Pygmy Forth User Manual

Frank Sergeant

Pygmy Forth

User Manual

Frank Sergeant

Editor in Stew Pot comic, by artist Frank Heitzman, used with permission

formatted October 29, 2017 at 04:15:42 PM

eBooks, PDF, and HTML versions formatted with the Nepo Press web-based service at http://nepotism.net.

Contents

Introduction

Pygmy Forth is the new, 32-bit or 64-bit Pygmy Forth for PCs/desktop computers. It was first released in October, 2017.

This manual corresponds to Pygmy Forth version 17.10, released October, 2017. The first two digits of the version number represent the year (17). The next two digits represent the month (10). If more than one release occurs in a month, the version will be adjusted slightly.

I now refer to the older 16-bit Pygmy Forth as Pygmy for DOS or Pygmy Forth for DOS.

I miss my Pygmy for DOS. This new Pygmy Forth is my attempt to replace it for the modern desktop environment (primarily Linux in my case).

For programming for PCs, I used to love my lovely Pygmy for DOS. However, the days of DOS are gone, and good riddance to Microsoft as far as I'm concerned. I am delighted with the speed and conveniences and openness of the Linux world, but I have been *missing* Forth in that environment. I know there are many alternative Forths, and I am sure they are great for those who like them, but none of them suits me the way Pygmy did.

In summary, I WANT MY PYGMY BACK for Linux/Mac/Windows.

Pygmy Forth is aimed at desktop systems (Linux, Unix, Mac OS, Windows). It is written in Python (version 3), so should run anywhere that Python runs. I run it mainly on Linux, so if you find it needs any minor adjustments to run elsewhere, please let me know and I'll add the details to this manual.

Pygmy Forth is distributed as a zip file. See the Quick Start section for how to run it with or without installing it.

Pygmy Forth is written in Python and it can be extended by writing CODE words (*primitives*) in Python, or by writing high-level Forth words.

Pygmy Forth gives you the benefits of Forth factoring, incremental and interactive testing, and simplicity, while providing easy access to the world of Python facilities and libraries.

This project is in its early stages as of fall, 2017.

If you find anything that does not work as expected or as described in the manual, please let me know.

Pygmy Forth inherits heavily from my various prior Forths, which inherited heavily from Chuck Moore's Forths, especially cmFORTH.

License

Pygmy Forth is licensed under the MIT license except as otherwise noted in specific files. See the file `LICENSE.txt` for the full text of the MIT license. Thus you are free to use it for just about any purpose. The only exception at the moment is the file `pygmy-mode.el` (an Emacs mode for editing pseudo block files) which is licensed under the GPLv2 (see the file `gpl-2.0.txt`).

Audience

This manual does not teach Forth, not even Pygmy Forth, from the ground up. Perhaps I can add some of that to the manual later.

Meanwhile, it will be helpful if you already have some familiarity with Forth, especially Pygmy for DOS and/or Riscy Pygness. If not, you might consider downloading Pygmy for DOS (http://pygmy.utoh.org/pygmyforth.html). It has some supplementary material which might help.

The Author

My name is Frank Sergeant. My email address is mailto:frank@pygmy.utoh.org (`frank@pygmy.utoh.org`). I will be pleased to receive comments, corrections, suggestions about the manual and Pygmy Forth. If you find a bug, please send the smallest example that illustrates the problem, as that will greatly increase the chance of a fix.

I wrote Pygmy for DOS, starting in the late 1980s, as a DOS Forth heavily based on (but not as lean as) Chuck Moore's cmFORTH. I produced several variants of Pygmy for microprocessors, including the 68HC11, 68000, Super 8, 8051, etc.

Later, I wrote Riscy Pygness, a version of Pygmy for the ARM and ARM Cortex. When developing on Riscy Pygness, the work is split between the host PC and the target ARM chip, thus requiring only very limited RAM/Flash on the target.

I have done programming work in a number of languages in addition to Forth, including assembly, Smalltalk, Lisp, Python, and Tcl.

A major project these days is doing both custom and automated eBook and printed book formatting at http://nepotism.net.

Quick Start

Note: When command line examples are given, a leading dollar sign ($) represents the terminal's shell or command prompt for an ordinary user, and a leading pound sign (#) represents the terminal's shell or command prompt for root (also called the superuser). **Do not type the leading pound or dollar sign**. Your exact shell prompt will likely look slightly different, e.g., `frank@oak:$` or `C:\>`.

To get to a command line (shell prompt), open a terminal in Linux or MacOS. For Windows, see https://www.lifewire.com/how-to-open-command-prompt-2618089 or do a web search for something like "getting a command prompt (DOS box) on Windows 10".

The examples below assume Linux, Unix, or MacOS. You'll need to adjust the commands and directories slightly for Microsoft Windows. For example, the Linux `cp` command would be the Windows `copy` command (or just use the Windows Explorer file browser).

