ВСР № 2.2. Справочник по математическим объектам и их представлению в Scilab

Переменная

Переменная в Scilab (англ. *Variable*) — это именованный массив всего с одним полем, которое хранит данные некоторого типа.

Среди типов данных можно выделить:

- Числа
  - Целые числа
  - Вещественные числа
  - Комплексные числа
- Строки
- Логические переменные

Создать переменную в среде не составляет труда. Для этого, как вы уже знаете, достаточно ввести ее имя и присвоить ей какое-либо начальное значение. Для переменной будет автоматически выделено место в памяти, а ее область видимости по умолчанию становится локальной.

Для того, чтобы посмотреть текущее значение переменной, достаточно просто обратиться к ней по имени, либо воспользоваться редактором переменных. В дальнейшем вы убедитесь, что редактор крайне удобен для матриц.

Запомните следующие правила, которым должны удовлетворять имена переменных и вообще любых объектов среды:

- Имя переменной может состоять из букв латинского алфавита (верхнего и нижнего регистра) и цифр;
- Имя переменной не может начинаться с цифры, но может начинаться с символов '%', '_', '#', '!', '$', '?';
- Регистр в имени играет роль, то есть переменные с именами var,VAR ,Var и т. п. разные;
- Запрещено совпадение имени переменной с зарезервированными словами, такими как имена объявленных функций, констант и др.;

Упражнение

Для начала попробуем создать несколько переменных

```
--> n1=25, n2=65.33, n3=-36.4e-6, n4=9+%i*4
```

В результате мы получили 4 переменных, 3 из которых хранят вещественные числа и одна — комплексное число. Обратите внимание, как вводится мнимая единица. В среде Scilab мнимая единица является предопределенной константой с именем ' i '. Знак процента указывает на то, что вы обращаетесь к *константе*.

Несмотря на то, что в переменной n1 нет дробной части, с точки зрения хранения этого числа в памяти оно вещественно и имеет тип *double*. Убедимся, что переменные хранят числа, для чего вызовем внутреннюю функцию type(). Чтобы не вызывать функцию четыре раза, мы объединим наши переменные в вектор. Для объединения объектов в вектор необходимо перечислить их через запятую внутри квадратных скобок

```
--> type([n1,n2,n3,n4])
ans =
    1.
```

В ответ функция возвращает число 1. Данным числом функция закодировала информацию о том, что в качестве аргумента была передана матрица, содержащая вещественные или комплексные числа удвоенной точности.

Создадим текстовую переменную следующей командой

```
--> n5='Scilab'
```

Вызовем функцию type() для переменной n5

```
--> type(n5)
ans =
    10.
```

Снова отметим, что число 10 кодирует информацию о том, что в массиве n5 хранятся текстовые строки. Не стоит заучивать наизусть что означает то или иное число, возвращаемое функцией type(), так как вы всегда можете воспользоваться справочной информацией.

Введите команду, которая откроет окно справки и перейдет к разделу, описывающему функцию type()

```
--> help type
```

Запомните команду help и всегда применяйте её в подобных случаях.

Логические переменные могут хранить в себе одну из двух предопределенных констант: %T (от англ. *True — Истина*) и %F (от англ. *False — Ложь*). Для логических типов данных применяются особые (логические) операции, тем не менее, этот тип данных совместим с числовым типом данных: любое ненулевое значение является эквивалентом %T, а соответственно нулевое значение — %F.

Убедимся в последнем на следующем поучительном примере.

```
--> n6=%T; n7=%F;
--> n4 & n6, n2 | n7
ans  =
 T
ans  =
 T
```

Обратите внимание, что логическая операция успешно прошла даже с комплексной переменной. Запомните это свойство, так как им пользуются при написании сценариев, чтобы лишний раз не создавать логические переменные.

Наконец, обратим ваше внимание на то, что интерпретатор динамически реагирует на присваивание. Это означает, что любая объявленная переменная может хранить в любой момент времени любой тип данных.

Запишите следующий сценарий

```
--> type(n4), n4='text'; type(n4), n4=[n1,n2,n3]; type(n4)
ans  =
   1.
ans  =
   10.
ans  =
   1.
```

Переменной n4, хранящей комплексное число, мы последовательно присвоили сначала текстовую строку, а затем вектор из числовых значений. Обратите внимание, что такие действия не приводят к ошибке интерпретатора. Если заглянуть вглубь этого процесса переприсваивания, то можно увидеть, что это многошаговый процесс: сначала необходимо освободить память от предыдущих данных, затем выделить столько памяти, сколько нужно для

хранения другого типа данных и, наконец, заполнить память данными. Разумеется, что все эти действия умело скрыты от глаз пользователя.

Также вы легко можете видеть, как стирается граница между переменными и массивами в рамках одного имени, так как в любом случае среда работает с памятью. В некоторых ситуациях данное свойство может быть губительно для данных, так как вы спокойно можете сослаться на имя, которое например ссылается на имеющийся в памяти массив и присвоить ему переменную (по сути тот же массив, но с одним полем и другими данными). При этом вернуть назад данные вы не сможете, если не приняли страховочные меры, например, защита от изменений, которую мы рассмотрим ниже.

Вы уже умеете создавать переменные, однако этого не достаточно для работы. Рано или поздно вы наталкнетесь на вполне естественные вопросы: «Сколько переменных сейчас в памяти?», «Какие имена свободны, а какие заняты?», «Как очистить память от ненужных мне объектов?» и другие.

Вообще говоря, для работы с любыми объектами в среде существуют предопределенные функции, которые всегда загружаются вместе со средой. Мы рассмотрим все вопросы в этом разделе, и далее будет полагаться, что данная здесь информация переносится на все последующие объекты среды.

Имя функции	Ключевые слова для параметров	Аргументы	Назначение
whos	**Необязательные**: **-type** typ **-name** nam	typ — текстовая строка, кодирующая тип данных nam — имена искомых переменных, либо их начальные фрагменты	Выводит список переменных в длинной форме (с указанием типа и размера), если вызывается без аргументов. Выводит список переменных, хранящих определенный тип данных (если указан параметр -type) или/и обладающих определенным именем или фрагментом этого имени (если указан параметр -name).
who()	аргументы перечисляются в скобках	**Необязательные**: 'local' или 'get' — список локальных переменных и констант в коротком виде 'global' — список глобальных переменных в коротком виде 'sorted' — возвращает сортированный список объявленных переменных (локальных и глобальных) с текущими значениями параметров памяти.	Без аргументов возвращает несортированный список объявленных переменных (локальных и глобальных) с текущими значениями параметров памяти.
who_user	нет	нет	Работает почти как who('local'), но выводит только переменные, созданные пользователем во время текущей сессии.
typeof()	нет	**Обязательный**: object — имя объекта среды	Возвращает в виде строки тип объекта, чье имя передается в качестве аргумента. В отличие от функции type(), более гибко работает с типами данных. О возвращаемых значениях смотрите в справке по данной функции.
clear	нет	**Необязательные**: Аргументами являются имена объектов, перечисляемых через пробел.	Удаляет из памяти объекты и освобождает имена. Если вызывается без аргуметов, то удаляются все незащищенные объекты текущей сессии.

predef()	нет	'a' или 'all' — защищает все переменные из списка who('get') 'c' или 'clear' — снимает защиту со всех переменных списка who('get'), кроме последних 7 (по умолчанию), некоторых предопределенных системных переменных и констант n (n>7) — устанавливает n последних переменных в качестве защищенных 'names' — возвращает список защищенных переменных	Функция предназначена для защиты переменных. Вызов без аргументов возвращает количество защищенных переменных. Остальные вариации перечислены в колонке с аргументами.
exists()	нет	**Обязательный**: name — имя переменной (передается строкой, то есть в кавычках) **Необязательный**: where — возможны варианты: 'l' — локальный, 'n' — нелокальный, 'a' — все	Функция проверяет существование объекта с указанным именем. Если объект существует, то функция возвращает 1, в противном случае 0. С помощью флага, можно задавать область поиска. По умолчанию используется вариант 'a'.

Функции, которые приведены, следует запомнить со всеми возможными аргументами, так как они используются при работе со средой повсеместно.

