

Язык aForth.

Описание и принципы программирования.

Назначение: Язык aForth является Forth-подобным языком общего назначения. В основу языка положены следующие принципы:

- Лёгкая расширяемость языка путём написания новых слов и создания словарей как в виде модулей, встраиваемых в язык, так и средствами самого языка.
- Принцип укомпиляция на лету. При определении нового слова, оно сразу преобразуется в код виртуальной aForth-машины.
- Хранение в виде текста. Все программы на aForth хранятся в виде текстовых (или скомпрессованных текстовых) файлов. Компиляция программ производится при загрузке.
- Самопрограммирование. Наличие средств языка, позволяющих определять новые слова, синтезированные в процессе выполнения программы.

Базовые понятия языка: Основные понятия языка aForth - стек, слова и переменные.

Стек - стековая структура, хранящая операнды любого типа, за исключением типа-массива байт. Стек имеет ограниченный размер (задаётся при сборке aForth). Значения снимаются со стека в порядке, обратном тому, в котором они были туда положены - по принципу последний пришёл-первый ушёл.

Слова - операторы, производящие действия над операндами, находящимися на стеке. Все слова в aForth-программе отделяются от других пробелами.

Строка - это единое слово, заключённое в кавычки. В состав строки так же могут входить специальные символы: \n - перевод строки, \t - табуляция, \\ - слэш и \` - кавычки. Все слова, имеющиеся в aForth-системе - входят в её словарь.

Символ - это одиночный символ, заключённый в апострофы. Например 'A', '0', '\n'. Имеются специальные символы (начинаются со слэша): \n - перевод строки, \t - табуляция, \\ - слэш и \` - кавычки. Во всех операциях символ полностью эквивалентен целому числу - коду данного символа.

Переменные - зарезервированные именованные участки памяти для хранения данных заданного типа. Переменные являются видом слов и входят в словарь.

Типы данных: Язык aForth поддерживает следующие типы данных: целые числа, числа с плавающей точкой, строки. Кроме того, ограниченно поддерживаются байты и короткие целые. При всех операциях над байтами и короткими целыми они преобразуются в целые.

Для строк определены операции сравнения (равно, больше, меньше) и сложения. Другие операции над строками стандартным словарём не предусмотрены.

Принципы написания программ.

Программа на aForth представляет собой текстовый файл, содержащий (в общем случае) комментарии, определения переменных, определения новых слов, интерпретируемые слова. Если файл вводится с клавиатуры - получаем автоматически интерактивную aForth-систему.

Рассмотрим пример aForth-программы:

```
(  
Программа, печатающая "Hello world".  
( Это вложенный комментарий )  
Программа определяет новое слово - hello_world
```

```
)
: hello_world "Hello world\n" . ;
hello_world
```

Рассмотрим построчно программу, печатающую классическую фразу `␣Hello world␣`.

Сначала программы - комментарий. Комментарий начинается со слова `(` - открывающая скобка и завершается словом `)` - закрывающая скобка. Всё что находится между скобками - является комментарием и никак не влияет на работу программы. Комментарии могут быть вложенными, что удобно для закоментирования части текста во время отладки.

Строка, начинающаяся со слова `:` (двоеточие) - определение нового слова. Встретив двоеточие, aForth-система переходит в режим компиляции. Слово `hello_world`, стоящее после двоеточия - считается названием нового слова, остальные слова - его телом. Тело нового слова может быть пустым. В случае пустого тела, определение слова выглядит так (определяем ничего не делающее слово, назвав его `empty-word`):

```
: empty-word ;
```

Тело нового слова заканчивается словом `;` (точка с запятой). Слово `;` переводит aForth-систему в режим интерпретации вводимых слов и помещает вновь определённое слово в словарь. В общем виде определение нового слова выглядит так:

```
: <новое_слово> <слово1> <слово2> ... <слово N> ;
```

В примере программы, печатающей `␣Hello world␣` телом слова нового слова `hello_world` является строка `"Hello world\n"` и оператор вывода вершины стека `.` (точка).

В момент определения нового слова (его компиляции) оно преобразуется в код виртуальной машины aForth и помещается в словарь.

Для того, чтобы выполнить действия, которые определены в слове - достаточно написать это слово. Это и сделано в последней строке программы - встретив слово `hello_world`, программа находит это слово в словаре и выполняет его код.

aForth-система может находиться в одном из двух режимов - компиляции (определения новых слов) и интерпретации (непосредственного исполнения встреченных слов).

