного относительно переменной $X$. В уравнении могут быть использованы скобки и 4 арифметических операции над переменной. Необходимо написать функцию, считывающую уравнение из файла и решающую его относительно $X$. Результат решения следует записать в файл. Результатами могут быть: указание на отсутствие решения (NO SOLUTIONS); указание на любое значение переменной (ANY $X$); указание на бесконечное множество решений, за исключением некоторого или некоторых (ANY X EXCEPT $y_1$, $y_2$, ..., $y_n$); найденный корень уравнения (ROOT IS $y$). Все решения должны быть представлены в целочисленной форме, либо в виде несократимых дробей.

    Примеры входного файла $\left(\text{для уравнения } 2 + \dfrac{0}{x-1} - 2 = 0 \text{ и } 8x - 20 = 0\right)$:

```
2+0/(X-1)-2
8*X-20
```

    Примеры выходного файла:

```
ANY X EXCEPT 1
ROOT IS 5/2
```

14. Написать функцию, получающую в качестве параметра имя файла, содержащего коэффициенты системы линейных уравнений, заданные в виде прямоугольной матрицы. С помощью допустимых преобразований привести систему к треугольному виду. Результат записать в файл.

15. Написать функцию, получающую в качестве параметра имя файла, содержащего множество координат точек (в двумерных декартовых координатах) и число $n$. Найти $n$-звенную незамкнутую ломанную наибольшей длины с узлами в данных точках. Результат записать в файл.
16. Написать функцию, получающую в качестве параметра имя файла, содержащего два числа – $n$ и $m$. Найти все пары дружественных чисел в диапазоне от $n$ до $m$.

    Два натуральных числа называются дружественными, если каждое из них равно сумме всех делителей другого, кроме самого этого числа. Результат записать в файл.
17. Написать функцию, получающую в качестве параметра имя файла, содержащего лабиринт и его размерность. Подсчитать количество изолированных областей в лабиринте. Передвижение возможно только по вертикали и горизонтали на незанятые стенами участки. Результат записать в файл.

    Пример входного файла:

```
15
4
###############
#  # #       ##
#  #   ## ### #
###############
```

    Результат:

```
2
```

18. Написать функцию, получающую в качестве параметра имя файла, содержащего карту кораблей и размерность карты. Каждый корабль представляет собой вертикальный или горизонтальный набор подряд идущих закрашенных клеток (1), разные корабли не соприкасаются по сторонам или углам и не накладываются друг на друга. Корабли могут быть более, чем из четырех клеток. Необходимо найти число кораблей. Результат записать в файл.

    Пример входного файла:

```
12 12
0 0 0 0 0 0 0 0 0 0 0 1
0 1 1 1 1 1 0 0 0 0 0 1
0 0 0 0 0 0 0 1 0 0 0 1
0 1 0 0 0 0 0 0 0 0 0 0
53
0 1 0 0 0 0 0 0 0 0 0 0
0 1 0 1 1 1 1 1 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0
0 1 1 0 0 0 0 0 0 0 0 0
```

    Результат:

```
7
```

### Контрольные вопросы и задания

1. Укажите синтаксис для объявления функции в CLIPS.
2. Приведите пример удаления функции.
3. Чем отличаются команды (`deffunction`), (`defgeneric`) и (`defmethod`)?
4. Если несколько перегруженных методов удовлетворяет набору входных параметров, то которые из них будут выполнены?
5. Какие функции используются для ввода и вывода информации? Укажите их синтаксис.
6. С помощью какой функции можно создать список? Приведите её синтаксис.
7. Приведите пример функции для работы со списками.
8. Приведите пример функции для работы со строками.
9. Что выполняют функции `explode` и `implode`?
10. Для чего используются функции `eval` и `build`? Приведите пример их работы.
11. Приведите пример работы 5–7 математических функций.
12. Приведите синтаксис присвоения переменной значения.
13. Сформулируйте правила создания условных конструкций.
14. Приведите синтаксис создания циклических конструкций.
15. Приведите пример генерирования случайного числа?

### Форма отчёта по лабораторной работе

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

Номер варианта студенту выдаётся преподавателем.

Отчёт на защиту предоставляется в печатном виде.

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

**Основная литература**


1. *Исаев С. В.* Интеллектуальные системы : учебное пособие / С. В. Исаев ; О. С. Исаева. — Красноярск : Сибирский федеральный университет, 2017. — 120 с. — Текст: электронный. — URL: https://www.iprbookshop.ru/84365.html.
2. *Пальмов С. В.* Интеллектуальные системы и технологии : учебное пособие / С. В. Пальмов. — Самара : Поволжский государственный университет телекоммуникаций и информатики, 2017. — 195 с. — Текст: электронный. — URL: https://www.iprbookshop.ru/75375.html.
3. *Пятаева А. В.* Интеллектуальные системы и технологии : учебное пособие / А. В. Пятаева ; К. В. Раевич. — Красноярск : Сибирский федеральный университет, 2018. — 144 с. — Текст: электронный. — URL: https://www.iprbookshop.ru/84358.

**Электронные ресурсы:**

5. CLIPS Reference Manual. Volume I. Basic Programming Guide. Version 6.40. — 428 с. — URL: https://www.clipsrules.net/documentation/v640/bpg640.pdf.
4. *Трофимов В. Б.* Экспертные системы в АСУ ТП : учебник / В. Б. Трофимов ; И. О. Темкин. — М., Вологда : Инфра-Инженерия, 2020. — 284 с. — Текст: электронный. — URL: https://www.iprbookshop.ru/98489.html.