Упражнение

Надеюсь вы не удалили последствия предыдущего упражнения. Вызовите команду whos. В ответ вы увидите следующее

```
-->whos
Name                Type         Size         Bytes

$                   polynomial   1 by 1       56
%driverName         string*      1 by 1       40
%e                  constant     1 by 1       24
%eps                constant     1 by 1       24
%exportFileName     constant*    0 by 0       16
%F                  boolean      1 by 1       24
%f                  boolean      1 by 1       24
%fftw               boolean      1 by 1       24
%gui                boolean      1 by 1       24
%helps              constant*    0 by 0       16
%i                  constant     1 by 1       32
%inf                constant     1 by 1       24
%io                 constant     1 by 2       32
%modalWarning       boolean*     1 by 1       24
%nan                constant     1 by 1       24
%pi                 constant     1 by 1       24
%pvm                boolean      1 by 1       32
%s                  polynomial   1 by 1       56
%T                  boolean      1 by 1       24
```

```
%t                    boolean       1 by 1        24
%tk                   boolean       1 by 1        16
%toolboxes            string*       1 by 1        40
%toolboxes_dir        string*       1 by 1        136
%z                    polynomial    1 by 1        56
atomsguilib           library                     320
atomslib              library                     856
cacsdlib              library                     4000
compatibility_functilib library                   4216
corelib               library                     688
data_structureslib    library                     464
datatipslib           library                     760
demo_toolslib         library                     544
demolist              string*       15 by 2       4648
development_toolslib  library                     496
differential_equationlib library                  448
dynamic_linklib       library                     744
elementary_functionslib library                   2984
fft                   fptr                        40
fileiolib             library                     624
functionslib          library                     752
genetic_algorithmslib library                     648
graphic_exportlib     library                     392
graphicslib           library                     3968
guilib                library                     488
help                  function                    5368
helptoolslib          library                     752
home                  string        1 by 1        96
integerlib            library                     1416
interpolationlib      library                     336
iolib                 library                     392
jvmlib                library                     296
linear_algebralib     library                     1448
m2scilib              library                     352
maple2scilablib       library                     288
matiolib              library                     328
modules_managerlib    library                     704
MSDOS                 boolean       1 by 1        16
n1                    constant      1 by 1        24
n2                    constant      1 by 1        24
n3                    constant      1 by 1        24
n4                    constant      1 by 3        40
n5                    string        1 by 1        48
n6                    boolean       1 by 1        24
n7                    boolean       1 by 1        24
neldermeadlib         library                     1168
optimbaselib          library                     1024
optimizationlib       library                     696
optimsimplexlib       library                     1248
output_streamlib      library                     360
overloadinglib        library                     15712
parameterslib         library                     424
polynomialslib        library                     904
pvmlib                library                     248
PWD                   string        1 by 1        96
SCI                   string        1 by 1        96
scicos_autolib        library                     408
scicos_utilslib       library                     576
SCIHOME               string        1 by 1        184
scinoteslib           library                     272
signal_processinglib  library                     1888
simulated_annealinglib library                    600
soundlib              library                     544
```

```
sparselib              library                456
special_functionslib   library                304
spreadsheetlib         library                328
statisticslib          library                1360
stringlib              library                648
tclscilib              library                384
texmacslib             library                312
timelib                library                520
TMPDIR                 string       1 by 1    128
uitreelib              library                512
umfpacklib             library                456
whos                   function               15416
xcoslib                library                928
```

В данном списке перечислены объекты, которые были объявлены средой во время инициализации и во время сессии. Из данного списка пользователь получает информацию о типе объекта, его размере в полях (если это массив) и размере в байтах. Именно поэтому данный список называется *длинным*. Попытаемся во всем этом многообразии отыскать только те переменные, которые мы объявили в предыдущем упражнении. Для этого введем команду

```
--> whos -name n
Name                   Type         Size         Bytes

n1                     constant     1 by 1       24
n2                     constant     1 by 1       24
n3                     constant     1 by 1       24
n4                     constant     1 by 3       40
n5                     string       1 by 1       48
n6                     boolean      1 by 1       24
n7                     boolean      1 by 1       24
neldermeadlib          library                   1168
```

Обратите внимание, что функция вернула список из объектов, начинающихся на символ 'n'. Среди таких объектов оказалась также библиотека функций neldermeadlib.

К сожалению функция whos не поддерживает одновременное использование своих параметров, поэтому в каждый отдельный вызов вы можете искать объект либо по его имени, либо по типу данных, которые он хранит. Попытаемся найти все объекты, хранящие строки, для чего введем

```
->whos -type string
Name                   Type         Size         Bytes

%_opt                  string       1 by 1       48
%_sel                  string       1 by 1       48
%driverName            string*      1 by 1       40
%toolboxes             string*      1 by 1       40
%toolboxes_dir         string*      1 by 1       136
demolist               string*      15 by 2      4648
home                   string       1 by 1       96
n5                     string       1 by 1       48
PWD                    string       1 by 1       96
SCI                    string       1 by 1       96
SCIHOME                string       1 by 1       184
TMPDIR                 string       1 by 1       128
```

Обратите внимание на символ звездочки у некоторых переменных в колонке их типа. Данный символ означает, что переменная глобальная, то есть зона ее видимости распространяется на все функции, загруженные в данной сессии. Мы вернемся к этому вопросу, когда будем изучать программирование в среде, а пока просто запомните смысл этого символа.

Отметим, что аргументы функции также можно передавать в скобках, несмотря на то, что об этом не было упомянуто в таблице выше. Делается это так

```
->whos('-type','string')
Name                Type          Size          Bytes

%_opt               string        1 by 1        48
%_sel               string        1 by 1        48
%driverName         string*       1 by 1        40
...
```

Данный способ является уже устаревшим и мы все же рекомендуем вам им не пользоваться, но и не запрещаем.

Мы научились искать объекты, однако, еще не научились искать только те, которые были объявлены в данной сессии. Для этого следует использовать команду who_user

```
-->who_user
 User variables are:

 whos     n7       n6       n5       n4       n3       n2       n1       help
 home

 Using 2636 elements ouf of 4990727
```

Обратите внимание, что функция вернула список объектов текущей сессии. Несмотря на то, что мы не создавали объекты help, whos и home, среда все равно отнесла их к пользовательским. Так как никакой информации об объектах мы теперь не наблюдаем, то данная запись называется *короткой*. Обратите внимание, что внизу отображается информация о заполненности стека. Отметим также, что функция в версии для Windows записывает этот массив в ans (или другой адресат). В версии для Unix-подобных систем этого замечено не было.

Обратите внимание на достоинства и недостатки функций whos и who_user: первая выводит исчерпывающую информацию о стеке и позволяет искать объекты в стеке, однако, вся информация об объектах иногда бывает не нужна; вторая функция возвращает только пользовательские переменные в коротком виде, что бывает полезно для проверки свободных имен.

Функция who сочетает в себе особенности whos() и who_user. Мы предлагаем читателю утолить свое любопытство и поэкспериментировать с вызовом функции who() с разными наборами аргументов, которые, в отличие от whos, можно комбинировать.

Теперь выясним тип объекта help.

```
-->typeof(help)
 ans  =

 function
```

О том, что это функция, вы могли догадаться ранее, когда вызывали, например, help type, где строка type являлась аргументом. Обратите внимание, что, в отличие от функции type, функция typeof возвращает не число, а строку, что намного понятнее человеку. Коды функции type очевидно намного полезнее в программировании сценариев.

Защитим наши переменные

```
-->predef('a')
 ans  =

   79.   90.
```

В ответ функция возвратила два значения: первое — сколько переменных было защищено до вызова и второе — сколько защищенных переменных стало.

Если вы попытаетесь теперь удалить, например, переменную n1, то получите в ответ сообщение об ошибке

```
-->clear n1
 !--error 13
Переопределение неизменной переменной.
```

С этого момента мы натыкаемся на одно неудобство функции predef(), а именно то, что она не позволяет защищать переменные и снимать защиту с переменных по отдельности. В результате чего необходимо быть очень внимательным, вызывая функцию clear и очень тщательно обдумывать какие переменные стоит защищать, а какие не стоит.

Давайте преднамеренно совершим очень неприятную ошибку, а именно освободим стек от библиотек с системными функциями. Не следует бояться, что мы разрушим среду, так как данная проблема решается простым перезапуском программы. Выполните команду

```
-->predef('c')
 ans  =

   90.    13.
```

Обратите внимание, что осталось защищенными только 13 объектов. Теперь вызовите функцию очистки

```
clear
```

Как вы знаете, без аргументов функция удаляет все незащищенные объекты. Теперь попытаемся воспользоваться функцией whos

```
-->whos
 !--error 4
Неизвестная переменная: whos
```

Очевидно, что функция перестала работать. Такой же результат вы можете наблюдать и с функцией help. Это произошло потому, что неосторожным вводом команды clear мы выгрузили все библиотеки с функциями, которые прописаны в главном сценарии среды. Так как вы не исправляли сам сценарий, то при следующем его запуске все возвратится на круги своя, а пока вы остались наедине со средой с минимальным набором компонент. Тем не менее, функция who должна работать, потому что она защищена от такой ситуации, и вы можете понаблюдать, что осталось в памяти.