После запуска система находится в режиме интерпретации. В режим компиляции система переходит в начале определения нового слова (оператор `:`) и возвращается в режим интерпретации по завершении определения нового слова (оператор `;`).

Нашу программу, печатающую `␣Hello world␣` можно написать и без определения нового слова, например так:

```
(
Программа, печатающая "Hello world".
Программа работает полностью в режиме интерпретации и не определяет новых слов.
)
"Hello world\n" .
```

Операторы ветвления и циклов можно использовать ТОЛЬКО в режиме компиляции (то есть при определении новых слов).

Общий порядок работы aForth-системы - следующий:

- Если встречается комментарий - он игнорируется и все, вложенные в него, комментарии - тоже.
- Если встречается слово, которое удалось распознать как целую, вещественную, строковую константу или переменную - оно помещается на вершину стека. Количество элементов на стеке при этом увеличивается на 1.
- Если встречается слово, которое есть в словаре - то выполняется его код (это может быть как машинный код - для встроенных слов, так и код виртуальной машины - для слов, определённых пользователем).
- Если слово не удалось распознать - система генерирует ошибку. В простейшем случае - выдаёт сообщение и прекращает выполнение программы.

Обратная польская запись.

Все выражения системы aForth записываются в форме обратной польской записи. Эта форма записи не содержит скобок и предназначена именно для записи выражений в концепции стека. Примеры:

2 2 + аналогично 2+2

3 2 + 4 * аналогично (3+2)*4

Выражения вычисляются слева направо без приоритетов. Это позволяет сделать весьма простую машину для их интерпретации.

Форма обратной польской записи в системе aForth относится к любым выражениям. Любая операция снимает со стека необходимое ей количество операндов, и ложит результат (при его наличии) на стек.

Например - операция сложения - бинарная. То есть требует двух операндов и должит на стек результат - один операнд. Рассмотрим по шагам действия системы aForth при выполнении операции сложения, а именно вычислении выражения 2+3 и вывода результата на экран. На языке aForth действие выглядит так:

2 3 + .

Теперь по шагам:

(Изначально считаем, что стек пуст)

2 (Помещаем на стек первый операнд)

3 (Помещаем на стек второй операнд)

+ (Слово + снимает со стека два операнда, складывает их и помещает на стек результат - число 5)

. (Снимает со стека результат - число 5 и выводит его на экран)

(После выполнения всех действий стек вновь пуст)

Слова вывода информации.

Для вывода информации в поток стандартного вывода (обычно - на дисплей) существуют стандартные слова:

. (точка) - Выводит значение, находящееся на вершине стека, целые числа печатаются в десятичном виде.

.x - Выводит значение, находящееся на вершине стека, целые числа печатаются в шестнадцатеричном виде строчными буквами.

.X - Выводит значение, находящееся на вершине стека, целые числа печатаются в шестнадцатеричном виде прописными буквами.

.C - Выводит символ, код которого находится на вершине стека в виде целого числа.

В примерах выше использовано множество операторов вывода для вывода строк и чисел. Все операторы вывода информации снимают со стека его вершину. Поэтому, если вершина нужна для дальнейшего использования - её надо продублировать словом dup.

Константы.

Константы бывают трёх типов - целые, вещественные, строковые. При распознавании встреченного слова как константы, aForth-система в режиме интерпретации сразу помещает её на стек. В режиме компиляции генерируется код, помещающий константу на стек, но реальное помещение константы на стек происходит только во время выполнения скомпилированного слова.

Целые константы. Целые константы записываются в десятичном или шестнадцатеричном виде. В шестнадцатеричном виде запись целой константы начинается с символов 0x или 0X. Примеры:

10 20 12 - целые константы в десятичном виде.

0x0A 0x14 0x0C - целые константы в шестнадцатеричном виде.

Вещественные константы. Вещественные константы отличаются от целых, что в их записи присутствует десятичная точка. Вещественные константы могут записываться только в десятичном виде. Примеры:

1.0 2.3 5.7 - вещественные константы.