Закройте среду и запустите ее заново.

В последнем упражнении последний пример показывает как важно аккуратно обращаться с памятью. Давайте разберем ситуацию «а как удалить все переменные с n1 по n7?». Вспомним правило, по которому работает стек: первым зашел — последним вышел. Это означает, что объекты как бы наслаиваются друг на друга при каждом объявлении, следовательно наши переменные в стеке были одними из последних.

Можно было запомнить сколько объектов было защищено до вызова функции predef('a') и вернуть защиту именно этому числу объектов вызовом predef(n) (если вы обратили внимание, в упражнении это число было равно 79, то есть введя predef(79)). Соответственно предыдущее число защищенных объектов мы можем получить вызвав predef() без аргументов. Если на верхушке стека оказывались бы только наши переменные, то с чистой совестью мы можем вызвать команду clear, так как библиотеки мы уже защитили. Если в верхушке стека вместе с нашими переменными есть еще какие-то объекты, то clear вызывать уже нельзя и следует перечислить удаляемые объекты через пробел в качестве аргументов.

Самостоятельно

С этого момента в учебнике будут появляться блоки, в которых читателю предлагается закрепить полученные знания. В этом блоке предлагается решить следующую задачу.

Создайте десять переменных, хранящих нули, с именами v#, где символ '#' означает число от 1 до 10. Переменные следует создавать строго в прямой последовательности;
В переменные v3, v5 и v9 запишите значения системных переменных, хранящих строковые значения. Выведите список предопределенных констант (то есть начинающихся с символа '%') и отыщите среди них математическую константу Пи. Значение Пи запишите в переменную v1;
Очистите память от переменных v2 и v10;
Защитите все переменные;
Очистите память от всех созданных вами переменных, кроме тех, что хранят строки, любым кажущимся для вас наиболее рациональным способом, но только так, чтобы в среде оставались все объекты проинициализированные перед началом сессии, и так, чтобы впоследствии они оставались защищенными;
Если оставшиеся пользовательские переменные остались защищенными, то снимите с них защиту;
Ответьте на вопрос «Какой тип данных хранит константа %fftw?».

Целые числа[править]

Вы могли обратить внимание, что научились создавать вещественные числа, но так и не научились создавать целые числа. В Scilab целые числа возможно создавать только через специальные функции. Во всех остальных случаях числовому значению всегда будет присваиваться вещественный тип данных.

Для хранения целого числа в памяти может быть использовано разное число битов, а именно 8, 16 и 32. От количества используемых битов зависит диапазон целых чисел. Кроме того, имеется возможность включения и отключения знакового бита, что бывает полезно, когда отрицательные целые числа не требуются.

В таблице ниже приведены функции для создания целых чисел и диапазоны возможных значений. Во всех случаях функция возвращает целое число, указанное в качестве аргумента, с определенным способом его хранения в памяти.

Имя функции	Тип целого числа	Диапазон	Код представления
int8()	знаковое 8-битное число		1
uint8()	беззнаковое 8-битное число		11
int16()	знаковое 16-битное число		2
uint16()	беззнаковое 16-битное число		12
int32()	знаковое 32-битное число		4

uint32()	беззнаковое 32-битное число		14

Какой тип целочисленной переменной следует применить, пользователь решает сам. Целые числа могут быть использованы как в чисто математических задачах, так и программировании, в качестве счетчиков.

Для работы с целочисленными переменными существует две функции:

- **iconvert(X, itype)** — меняет представление в памяти в общем случае матрицы из вещественных или целых чисел. Функции при этом следует передавать имя этой матрицы X и код внутреннего представления целого числа itype из последней колонки таблицы выше;
- **inttype(X)** — возвращает код внутреннего представления в общем случае матрицы целочисленных данных.

Упражнение

Создадим целочисленную переменную

```
--> integer=uint8(16)
integer  =

 16
```

Обратите внимание на то, что пропала точка, разделяющая целую и дробную части. Прибавим к целому числу вещественное

```
-->integer=integer+32.6
 integer  =

 48
```

Обратите внимание, что вещественное число не влияет на тип данных. Теперь из результат вычтем 49.

```
-->integer=integer-49
 integer  =

 255
```

Вместо −1 мы получили 255. Так как мы используем беззнакоый тип целого числа этого и следовало ожидать. Ситуация, когда мы выходим за диапазон возможных значений, называется *переполнением*. Переполнение практически всегда нежелательное явление, поэтому следует четко представлять в каких диапазонах решается та или иная задача.

Отметим, что в функциях можно указывать более сложные алгебраические конструкции, например

```
--> integer=int32(3*2-45^4)
 integer  =

 -4100619
```

Введем команду, устанавливающую тип целочисленной переменной

```
-->inttype(integer)
```

```
 ans  =
   4.
```

Посмотрев по таблице код внутреннего представления, приходим к выводу, что это действительно целая 32-битная знаковая переменная. Вызовем команду, меняющую внутреннее представление переменной, например, отключим знаковый бит

```
-->iconvert(integer,14)
 ans  =
  4290866677
```

Теперь знаковый бит кодирует часть числа, увеличив тем самым возможный диапазон, однако, мы потеряли исходное число. Теперь уберем часть битов

```
-->iconvert(integer,2)
 ans  =
  28149
```

Разберемся, почему получилось именно такое число. Несложно представить двоичную форму числа 4290866677, а именно

1111 1111 1100 0001 0110 1101 1111 0101.

Преобразовав внутреннее представление в 16-битное, мы отсекли 16 старших битов (находящихся слева) и получили

0110 1101 1111 0101.

Если выполнить обратное преобразование в десятиричную систему счисления, то получается как раз число 28149.

Комплексные числа[править]

Теперь поговорим о комплексных числах. Показанный ранее способ ввода комплексного числа не всегда удобен и требует ввода знаков '+' и '%i'. Ввод комплексного числа в значительной мере можно упростить, если воспользоваться функцией **complex(a, b)**, в которой в качестве аргументов указать действительную и мнимую части соответственно. Согласно справочной информации, данная функция позволяет избежать проблем в объявлении комплексных переменных, в которых возможно появление констант %nan и %inf.

Попутно познакомим читателя с константами %nan и %inf. Константа %nan (англ. *Not-a-Number*) кодирует особое состояние вещественного числа и возвращается, когда результат не может быть определен числовым значением (например, при неопределенности типа ноль на ноль). По умолчанию, режим, при котором в случае неопределенного результата формируется %nan, отключен и, если вы попробуете разделить на ноль, интерпретатор вернет сообщение об ошибке. Для включения режима необходимо вызвать функцию **ieee()**, где в качестве аргумента следует передать 2.

```
--> ieee(2)
--> 0/0
ans =
   Nan
```

Константа %inf (англ. *Infinity*) является машинным представлением бесконечно большого числа. Обе эти константы могут быть использованы для объявления комплексных чисел, например

```
--> complex(%nan,%inf)
ans  =
```

```
  Nan + Infi
```

Покажем ситуацию, в которой показана необходимость в функции complex() и которая представлена в качестве примера в справке. Попытаемся объявить следующую комплексную переменную

```
--> %inf+%i*%inf
ans =
   Nan + Infi
```

Очевидно, что инициализация прошла с ошибкой, потому что в действительной части ожидалось %inf. Это произошло потому, что системой произведение %i*%inf интерпретируется как (0+%i)*(%inf+%i*0), что дает промежуточное выражение 0*%inf = %nan.

Корректно проинициализировать значение в этом случае возможно только с помощью функции

```
--> complex(%inf,%inf)
ans  =
  Inf + Infi
```

Для работы с комплексными числами также существует небольшой набор функций:

- **conj(X)** — возвращает комплексное число сопряженное числу X;
- **real(X)** — возвращает действительную часть комплексного числа X;
- **imag(X)** — возвращает мнимую часть комплексного числа X;
- **isreal(X)** — возвращает логическое %T, если передаваемый аргумент является комплексным числом, %F в противном случае.
- **imult(X)** — умножает мнимую единицу на аргумент. Согласно справочной информации, данную функцию ркомендуется использовать, когда приходится иметь дело с константами типа %nan и %inf, появляющихся в комплексном числе.

Предлагаем читателю самостоятельно попробовать все функции над комплексными числами.

Строки[править]

Строковый тип данных образуется заключением символов в одинарные или двойные кавычки.

```
--> a='Scilab', b="is smart"
```

В памяти строка представляет собой массив кодов символов, из которых она образуется. Однако, после объявления строки, она представляется для пользователя единым куском, поэтому без специальных функций для работы со строками вы не сможете добраться до отдельно взятого символа строки.

Функций для работы со строками уже несколько больше, чем для работы, например, с теми же комплексными числами, и простое их перечисление будет просто не эффективным способом их изучить. Мы познакомимся с этими функциями чуть позже, когда будем решать практические задачи.

Пока вам следует запомнить, что строки сами по себе поддерживают лишь операцию конкатенации (объединения), которая вызывается с помощью оператора сложения. Например,

```
--> a+' '+b
ans  =
```

```
Scilab is smart
```

Переменная ans[править]

Переменная ans (от англ. *answer* — ответ) — это зарезервированная переменная, в которую сохраняется любой последний «безымянный» результат. Переменная ans является особой, например, на нее не действуют функции predef и clear. Зона ее видимости всегда остается локальной.

Переменная ans может быть использована для проведения цепочки вычислений, промежуточные результаты которых вам не нужны.

```
-->2*2
 ans  =
    4.
-->ans+4
 ans  =
    8.
-->ans/ans
 ans  =
    1.
```

Матрицы и векторы[править]

Вектор в Scilab — это упорядоченная совокупность элементов (одномерный массив) одного типа данных. Упорядоченность для пользователя в этом смысле проявляется в том, что к каждому элементу вектора можно обратиться по его уникальному порядковому номеру или *индексу*. В среде Scilab все индексы начинаются с единицы, что немного не привычно, так как например в программировании на языке Си те же индексы массивов начинаются с нуля.

Ранее мы уже не раз создавали вектора и вы уже знаете, что для этого нужно элементы заключить в квадратные скобки, то есть

```
Vector = [e1, e2, e3]; // Вектор с тремя элементами в виде переменных с именами e1, e2 и e3
```

При объявлении вектора совершенно не обязательно отделять их друг от друга запятыми — достаточно простого пробела. Например, следующее объявление не приведет к синтаксической ошибке

```
Vector = [e1 e2 e3];
```

Вектор, элементы которого представляют собой числовую последовательность в виде арифметической прогрессии, может быть создан особым образом. Для этого используется конструкция

```
<начальное значение>:<шаг>:<конечное значение>
```

Например, создадим вектор с начальным значением −5, конечным значением 10 и шагом между элементами 2

```
--> -5:2:10
ans  =
 - 5.  - 3.  - 1.    1.    3.    5.    7.    9.
```

Вообще говоря, шаг можно не указывать. В этом случае его значение по умолчанию принимается равным 1. Если шаг не указывается, то начальное значение всегда должно быть меньше конечного, иначе Scilab создаст пустой вектор.

```
-->1:5
 ans  =
   1.    2.    3.    4.    5.
```

Вектор может быть записан столбцом или строкой. В первом случае он называется вектор-столбцом, а во втором — вектор-строкой. От того, как представлен вектор (столбцом или строкой), зависит применимость некоторых операторов. Например, из линейной алгебры вы знаете что скалярное произведение двух векторов возможно только, если первый множитель представлен строкой, а второй — столбцом.

Выше мы создавали векторы-строки. Векторы-столбцы создаются путем отделения каждого элемента точкой с запятой, то есть

```
-->[1; 2; 3]
 ans  =
   1.
   2.
   3.
```

Вектор-столбец также может быть получен операцией транспонирования (оператор в виде апострофа)

```
-->[1 2 3]'
 ans  =
   1.
   2.
   3.
```

что иногда бывает намного удобнее, чем создание с помощью точки с запятой.

Так как каждый элемент внутри вектора имеет свой идентификатор в виде индекса, то к нему можно обратиться. Для того, чтобы обратиться к элементу вектора, необходимо записать имя вектора, а затем в круглых скобках указать индекс.

```
-->A=1:5
 A  =
   1.    2.    3.    4.    5.
-->A(4) // Обращаемся к элементу 4 внутри вектора A
 ans  =
   4.
-->A=A';
-->A(4) // Обратите внимание, что операция транспонирования не влияет на индексацию
 ans  =
   4.
```

В любой момент вы можете удалить элемент вектора. Для этого достаточно на желаемой позиции создать, так называемую, *пустую матрицу* в виде конструкции '[]'.

```
-->A
 A  =
   1.    2.    3.    4.    5.
-->A(2)=[] // Удаляем элемент 2 внутри вектора A
```

```
 A  =
   1.    3.    4.    5.
```

Если вы хотите обратиться к последнему элементу вектора, но не знаете его размер, вы можете воспользоваться служебным символом в виде знака доллара '$'.

```
-->A($)
 ans  =
   5.
```

Мы не зря начали свое изложение именно с понятия вектора, так как матрица по сути является расширением этого понятия.

Матрица в Scilab — это двухмерный массив однотипных элементов. Можно понимать матрицу как несколько векторов-строк, записанных столбцом.

Создать матрицу в Scilab можно одним из нескольких способов:

- Матрицу можно создать из составляющих ее элементов;
- Из имеющихся векторов, упорядочив их строками или столбцами;
- Одной из специальных функций.

В общем случае синтаксическая конструкция имеет вид

```
[x11, x12, …, x1n; x21, x22, …, x2n; …; xm1, xm2, …, xmn]
```

Таким образом, вы создаете векторы-строки, которые отделяете точкой с запятой. Запятая в этом случае, как и с вектором, не обязательна.

```
-->A=[1 2; 3 4] // создадим матрицу из составляющих ее элементов
 A  =
   1.    2.
   3.    4.
```

Матрицу можно собрать из векторов. Вот два поучительных примера

```
--> [1:5; 5:-1:1]
ans =
   1.  2.  3.  4.  5.
   5.  4.  3.  2.  1.
--> [(1:5)' (5:-1:1)']
ans =
   1.  5.
   2.  4.
   3.  3.
   4.  2.
   5.  1.
```

Обратите внимание на второй случай, как сначала мы транспонируем векторы, создаваемые особой конструкцией, а затем присоединяем их столбцами друг к другу.

В любом случае, вы всегда должны помнить, что нельзя допускать конфликты с размерами векторов. Например, если вы образуете матрицу из векторов-строк, то число элементов в них должно быть одинаковым, иначе произойдет ошибка.

При работе с матрицей сохраняются абсолютно все принципы, что ранее были показаны ранее с векторами, с той разницей, что каждый элемент имеет уже два индекса. Первый из них мы будем называть строковым, а второй — столбцовым. Первым всегда идет строковый индекс.

Разберем эти принципы в следующем упражнении.

Упражнение

Для начала сгенерируем матрицу, над которой будем экспериментировать. Так как нам по существу все равно, чем она будет заполнена, воспользуемся стандартной функцией rand(), которая генерирует массивы, заполняя их псевдослучайными числами.

```
-->A=rand(4,4) // создает матрицу 4x4
 A  =
   0.2113249   0.6653811   0.8782165   0.7263507
   0.7560439   0.6283918   0.0683740   0.1985144
   0.0002211   0.8497452   0.5608486   0.5442573
   0.3303271   0.6857310   0.6623569   0.2320748
```

Обратимся к элементу, стоящему во второй строке и третьем столбце

```
-->A(2,3)   // 2-я строка, 3-й столбец
 ans  =
   0.0683740
```

Кроме простого обращения к элементам, можно обратиться ко всему столбцу или строке. Для этого используется символ двоеточия ':', который кодирует слово «все». Например,

```
-->A(:,3)  // обращение к 3-му столбцу
 ans  =
   0.8782165
   0.0683740
   0.5608486
   0.6623569
```

При использовании такого обращения на первых порах очень легко запутаться и вообще перестать понимать к чему вы обращаетесь. Чтобы этого не происходило, начинайте про себя проговаривать «я обращаюсь к <число> столбцу (строке)», при этом держа взгляд не на двоеточии, а на индексе с числом.