Строковые константы. Строковые константы - строки символов, заключённые в кавычки. В состав строки так же могут входить специальные символы: \n - перевод строки, \t - табуляция, \\ - слэш и \' - кавычки. Пример:

Мамау \nпу \ty

Пример программы, работающей с разными типами констант:

```
( Целые константы )
"Работа с целыми константами: " .
1 . " " . 2 . "\n" . "1 + 2 = " .
1 2 + . "\n" .
( Вещественные константы )
"Работа с вещественными константами: " .
1.0 . " " . 2.0 . "\n" . "1.0 + 2.0 = " .
1.0 2.0 + . "\n" .
( Строковые константы )
"Работа со строковыми константами: " .
"Ёжик в " . " тумане" . "\n" . "\"Ёжик в\" + \" тумане\" = " .
"Ёжик в " . " тумане" + . "\n" .
```

Переменные.

Переменные могут быть тех же типов, что и константы. Определение переменной в общем случае выглядит так:

<Константа-инициализатор> variable <слово-имя переменной>

Переменная получает тот же тип, что и <Константа-инициализатор>. Имя переменной не должно совпадать со словами, имеющимися в словаре.

Пример:

```
( Создаём целую переменную )
0 variable INTVAR
( Создаём вещественную переменную )
0.0 variable REALVAR
( Создаём строковую переменную )
"Ёжик " variable STRVAR

( Печатаем их значения )
"Целая переменная INTVAR = " . INTVAR . "\n" .
"INTVAR = INTVAR + 1.0 = " .
INTVAR 1.0 + INTVAR !! INTVAR . "\n" .

"Вещественная переменная REALVAR = " . REALVAR . "\n" .
"REALVAR = REALVAR + 1 = " .
REALVAR 1 + REALVAR !! REALVAR . "\n" .

"Строковая переменная STRVAR = " . STRVAR . "\n" .
"STRVAR = STRVAR + \"в тумане\" = " .
STRVAR "в тумане" + STRVAR !! STRVAR . "\n" .
```

Для помещения значения переменной на вершину стека - достаточно написать её имя. Для помещения значения с вершины стека в переменную служит оператор !! (два восклицательных знака). Для него необходимы как минимум два значения на стеке - значение, помещаемое в переменную и (на вершине стека) - сама переменная. Пример:

```
0 variable INTVAR
5 INTVAR !!
```

Поместит в переменную INTVAR значение 5. Значение, помещаемое в переменную приводится К ТИПУ ПЕРЕМЕННОЙ - то есть если вещественное значение помещаем в целую переменную, оно усекается до целого. Если целое значение - в вещественную переменную, то значение преобразуется в вещественную форму. Строковые значения могут помещаться только в строковые переменные.

Массивы байт.

Существует особый тип переменных - массив байт. Этот тип переменной предназначен для хранения произвольной последовательности байт произвольной длины. Определяются массивы байт следующим образом.

```
<длина в байтах> array <имя массива>
```

Например:

```
10 array ten_bytes
0 array empty_array
```

Длина массива при определении может быть равна 0. Тогда это пустой массив, длина которого может быть изменена до нужного значения словом `aresize`. Пример программы, демонстрирующей эти действия:

```
10 array A
"Длина массива A = " . A size . " байт\n" .
"Адрес массива A = " . A & . "\n" .

"Изменяем размер массива A на 20\n" .
20 A aresize

"Длина массива A = " . A size . " байт\n" .
"Адрес массива A = " . A & . "\n" .

"Заполняем массив A цифрами\n" .
1 A & 0 + setmchar ( Слово setmchar помещает байт по адресу: <байт> <адрес> setmchar )
2 A & 1 + setmchar
3 A & 2 + setmchar
4 A & 3 + setmchar
5 A & 4 + setmchar
6 A & 5 + setmchar
7 A & 6 + setmchar
8 A & 7 + setmchar
9 A & 8 + setmchar
0 A & 9 + setmchar

"Читаем цифры из массива\n" .
A & 0 + getmchar . "\n" . ( Слово getmchar помещает на вершину стека байт по адресу: <адрес> getmchar )
A & 1 + getmchar . "\n" .
A & 2 + getmchar . "\n" .
A & 3 + getmchar . "\n" .
A & 4 + getmchar . "\n" .
A & 5 + getmchar . "\n" .
A & 6 + getmchar . "\n" .
A & 7 + getmchar . "\n" .
A & 8 + getmchar . "\n" .
A & 9 + getmchar . "\n" .
```

Массивы можно определить так же через константные массивы. Константа-массив - последовательность слов, заключённая в слова `{ }` (фигурные скобки). В определение массива могут входить:

- Константы любого типа.
- Переменные любого типа.
- Массивы-переменные.