Теперь обратимся к последней строке

```
-->A(4,:)  // обращение к 4-ой строке
 ans  =
   0.3303271   0.6857310   0.6623569   0.2320748
```

Опять же, если вы не знаете размера матрицы или не хотите вникать в такую подробность, то можете воспользоваться знаком доллара. Например, обратимся сначала к последнему столбцу, а потом к элементу, стоящему на пересечении последней строки и столбца

```
-->A(:,$)
 ans  =
   0.7263507
   0.1985144
   0.5442573
   0.2320748

-->A($,$)
```

```
 ans  =
   0.2320748
```

В качестве индексов матриц могут выступать векторы. В этом случае они выполняют роль диапазона индексов по строке или столбцу, что позволяет извлекать из матрицы сразу несколько отдельных элементов. Извлечем из нашей матрицы середину.

```
-->A([2 3],[2 3])  // извлекаем из A элементы (2,2), (2,3), (3,2) и (3,3)
 ans  =
   0.6283918   0.0683740
   0.8497452   0.5608486
```

Следующая конструкция может показаться очень запутанной и непонятной

```
-->A([1 $],[2 3 4])
 ans  =
   0.6653811   0.8782165   0.7263507
   0.6857310   0.6623569   0.2320748
```

Если вы внимательно ее изучите, то догадаетесь, что Scilab извлекает элементы с адресами, являющиеся сочетаниями без повторений двух множеств, образуемых векторами. Чтобы не путаться при этом, вам всегда нужно представлять себе, что Scilab извлекает элементы построчно. Разберем предыдущий пример. В качестве строкового индекса мы представили вектор [1 $], который говорит нам, что Scilab обработает строки исходной матрицы с номерами 1 и последним номером. Далее возникает вопрос «из каких столбцов будут извлекаться элементы?», на который отвечает диапазон столбцовых индексов [2 3 4], то есть из столбца 2, столбца 3 и столбца 4. Таким образом полный набор сочетаний с учетом того, что обработка идет построчно, будет такова: (1,2), (1,3), (1,4), диапазон по столбцам закончился — переходим на новую строку: ($,2), ($,3), ($,4). Таким образом, вы можете с помощью векторов пропускать неугодные вам строки или столбцы.

Важно, чтобы индексы не противоречили реальным размерам матрицы, в противном случае система вернет ошибку обращения к матрице. Для того чтобы извлекать миноры из исходной матрицы, можно воспользоваться конструкцией с двоеточием, определяя диапазоны индексов с шагом. Например так,

```
-->A([2:4],[1:3])
 ans  =
   0.7560439   0.6283918   0.0683740
   0.0002211   0.8497452   0.5608486
   0.3303271   0.6857310   0.6623569
```

Последняя запись эквивалентна следующей

```
-->A([2 3 4],[1 2 3])
 ans  =
   0.7560439   0.6283918   0.0683740
   0.0002211   0.8497452   0.5608486
   0.3303271   0.6857310   0.6623569
```

В матрице вы также можете удалять строки и столбцы. Для этого вам нужно к ним обратиться и на их место поставить пустую матрицу. Удалим столбец 3 и строку 2.

```
-->A
 A  =
   0.2113249   0.6653811   0.8782165   0.7263507
   0.7560439   0.6283918   0.0683740   0.1985144
   0.0002211   0.8497452   0.5608486   0.5442573
   0.3303271   0.6857310   0.6623569   0.2320748
```

```
-->A(:,3)=[]; A(2,:)=[]
 A  =
   0.2113249   0.6653811   0.7263507
   0.0002211   0.8497452   0.5442573
   0.3303271   0.6857310   0.2320748
```

Векторами и матрицами все не заканчивается. Пользователь может создавать массивы больших размерностей, например, создадим трехмерный массив

```
-->B=rand(2,2,2)
 B  =
(:,:,1)
   0.4419389   0.1607456
   0.2987960   0.4816621
(:,:,2)
   0.8496411   0.6894972
   0.1152915   0.6928246
```

Для того, чтобы представить себе трехмерный массив, воспользуйтесь следующей аналогией. Представьте себе лист бумаги, на котором вы мысленно можете записать матрицу. Из примера это матрица с адресом (:,:,1). Один лист представляет первые два измерения (строки и столбцы матрицы). Теперь вы откладываете первый лист и берете второй, на котором пишите уже вторую матрицу (:,:,2). Затем второй листок вы накладываете поверх первого. Если бы в нашем примере было, скажем, не 2 матрицы, а 30, то столб из листков был бы относительно высоким. Именно высота этого столба представляет третье измерение.

Не теряя общности, вы можете продолжать рассуждения для четырехмерного массива, когда четвертым измерением предстает одна из сторон стола, на котором в ряд вы выкладываете столбики бумаги — трехмерные массивы. При пятом измерении вы раскладываете столбики уже по всему столу, а при шестом вы уже имеете много столов и так далее.

Количество измерений массива упирается в текущий размер стека, иными словами, пока есть память для хранения таких массивов. Хотя вам никто не запрещает пользоваться многомерными массивами, по возможности лучше держаться в пределах двух-, трехмерных массивов. Это связано не сколько с ограничением размера памяти, сколько неудобством ручной обработки таких массивов. Например, ввести вручную многомерный массив невозможно — для этого нужно воспользоваться специальной функцией **hypermat()**.

```
// Пример применения функции hypermat(D[, V])
// D - вектор, каждый элемент которого представляет размер измерения многомерного массива
// [, V] - необязательный вектор, значениями которого заполняется многомерный массив.
// Его размер должен быть равен количеству всех элементов многомерного массива. Если он не указан, многомерный массив заполняется нулями.
-->A=hypermat([2 2 3])
 A  =
(:,:,1)
   0.   0.
   0.   0.
(:,:,2)
   0.   0.
   0.   0.
(:,:,3)
   0.   0.
   0.   0.
-->A=hypermat([2 2 3],1:12)
 A  =
(:,:,1)
```

```
   1.  3.
   2.  4.
(:,:,2)
   5.  7.
   6.  8.
(:,:,3)
   9.  11.
   10.  12.
```

Все принципы работы с массивами, рассмотренные ранее, распространяются и на многомерный случай.

Операции над матрицами[править]

Для матриц и векторов предусмотрены следующие операции:

- сложение '+' — поэлементное сложение двух матриц. Матрицы должны быть одинаковых размеров;
- вычитание '-' — поэлементное вычитание элементов двух матриц. Матрицы должны быть одинаковых размеров;
- транспонирование " ' " (оператор в виде апострофа) — представление столбцов матрицы строками;
- матричное умножение '*' — умножение одной матрицы на другую. Матрицы должны быть совместны (число столбцов первого множителя должно быть равно числу строк второго). Если один из операндов представлен переменной, то каждый элемент матрицы будет помножен на эту переменную;
- возведение в степень '^' — умножение матрицы на себя n-ое количество раз;
- правое деление '/' — первый операнд делится на второй. Матрицы должны быть совместны;
- левое деление '\' — второй операнд делится на первый. Матрицы должны быть совместны;
- поэлементное умножение '.*' — матрицы перемножаются поэлементно (не путать с матричным умножением). Матрицы должны иметь одинаковые размеры;
- поэлементное возведение в степень '.^' — каждый элемент матрицы возводится в степень (не путать с возведением в степень матрицы);
- поэлементное правое деление './' — элементы первой матрицы делятся на элементы второй (не путать с делением матриц). Матрицы должны быть одинаковых размеров;
- поэлементное левое деление '.\' — элементы второй матрицы делятся на элементы первой (не путать с делением матриц). Матрицы должны быть одинаковых размеров.

Обратите внимание, что во всех поэлементных операциях присутствует точка. К массивам также применимы логические операции, которые производятся исключительно поэлементно.

Самостоятельно

Создайте матрицу с помощью функции rand() размером 4x4 и присвойте ей имя 'A'.
Создайте вектор 'v' из последней строки матрицы A. Помножьте вектор v на матрицу A и результат поставьте на место столбца 1 матрицы A (не забывайте о существовании операции транспонирования).
Помножьте матрицу A на вектор v (не забывайте об условии «совместности»). Результат запишите во вторую строку матрицы A.
Создайте единичную матрицу того же размера, что и A, и присвойте ей имя 'B'.
Умножьте матрицу A на B. Затем поэлементно умножьте матрицу A на B. Объясните полученные результаты.
В матрице удалите последний столбец и последнюю строку так, чтобы ее размер стал равным 3x3.
Заполните диагональные (и по главной, и по побочной) элементы нулями любым кажущимся для вас рациональным способом (не забывайте и о существовании редактора переменных).
Поэлементно возведите матрицу A в квадрат.
Составьте из ненулевых элементов матрицы A новую матрицу 2x2 по следующему правилу: первый столбец матрицы представлен ненулевыми элементами второго столбца, ненулевой элемент первого столбца должен встать на позицию (2,2), а ненулевой элемент третьего столбца на оставшуюся свободную позицию. Результат запишите в A.
Возведите матрицу в степень куба.

Примечание: в пункте 9 мы надеемся, что находчивый читатель догадался до следующей конструкции

```
-->A=[A([1 3],2) A(2,[3 1])`]
```

Если вы поступили по-другому, то не переживайте, потому что в Scilab одна и та же задача может решаться многими способами, и на ранних порах важно достигли ли вы требуемого результата.

Функции для работы с матрицами[править]

Так как все объекты в Scilab являются представлением массивов данных, то большинство функций «заточены» на работу с объектами как с массивами. Для работы с векторами и матрицами существует большое количество внутренних функций и простое их перечисление так же не эффективно в рамках обучения работе в среде. Необходимо пробовать их на практике и учиться на результатах.

Тем не менее, можно выделить ограниченный набор функций для работы с массивами, используемые так часто, что их заучивание является приоритетной целью. Именно этому мы посвятим данный раздел. Ниже в таблице перечислены функции, которые рекомендуется запомнить.

Имя функции	Аргументы	Назначение
size(V,[flag])	V — массив; flag — возможны следующие варианты: 'r' или 1 — возвращает число строк; 'c' или 2 — возвращает число столбцов	Функция возвращает размеры массива в виде вектора, если не указаны флаги.
length(V)	V — массив	Функция возвращает число элементов в массиве
max(V,[flag]) **max(V1,V2,…,Vn)**	V — массив	Функция возвращает элемент с максимальным значением в указанном массиве или группе массивов (см. второй вариант вызова).
min(V,[flag])	Аналогично функции **max()**	Функция возвращает элемент с минимальным значением в указанном массиве или группе массивов.

Разреженная матрица[править]

Несложно догадаться, что общий объем памяти, необходимый для хранения матрицы, равен количеству элементов матрицы помноженному на объем памяти, который необходимо выделить для хранения типа данных, записываемого в эту матрицу. Такой способ хранения приемлем, когда каждый элемент несет в себе данные, однако, на практике имеют место быть особые виды матриц, называемые разреженными.

В разреженных матрицах большая часть элементов несет нулевое значение, что делает обычное выделение памяти на всю матрицу не эффективным. В системе Scilab предусмотрено создание разреженных матриц, в котором хранение элементов оптимизировано: память выделяется только на ненулевые элементы. Создать разреженную матрицу можно с помощью функции

```
sparse([i1 j1;i2 j2;...],[n1,n2,...])
```

В качестве аргументов данной функции необходимо передать два вектора:

- в первом векторе необходимо перечислить адреса ненулевых элементов;

- в втором векторе необходимо перечислить значения, которые будут записаны по данным адресам.

```
// Пример
--> A=sparse([1 1;2 1;4 4;1 3],[25 26 31 32])
 A =

(   4,   4)  sparse matrix

(   1,   1)    25.
(   1,   3)    32.
(   2,   1)    26.
(   4,   4)    31.
```

Обратите внимание на особый формат представления разреженной матрицы. Разреженная матрица ничем не отличается, кроме способом выделения памяти, следовательно к этим матрицам применимы все операции рассмотренные ранее.

```
//Пример
-->B=zeros(4,4);
-->A+B
 ans  =
   25.    0.    32.    0.
   26.    0.    0.     0.
   0.     0.    0.     0.
   0.     0.    0.     31.
-->A(4,1)
 ans  =
(    1,    1) zero sparse matrix
-->A(1,1)
 ans  =
(    1,    1) sparse matrix

(    1,    1)       25.
```

Иногда формат вывода разреженных матриц, используемый по умолчанию, не всегда удобен, поэтому специально для них используется функция **full()**.

```
-->full(A)
 ans  =
   25.    0.    32.    0.
   26.    0.    0.     0.
   0.     0.    0.     0.
   0.     0.    0.     31.
```

Списки[править]

Список в Scilab — это массив (или структура), состоящий из элементов разных типов данных.

Список может быть создан вручную или с помощью одной из следующих функций:

- **list()** — создает список, поля которого могут содержать произвольный тип данных;
- **tlist()** — создает *типизованный список*;
- **mlist()** — создает матричноориентированный типизованный список.

Типизованные списки используются главным образом для создания классов объектов в рамках объектно-ориентированного программирования.

Упражнение

Для начала создадим простой список с помощью функции **list()**. Для этого достаточно вызвать функцию и в качестве аргументов перечислить объекты списка. При этом мы не ограничены типами объектов.

```
-->lt=list(complex(12,6),[23 12],[1 2;3 4]);
```

Чтобы обратится к элементу списка, достаточно указать его индекс. Например, обратимся к первому элементу и к элементу (4,4) матрицы

```
-->lt(1),lt(3)($,$)
 ans  =
   12. + 6.i
 ans  =
   4.
```

Обратите внимание, как происходит обращение к матрице, хранящейся в списке: сначала индекс списка, а затем индекс матрицы. Список может наращиваться с начала и с конца. Для наращивания с начала необходимо обратиться к нулевому элементу списка, после чего добавленный элемент станет первым, а все остальные сместятся на позицию вправо. Для добавления элемента в конец нужно обратится к элементу списка с индексом ($+1).

```
-->lt(0)=25;
-->lt(1)
 ans  =
   25.
-->lt($+1)=24;
-->lt($)
 ans  =
   24.
```

В любой момент можно удалить элемент списка. Для этого нужно воспользоваться специальной функцией **null()**.

```
-->lt(3)=null()
 lt  =
     lt(1)
   25.

     lt(2)
   12. + 6.i

     lt(3)
   1.    2.
   3.    4.

     lt(4)
   24.
```

Во многом работа со списком схожа с работой над вектором. Список **list()** обычно используется, когда данные в нем обезличены. В остальных случаях удобно использовать типизованный список.

Типизованный список[править]

Типизованные списки используются в среде главным образом для определения новых структур данных. Многие функции в Scilab перегружены с помощью типизованных списков. Также некоторые объекты в среде на самом деле являются типизованными списками, просто-напросто это скрыто от глаз пользователя.

Коренным отличием типизованного списка является то, что у него есть поля. Полем по сути является элемент списка, однако, каждому полю списка присваивается уникальный идентификатор или, говоря проще, имя.

Для работы с типизованными списками служат следующие функции:

- **tlist()** — создает типизованный список.
- **fieldnames()** — позволяет вывести все поля типизованного списка.
- **definedfields()** — позволяет вывести поля, в которых есть данные.
- **setfield()** — позволяет установить поле для типизованного списка.
- **getfield()** — позволяет извлечь из поля данные.

Упражнение

Для примера создадим типизованный список, который будет хранить информацию о некотором двигателе. Пусть у этого списка будет три поля: тип двигателя, маркировка, мощность и размерность мощности.

```
-->tt=tlist(['Engine','type','power','dim'],'П61',5,'кВт')
 tt  =
      tt(1)
!Engine  type  power  dim  !
      tt(2)
 П61
      tt(3)
   5.
      tt(4)
 кВт
```

В предыдущем вызове мы создали типизованный список с именем *Engine*. Именно наличие имени отличает этот вид списков от обычных list-списков. В самом вызове вы можете заметить, что первый аргумент является вектором, в котором вы должны последовательно передать имя списка и имена полей этого списка.

Последующие аргументы являются необязательными и в них вы можете проинициализировать поля нового списка. В данном примере мы присвоили полю type строку 'П61', полю power — число 5 и полю dim — строку 'кВт'. Теперь, чтобы обратится к конкретному значению списка, вы должны знать имя поля в котором оно хранится и тип данных того, что в этом поле хранится. Например так

```
-->tt.type
 ans  =
 П61
```

В последнем вызове мы сначала написали имя типизованного списка, к которому хотим обратиться, а затем через точку имя поля, к которому обращаемся.

Типизованные списки наследует все приемы работы, присущие обычным list-спискам. Например,

```
-->tt(2)                // к любому элементу списка можно обратиться по индексу
 ans  =
 П61
-->tt(1)                // при этом обратите внимание, что смысловая часть начинается со второго индекса
 ans  =                 // в первой позиции хранятся имя списка и имена его полей
!Engine  type  power  dim  !

-->tt(1)(1)              // самый простой способ получить имя списка это обратится к вектору, записанному в первой
позиции
 ans  =
 Engine

-->tt(3)=null();         // как и в list-списках элементы удаляются функцией null()
```

```
                    // при этом все элементы, стоявшие после удаляемого, сдвигаются к голове списка
-->definedfields(tt)
 ans  =
   1.    2.    3.

-->tt(3)
 ans  =
 кВт

// при этом удаление элемента (1) в типизованном списке не приводит к ошибке, а просто преобразует
// типизованный список в простой
-->typeof(tt)
 ans  =
 Engine
-->tt(1)=null();
-->typeof(tt)
 ans  =
 list
```

Вы можете задаться вопросом: «зачем нужен типизованный список, если он отличается от обычного списка разве что именем?». Ответ на этот вопрос кроется в его названии «типизованный». Типизованные списки позволяют создавать пользовательские типы данных. На текущем этапе вероятнее всего это вам ни о чем не говорит, но вы все поймете, когда начнете писать программы в Scilab.