Пример определения массива A через константу-массив:

```
10 variable I
12.3 variable R
ȳСтрокаȳ variable S
{
    1 4.0 ȳЁжиȳ ( Константы )
    I R S ( Значения соответствующих переменных )
} array A
{ A } array B ( Создает массив B, копию массива A )
{ ȳНовое значение массива Aȳ } A !! ( Создает массив и заменяет содержимое массива A на него )
0 array C ( Пустой массив 0й длины )
{ } array D ( Пустой массив 0й длины - полностью аналогично строке выше.)
```

Строки в массиве сохраняются как последовательность байт с завершающим символом '\0'. Вещественные сохраняются как последовательность байт, количество которых соответствует типу real. Целые по умолчанию сохраняются как целые, но с помощью слов-модификаторов могут сохраняться как короткие целые или байты. Например:

```
{
    byte 1 2 3 4 ( Модификатор byte - целые занимают по одному байту )
    short 1 2 3 4 ( Модификатор short - целые сохраняются как короткие целые )
    int 1 2 3 4 ( Модификатор byte - целые сохраняются как целые )
}
```

Модификаторы массивов byte, short и int могут стоять в любом месте определения массива-константы. Область их действия распространяется до конца определения массива-константы.

Определение массивов-констант работает ТОЛЬКО в режиме интерпретации и не может использоваться при определении новых слов. Однако, слово doword (см. ниже) позволяет синтезировать содержимое массива во время выполнения из других переменных и массивов:

```
(
    С константами-массивами можно работать только в режиме интерпретации. Но - голь на выдумки хитра
    "dict.af" include ( Пустой массив A )
    { } array A "Пустой массив. A=" . A size . CR
    ( Переменные )
    0x10 variable I1
    0x20 variable I2
    0x30 variable I3
    ( mode - хранит режим синтеза массива - byte, short или int )
    "" variable mode
    ( Слово array-syntes - синтезирует строку "{ <mode> I1 I2 I3 }" и выполняет её в режиме интерпретации
    : array-syntes "{ " mode + " I1 I2 I3 }" + doword swap !! ;
    "Синтезируем массив байт" . CR
    "byte" mode !! A array-syntes A VARDUMP CR
    "Синтезируем массив коротких целых" . CR
    "short" mode !! A array-syntes A VARDUMP CR
    "Синтезируем массив целых" . CR
    "int" mode !! A array-syntes A VARDUMP CR
```

Операторы ветвления.

В языке предусмотрен оператор ветвления if then else. Операторы ветвления НЕ МОГУТ ИСПОЛЬЗОВАТЬСЯ в режиме интерпретации. То есть они могут применяться только в режиме компиляции при определении новых слов. Существуют две формы оператора ветвления - краткая и полная:

Полная форма оператора ветвления:

```
<условие>
if
<Слова, выполняемые, если условие не равно 0 (истина)>
else
<Слова, выполняемые, если условие равно 0 (ложно)>
endif
```

Краткая форма оператора ветвления:

```
<условие>
if
<Слова, выполняемые, если условие не равно 0 (истина)>
endif
```

Пример программы, использующей слова с оператором ветвления приведён ниже. В программе определяются три слова: is-positive, is-negative и is-zero, которые проверяют вершину стека на положительность, отрицательность или нулевое значение и выводят соответствующие сообщения.

```
( Пример использования оператора ветвления )
( Проверка вершины стека на то, что она положительна )
: is-positive
0 > ( Проверка на то, что вершина стека больше 0 )
if
"В ершина стека положительна\n" .
endif
;

( Проверка вершины стека на то, что она отрицательна )
: is-negative
0 < ( Проверка на то, что вершина стека меньше 0 )
if "В ершина стека отрицательна\n" .
endif ;

( Проверка вершины стека на то, что она ноль )
: is-zero
0 = ( Проверка на то, что вершина стека равна 0 )
if "В ершина стека - ноль\n" .
endif ;

"10 - 5 = " . 10 5 - . "\n" .
10 5 - is-positive
10 5 - is-negative
10 5 - is-zero
"5 - 10 = " . 5 10 - . "\n" .
5 10 - is-positive
5 10 - is-negative
5 10 - is-zero
"5 - 5 = " . 5 5 - . "\n" .
5 5 - is-positive
5 5 - is-negative
5 5 - is-zero
```

Другой вариант той же программы, где все проверки сведены к одному слову `signum`:

```
( Пример использования оператора ветвления )
: signum
dup ( Дублируем значение на вершине стека с помощью оператора dup. )
0 = ( Проверка на то, что вершина стека равна 0 )
if
  "Вершина стека - ноль\n" .
drop ( Удаляем лишнее значение с вершины стека )
else
  0 < ( Проверка на то, что вершина стека меньше 0 )
  if
    "Вершина стека отрицательна\n" .
  else
    "Вершина стека положительна\n" .
  endif
endif
;

"10 - 5 = " . 10 5 - . "\n" . 10 5 - signum
"5 - 10 = " . 5 10 - . "\n" . 5 10 - signum
"5 - 5 = " . 5 5 - . "\n" . 5 5 - signum
```

Циклы.

В системе aForth существует два оператора циклов - с предусловием и с постусловием. Операторы циклов НЕ МОГУТ ИСПОЛЬЗОВАТЬСЯ в режиме интерпретации. То есть они могут применяться только в режиме компиляции при определении новых слов.

Цикл с предусловием может не выполняться ни разу. Общая форма цикла с предусловием:

```
begin <условие>
while
  <тело цикла, выполняется если условие - не 0 (истина)>
repeat
```

Цикл с постусловием выполняется хотябы один раз всегда. Общая форма цикла с постусловием:

```
begin
  <Тело цикла, выполняется хотябы один раз>
  <Условие>
until
```

Если <условие> не равно 0 (истина), то происходит выход из цикла с постусловием, если же <условие> равно 0 (ложь), то тело цикла выполняется вновь. Пример программы, которая двумя способами печатает все цифры от 0 до 9:

```
( Цикл с предусловием )
( Распечатка чисел от 0 до 9 )
: loop-begin-while-repeat
0 ( Начальное значение счётчика - на стек )
begin ( Начало цикла )
dup ( Дублируем значение на вершины )
10 < ( Проверяем его на то, что оно меньше 10 )
while
dup . "\n" . ( Печать вершины стека )
1 + ( Увеличение счётчика на 1 )
```



```

repeat ( Повтор, начиная с begin )
drop ( Удаляем со стека значение-счётчик )
;

( Цикл с постусловием )
( Распечатка чисел от 0 до 9 )
: loop-begin-until
0 ( Начальное значение счётчика - на стек )
begin
dup . "\n" . ( Печать вершины стека )
1 + ( Увеличение счётчика на 1 )
dup ( Дублируем значение на вершины )
10 = ( Проверяем его на равенство 10 )
until
drop ( Удаляем со стека значение )
;

"Распечатка чисел от 0 до 9 begin-while-repeat\n" . loop-begin-while-repeat
"Распечатка чисел от 0 до 9 begin-until\n" . loop-begin-until

```

Работа со стеком.

Для работы со стеком используются следующие слова: dup, drop, over, pover, swap.

- dup - Дублирует содержимое вершины стека. При этом кол-во элементов увеличивается на 1. Пожалуй самая часто используемая операция со стеком, поскольку все слова, требующие операндов на стеке - снимают с вершины эти операнды. Поэтому перед тем, как проверить значение на вершине стека (например сравнить его с чем-нибудь) или вывести это значение на экран - его надо продублировать. Пример:

```
2 dup . . ( Заносит двойку на стек, дублирует её, и печатает два верхних значения на стеке )
```

- drop - уничтожить значение на вершине стека. Удаляет со стека значение, находящееся на его вершине. При этом кол-во элементов уменьшается на 1. Пример:

```
3 2 drop . ( Напечатает 3. Двойка уничтожена оператором drop )
```

- over - дублирует значение, находящееся перед вершиной стека. При этом кол-во элементов увеличивается на 1. Пример:

```
5 7 over . . . ( Напечатает 575. 5ка продублирована на вершине стека. )
```

- swap - обмен вершины стека и предыдущего элемента стека. Пример:

```
5 7 . . ( Напечатает 75 )
5 7 swap . . ( Напечатает 57 )
```

- pover - Копирование произвольного значения стека. На вершине - номер от начала стека 0 - верхнее значение и т.п. Выражение `y0 pover y` то же, что и слово dup; `y1 pover y` то же, что и слово over.

```
1 0 pover . ( Напечатает 1, то же, что и dup )
3 1 1 pover . ( Напечатает 3, то же, что и over )
5 4 3 2 3 pover ( Напечатает 5 )
```

Выполнение строк как слов aForth. Синтез новых слов.