Мы пока опустим вопрос о пользовательских типах данных (с ним вы познакомитесь в главе Программирование) и сосредоточимся на особенностях работы с типизованными списками. Хотя к элементам типизованного списка можно обращаться по индексам, в уме вы должны держать, что это следует делать только на уровне разработчика, а при непосредственном использовании пользователь все же будет использовать имена полей. Поля в типизованном списке по желанию можно добавлять. Например, к существующему списку добавим поле номинального напряжения и сразу же проинициализируем его

```
// к сожалению функция setfield() не умеет назначать имена полям,
// поэтому мы это делаем вручную
-->tt(1)($+1)='Voltage'; setfield(5,220,tt);
-->tt.Voltage
 ans  =
   220.
-->fieldnames(tt)      // проверяем имена полей списка
 ans  =
!type    !
!        !
!power   !
!        !
!dim     !
!        !
!Voltage  !
```

Вы наверное уже обратили внимание, что на операции извлечения из списка и присваивание значения элементу в списке наложены две функции **getfield()** и **setfield()** соответственно. Они могут использоваться во всех типах списков, но скорее всего они написаны больше для типизованных списков. Вообще говоря, они не приносят большого преимущества, но их желательно использовать в сценариях, чтобы создать единообразие обращения к спискам. Это увеличивает удобочитаемость кода.

Кроме, условно говоря, tlist-списка, в Scilab есть еще одна разновидность типизованного списка — mlist-список. При обычной работе с этими списками пользователь не почувствует между ними разницы. Разница начинает проявляться при программировании пользовательских структур данных. Этот вопрос мы будем рассматривать чуть позже в главе Программирование.

Из этой части вы должны вынести, что типизованные списки являются надстройками над списками типа list и позволяют создавать пользовательские типы данных с уникальным именем в виде строки.

Кроме того, к элементам типизованного списка можно обращаться по их уникальным именам, что позволяет писать удобные для пользователя интерфейсы.

Другие[править]

В Scilab присутствуют объекты, которые могут быть созданы только с помощью функций, а их внутренняя организация скрыта от пользователя под оболочкой. Такие объекты мы классифицировали как другие, и им посвящен данный раздел.

Полином[править]

Из алгебры вы знаете, что полиномом называется алгебраическое уравнение вида

Полином в среде Scilab может быть определен как объект polynomial, с которым работают специальные функции, например функции поиска корней. Полином можно определить через функцию **poly()**.

```
poly(a,vname,['flag'])
// a — число или матрица (смысл зависит от используемого флага).
// vname — имя символьной переменной в виде строки. В строке используется первые 4 символа.
// ['flag'] — флаг, определяющий принцип расстановки коэффициентов полинома.
// Возможны варианты:
// 'r' или 'roots' — коэффициенты формируются относительно корней полинома, представленных в 'a';
// 'c' или 'coeff' — коэффициенты формируются из значений, представленных в 'a'.
```

Для работы с полиномами существует несколько функций:

- **varn()** — позволяет поменять или узнать символичесуцю переменную у указанного полинома;
- **coeff()** — позволяет выписать коэффициенты полинома. Коэффициенты выписываются, начиная с младшей степени при символической переменной, и записываются в вектор-строку.

Упражнение

Введем полином второго порядка, который мы разрешали ранее в разделе «Работа с сессией». Напомним, что его коэффициенты −10, 4, 2.

```
-->p=poly([-10 4 2],'x','c')
 p  =
              2
  - 10 + 4x + 2x
```

Теперь попробуем поменять символьную переменную на 's'.

```
-->p=varn(p,'s')
 p  =
              2
  - 10 + 4s + 2s
```

Для поиска корней полинома в Scilab существует функция roots(). Применим ее на данном полиноме.

```
-->R=roots(p)
 R  =
  - 3.4494897
    1.4494897
```

Если бы вы знали об этой функции и об объекте polynomial раньше, то упражнение по работе с командным окном показалось бы вам по меньшей мере странным, так как вы добились того же результата всего в две строки. Теперь относительно этих корней сгенерируем тот же полином

```
-->poly(R,'x','r')
 ans  =
          2
 - 5 + 2x + x
```

Обратите внимание на то, что Scilab учел кратность.

```
-->2*ans
 ans  =
            2
 - 10 + 4x + 2x
-->coeff(ans)
 ans  =
 - 10.    4.    2.
```

В качестве первого аргумента вы можете передавать матрицу, однако генерировать полином можно будет только с флагом 'r'. Например так

```
-->A=[1 2 3; 4 5 6]
 A  =
   1.    2.    3.
   4.    5.    6.

-->poly(A,'x','r')
 ans  =
                2     3     4     5   6
   720 - 1764x + 1624x - 735x + 175x - 21x + x
```

Вы также легко можете вычислить значение полинома от любого значения аргумента с помощью функции **horner()**[1]. Для этого функции необходимо передать два аргумента: имя полинома и массив значений аргумента. Например, для полинома, использовавшегося в начале упражнения,

```
-->horner(p,8)
 ans  =
   150.
-->horner(p,[2.6 5 4])
 ans  =
   13.92    60.    38.
```

Объект rational[править]

Еще одним объектом, который является скорее производным от объекта polinomial, является rational. Объект rational (от «рациональное число») образуется делением одного полинома на другой, при этом этот объект имеет свой внутренний идентификатор и инициируется именно как rational.

```
-->r=poly([1 2 3],'x','c')/poly([4 5 6],'x','c')
 r  =

          2
   1 + 2x + 3x
   -----------
          2
```

```
   4 + 5x + 6x

-->typeof(r)
 ans  =
 rational
```

Объект rational полностью наследует все функции полинома, например, для него также применима функция **horner()**.

```
-->horner(r,5)
 ans  =
   0.4804469
```

Если углубиться во внутреннее устройство, то вы можете увидеть, что и rational, и полином являются типизованными списками. Объявить объект rational можно также через специальную функцию **rlist**, которая на самом деле является макросом, использующим **tlist()**.

```
//rlist() ожидает в качестве аргументов числитель и знаменатель
-->rlist(5*s^2+s,2*s*s)
 ans  =
         2
    s + 5s
    ------
       2
     2s
```

Объекты rational могут использоваться в теории систем, для объявления передаточных функций.

Математическая функция[править]

В Scilab есть предопределенные математические функции, такие как тригонометрические, экспонента и другие. Но что, если вам необходимо определить собственную функцию? Далее мы будем отличать математические функции и программируемые функции. Разница между ними заключается в том, что программируемые функции призваны реализовывать некоторый алгоритм, в то время как математическая функция отражает связь между множеством аргументов и множеством значений функции.

Отметим, что математическую функцию можно объявить через программирование, но так как обычно ее тело состоит из одной строчки, то рациональнее всего объявлять ее через специальную функцию **deff()**. Общий синтаксис имеет вид

```
deff('[Y1,Y2...]=Fname(X1,X2,...)',['Y1=выражение_1';'Y2=выражение_2;...'])
// [Y1,Y2...] — вектор возвращаемых переменных (имена назначаете сами)
// Fname — имя функции, которое вы назначаете сами
// (X1,X2,...) — список из аргументов функции
// 'Y1=выражение_1;Y2=выражение_2;...' — для каждой выходной переменной должно быть определено выражение,
// которое может зависеть от аргументов, а может и не зависеть.
```

Данная конструкция поначалу кажется сложной и неизбежны ошибки при ее вводе. Главное не забывать, что аргументы для функции deff() вы передаете в виде строк, которые затем разбиваются на лексемы и из которых среда строит программируемую функцию.