Имеется возможность выполнить строку (константу или переменную) так, словно она является частью программы на aForth.

Слово `doword` выполняет строку как команду aForth. Пример:

```
y: twoadd 2 2 + . ;y variable cmdvar
cmdvar doword ( Генерируется новое слово twoadd из строки )
twoadd ( Напечатается 4 )
```

Обзор слов стандартного словаря `stddict`.

Подробное описание слов смотреть в документах `aFort-dicts.pdf` или `aFort-dicts.dvi`. Данные документы как и данный обзор будут изменяться и дополняться.

В стандартный словарь входят следующие группы слов (некоторые из слов уже были рассмотрены выше):

- Арифметические операции - сложение (+), вычитание(-), умножение(*), деление(/). Все арифметические операции требуют двух операндов на стеке. Операция сложения работает как с числами так и со строками. Кроме того, складывать можно строки и символы (то есть целые числа). При сложении символа и строки - символ добавляется в начало строки. При сложении строки и символа - символ добавляется в конец строки:

```
yСтрока y 'A' + . y\ny . ( Напечатает строку yСтрока Ay )
yСтрока y 65 + . y\ny . ( Напечатает строку yСтрока Ay )
'A' yСтрока y + . y\ny . ( Напечатает строку yA Строкаy )
65 yСтрока y + . y\ny . ( Напечатает строку yA Строкаy )
```

- Операции сравнения - равенство (=), больше (>), меньше (<). Так же снимают со стека два операнда. Эти операции сравнивают два числа или две строки и выдают результат сравнения на стек в виде 0 (ложь) или 1 (истина).
- Операции управления программой - начало определения нового слова (:), конец определения нового слова (;), включение файла (include). Операции определения новых слов уже описаны выше. Слово включения файла (include) позволяет подключить другой файл к файлу программы, как если бы он был частью данного файла. Это слово позволяет разбить большой файл программы на меньше по размеру. Слово include используется в формате <имя_файла> include. Поскольку <имя_файла> - строка, то может использоваться любая строковая переменная или константа.
- Операции сохранения словаря aForth-системы - слово (save). Сохраняет все слова - определение слов, текущие значения переменных и массивов. Не сохраняет состояния стеков!
- Операции управления потоком команд - операторы ветвления и циклы. Они уже описаны выше. Это операторы if else endif; begin while repeat until.
- Операторы вывода - . .C .x .X. Так же были описаны выше.
- Операторы работы со стеком - dup, drop, over, pover, swap. Так же были описаны выше.
- Операторы получения размеров различных типов данных и переменных. Слова - intsize, realsize, shortsize и size.

`intsize` - поместить на стек размер целого (в байтах).

`realsize` - поместить на стек размер вещественного (в байтах).

`shortsize` - поместить на стек размер короткого целого (в байтах).

`size` - поместить на стек размер (в байтах) переменной или массива, находящегося на вершине стека.

`aresize` - изменить размер массива (array) на заданный.

- Операторы работы с памятью, а именно слова:

`malloc` - выделить блок памяти.
`free` - освободить блок памяти.
`realloc` - изменить размер блока памяти.
`setmchar`, `setmint`, `setmshort`, `setmstr` - поместить в память по заданному адресу значение соответствующего типа.
`getmchar`, `getmint`, `getmshort`, `getmstr` - поместить на стек значение соответствующего типа, взятое по адресу.
`&` - получить адрес переменной или массива, находящегося на вершине стека.

- Операции работы с массивами и переменными. Слова - `array`, `variable`, !! Эти слова уже описаны выше.
- Слова, показывающие информацию о версии aForth и времени её сборки - `version`, `subversion`, `mktime`.

`version` - Поместить на стек версию aForth - строку, содержащую несколько чисел, разделённых точками.
`subversion` - Поместить на стек подверсию версии aForth - строку.
`mktime` - Поместить на стек время сборки - строку в формате дд.мм.гггг-чч.мм.сс

- Слова для работы с файлами.

`open` - Открыть файл.
`creat` - создать или усечь файл.
`close` - Закрыть файл.
`getchar` - Считать символ из файла.
`putchar` - Записать символ в файл.