Упражнение

Объявим квадратичную функцию следующим образом

```
-->deff('y=f(x)','y=x^2')
```

Убедимся в том, что мы объявили функцию

```
-->whos -name 'f'
Name                Type          Size          Bytes

f                   function                    192
fft                 fptr                    40
fileiolib           library                     624
functionslib        library                     752
```

Отметим, что наша функция хранится в памяти в компилированном или бинарном виде, так как по умочанию был применен флаг 'c'. Компилированные функции работают быстрее, однако после компиляции они привязываются к операционной системе, в которой работает Scilab. В большинстве случаев вам не нужно задумываться над тем «компилировать функцию или нет?». О компиляции функций мы поговорим, когда будем учиться писать библиотеки функций, а сейчас попробуем вызвать функцию

```
-->f(25)
 ans  =
   625.
```

Очевидно, что функция работает правильно. Отметим, что deff() еще и интегрирует функцию в среду, в связи с чем вы можете передавать в качестве аргументов векторы. Например так

```
-->f([2 3 4 5])
 ans  =
   4.    9.    16.    25.
```

Усложним функцию следующим образом

```
-->deff('[x y]=f(z,u)',['x=z^2-u';'y=u^3'])
// Примечание: в версии для Linux может возникнуть ошибка 37. Чтобы ее присечь, необходимо обязательно разделять элементы вектора запятой, т.е. [x,y].
```

Созданная функция переопределится, о чем вам скажет предупреждающее сообщение. Если вы не хотите постоянно видеть это предупреждение, то воспользуйтесь функцией **funcprot(0)**.

Этой строкой мы создали сложную функцию, которая принимает уже два обязательных аргумента и возвращает вектор результатов. Функция сложная потому, что ее тело состоит уже из двух строк. Обратите внимание, что тело сложной функции должно передаваться **deff()** вектором-столбцом, каждая строка которого является инструкцией, записываемой строковым типом данных.

Вызовем нашу функцию

```
-->f(2,2)
 ans  =
   2.
```

Очевидно, что мы получили не то, что хотели, ведь функция должна была вернуть вектор. К сожалению, в данном случае *ans* не переопределилась автоматически системой в вектор и был записан только результат расчета переменной x, потому что при объявлении в **deff()** мы ее записали первой. Чтобы получить ответ целиком, мы должны слева от вызова явно показать вектор. В этом случае мы должны объявить некоторые переменные, в которые будет записываться результат, либо воспользоваться переменной *ans* вот так

```
-->[ans ans]=f(2,2)
 ans  =
    2.
 ans  =
    8.
```

В этом случае в *ans* сначала запишется результат первой инструкции, а затем результат второй инструкции. Такой способ записи результата приемлем, если мы хотим просто посмотреть результат расчета, но сам результат не планируем никак использовать. Запишем результат расчета в вектор следующим образом

```
-->[a(1),a(2)]=f(2,2);
-->a
 a  =
    2.
    8.
```

Все аргументы нашей сложной функции являются обязательными и нельзя вызвать функцию, не передав их все. Например, попробуйте сделать такой вызов

```
-->f(3)
 !--error 4
Неизвестная переменная: u
at line       2 of function f called by :
f(3)
```

В дальнейшем мы научимся обходить это ограничение, перегружая функцию.

Обратите внимание на функцию **funcprot()** (от англ. *function protection*). С помощью **funcprot()** вы можете защитить ваши функции от случайного их переопределения. Функция принимает один целочисленный аргумент:

- 0 — отключение механизма защиты;
- 1 — формирование предупреждения (используется по умолчанию);
- 2 — формирует ошибку при попытке переопределения существующей функции.

Вообще говоря, при определении функции через **deff()** она может иметь сколь угодно много инструкций, которые необходимо записывать в вектор, как это было показано в упражнении. Однако, запись становится неудобочитаемой и в этом случае целесообразнее использовать уже вторую конструкцию **function…endfunction**, а также пользоваться редактором *scinotes*, о котором мы будем говорить дальше. Общий синтаксис выглядит так

```
function <выходные переменные>=имя_функции(аргументы)
...
инструкции
...
endfunction
```

Данная конструкция, как правило, используется для написания пользовательских программируемых функций, однако, в зависимости от предпочтений пользователя, она может использовать и вместо функции **deff()**. Использовать эту конструкцию можно в интерпретаторе, который воспринимает ее по особому. Сначала необходимо ввести первую строку с ключевым словом *function*, именем функции, аргументами и выходными переменными. Далее вы можете нажать <Enter> и интерпретатор будет воспринимать все, что вы вводите, как инструкции для данной функции до тех пор, пока вы не введете ключевое слово *endfunction*. Если вы не допустили синтаксических ошибок в

теле функции, то она будет готова к использованию, в противном случае интерпретатор выведет вам ошибку и объявлять придется заново. Именно поэтому при написании больших функции следует пользоваться исключительно *scinotes*.

В рамках данного раздела покажем как можно объявить функцию f через конструкцию **function…endfunction**.

```
-->function x=f(z)
-->x=z^2
-->endfunction

-->f(4)
 ans  =
    16.
```

И еще структуры

В Scilab реализовано два особых типа структурных данных, которые созданы скорее для переносимости ваших сценариев из одной среды в другую. Если вас не заботит переносимость ваших будущих кодов, то вы вообще ими можете не пользоваться, так как в Scilab все замечательно работает и со списками.

Первый особый тип данных это **Struct** (структура), который был введен для переносимости сценариев между Scilab и двумя другими математическими пакетами MATLAB и Octave. Внешне Struct очень похож на типизованный список. В отличие от типизованного списка, который хранит имя в себе, именем структуры является переменная, в которую вы эту структуру записываете. В остальном структура похожа на обычный список, в котором есть имена полей и значения им присвоенные. Например,

```
-->s=struct("firstname","Валерий","Age",23)
 s  =
   firstname: "Валерий"
   Age: 23
-->s.firstname
 ans  =
 Валерий
```

При объявлении структуры на нечетным позициях в аргументах всегда стоят имена полей, а на четных — присваиваемые значения. В отличии от списков Scilab, у структур сразу удобочитаемый для пользователя формат вывода, когда как в Scilab формат нужно назначать перегрузкой.

Неудобства структуры проявляют себя при программировании. Новые поля создаются за один ход

```
-->s.address = 'пр. Ленина 55-6'
 s  =
   firstname: "Валерий"
   Age: 23
   address: "пр. Ленина 55-6"
```

Однако, этот факт заставляет вас постоянно держать внимание на именах. Вот такая типичная ошибка может произойти при обращении к нашей структуре

```
-->s.age=[]                 // забыв, что имя поля Age пишется с прописной буквы, мы создали еще одно поле
 s  =
   firstname: "Валерий"
   Age: 23
   address: "пр. Ленина 55-6"
```

```
  age: [0x0 constant]
-->s.age=null()              // однако, в Scilab структуры реализуются mlist-списками, поэтому мы можем применять null()
 s  =
  firstname: "Валерий"
  Age: 23
  address: "пр. Ленина 55-6"
```

Также в структурах нельзя обращаться к полям по индексам. Если структуры хранят в себе данные, например работников, и работников несколько, то единственный способ их как-то объединить это записать их в вектор. В Scilab эту же операцию удобно выполнять с помощью mlist-списков, которые могут быть перегружены вашими собственными операциями.

Другой тип структурных данных это **cell** (ячейка). Этот тип данных частично совместим с MATLAB и Octave. Cell сочетает в себе особенности списка и матрицы. Как вы помните, в матрицах можно хранить данные только одного типа данных, но в cell это ограничение не действует. Таким образом cell — это матрица, которая способна хранить данные разных типов.

Чтобы создать cell нужно вызвать одноименную функцию

```
-->c = cell(2,3)          // мы создаем cell-матрицу 2x3
 c  =
!{}  {}  {}  !
!          !
!{}  {}  {}  !
```

В отличие от массивов, к элементу cell нельзя обратиться по индексам для заполнения, как мы это знаем. Чтобы заполнить позицию в cell-матрице, нужно воспользоваться специальным служебным словом

```
-->c(1,3).entries = 'Сюда запишем строку';..
-->c(2,2).entries = 12
 c  =
!{}  {}  "Сюда запишем строку"  !
!                                !
!{}  12  {}                      !
```

Индексы в cell сами по себе используются исключительно для извлечения по всем правилам как в матрицах

```
-->c(2,2)
 ans  =
 12
-->c(1,:)                    // результатом выделения является также cell-матрица
 ans  =
!{}  {}  "Сюда запишем строку"  !
```

Из cell можно создавать гиперматрицы, как это было с помощью функции **hypermat()**, но, в отличие от простой гиперматрицы, cell-гиперматрицы будут способны хранить данные разных типов.

```
-->c = cell(2,2,2)
 c  =
(:,:,1)

!{}  {}  !
!        !
!{}  {}  !
```

```
(:,:,2)

!{}  {}  !
!      !
!{}  {}  !
```

Таким образом, cell удобно использовать, когда вы «меньшей кровью» хотите получить объект, обладающий свойствами матриц и list-списков.

Если вы предполагаете более сложные действия над собственными объектами и не предполагаете переносимость, лучше всего все же пользоваться типизованными списками Scilab.