To run it without "installing" it:

- Verify Python version 3 is installed
- Download Pygmy Forth as a .zip file from http://pygmy.utoh.org/pygmy64.html
- Unzip the .zip file into the directory of your choice
- Run it in a terminal from that directory as either

```
$ python pygmykernel.py
```

or

```
$ python3 pygmykernel.py
```

To "install" it:

- Verify Python version 3 is installed
- Download Pygmy Forth as a .zip file from http://pygmy.utoh.org/pygmy64.html.
- Select or create a directory named `pygmy` or `pygmyforth`. This directory can be a system wide directory, such as a subdirectory under `/usr/local/`, or a directory under your home directory `/`
- Unzip the .zip file into that directory

- Copy or link the files `pygmykernel.py` and `pygmy.fth` and `pygmy.sh` to a directory in your path such as `/usr/local/bin` (perhaps `C:\Program Files` on Windows)

- Edit `pygmy.sh`, if necessary. On Windows, create a similar batch file named `pygmy.bat`

Verify Python Is Installed

Pygmy Forth requires Python version 3 (or later). You can test for this at a command line by typing the following in a terminal (a command prompt):

```
$ python --version
```

If the response is something like this,

```
Python 3.6.2
```

all is good because the first digit is 3. But if you get a response like this

```
Python 2.7.14
```

you are running version 2 and that won't work. Even so, you still might have version 3 available under a slightly different name. Try this:

```
$ python3 --version
```

If you do not have version 3 available, use your system's package manager to install it or go to https://www.python.org/ to download and install it. It is free, and easy to install. My video at https://www.youtube.com/watch?v=CmUhs6-Aa9k might help. Also, you can do a web or YouTube search for "How to Install Python on <my operating system>".

If you get a message like the following when you try running Pygmy Forth,

```
  File "pygmykernel.py", line 425
    print ("%s " % i, end=END)
                           ^
SyntaxError: invalid syntax
```

that is a clue that you are **not** running Python version 3.

Example full installation

Here is how I might install it on a Linux system. Suppose I have downloaded `pygmyforth1710.zip` into my `/Downloads` directory and that my home directory is `/home/frank` (also known as `/`). Remember, the dollar sign represents my ordinary user account (`frank`) prompt and the pound sign represents the superuser account (`root`) prompt. Here is what I would type (with comments below each line):

```
$ python --version
  (uh-oh, on my system,
    python is just version 2)
$ python3 --version
  (oh, good, I get version 3
    with the python3 command)
$ echo $PATH
  (make sure /usr/local/bin
    is in my path)
$ sudo -i
  (temporarily become root,
    alternatively, use the su command)
# cd /usr/local
  (change to the /usr/local directory)
# mkdir pygmy
  (create a pygmy directory
    under /usr/local)
# cp /home/frank/Downloads/pygmyforth1710.zip .
  (copy the zip file to
    the pygmy directory)
# unzip pygmyforth1710.zip
  (unzip the file)
# cd ..
  (move up a level to /usr/local)
# cd bin
  (move down a level to /usr/local/bin)
# ln -s /usr/local/python/pygmykernel.py .
  (link the Pygmy kernel program
    into /usr/local/bin)
# ln -s /usr/local/python/pygmy.fth .
  (link the Pygmy Forth code
    into /usr/local/bin)
# ln -s /usr/local/python/pygmy.sh .
  (link the shell script
    into /usr/local/bin)
# cat pygmy.sh
  (display the contents of the
    shell script to verify it
    is correct. For example,
    you may need to change
    'python3' to 'python')
# exit
```

```
  (exit the root account)
$
  (now I am once again running
    as my regular user account)
```

After that, any user, from any directory, should be able to start Pygmy by running the following in a terminal:

```
$ pygmy.sh
```

Note, if you want the start-up command to be even shorter, you can rename or copy `pygmy.sh` to just `pygmy` (or even `pyg`).

Running Pygmy Forth

You can start Pygmy Forth like this:

```
$ pygmy.sh
```

or, since the `$*` in the shell script passes all command line arguments to `pygmykernel.py`, you can run it like this to load additional files:

```
$ pygmy stars.fth
```

or

```
$ pygmy myutils.fth myapp.fth
```

The shell script starts `pygmykernel.py`, which loads the rest of the Forth system from `pygmy.fth`. The file `pygmy.fth` is written in Forth. You may load additional Forth files as well.

Since `pygmykernel.py` is marked executable (on systems that respect that), if the default Python is version 3 or later, another way to start Pygmy Forth is just by running

```
$ pygmykernel.py
```

Assuming it is in your path, the above runs `pygmykernel.py`.

```
$ pygmykernel.py stars.fth
```

After that, `pygmykernel.py` loads any additional files listed on the command line, such as the file `stars.fth` in the above example. It looks for the additional files in the current directory unless a different path is specified.

Once you have started Pygmy Forth in the terminal, you will get an interactive prompt where you can type Forth commands or load Forth files. It looks like this:

```
Pygmy Forth version 17.10
Welcome to Pygmy Forth
>
```

Exiting from Pygmy Forth

To exit from Pygmy Forth, type

```
BYE
```

Implementation Language

Pygmy Forth is written in Python (Python version 3), so it will run everywhere: Linux, Mac, Windows, 32 bits or 64 bits. (I don't know about cell phones and tablets. Please let me know if you try it.)

High-level Forth words are compiled as Python procedures, rather than as lists of tokens or addresses. CODE words are supported and are written in Python, not assembly language. That is, you can think of Python as being Pygmy Forth's assembly language.

Vocabularies

There are two:

- FORTH containing non-immediate words
- COMPILER containing immediate words

Multitasking

I hope it will be supported, but I have not begun to think about it yet.

I/O Redirection

This needs some more thought.

Definitions

New Forth words are defined conventionally. High-level definitions begin with a colon and end with a semicolon.

When typing interactively, a new word must be defined entirely on one line. So, don't press the Enter key until after typing the ending semicolon. However, there is no limit to line length, so just let it wrap on the terminal as necessary.

Here are some examples:

```
: STAR ( -)   '* EMIT ;
: STARS ( n -)  FOR STAR NEXT ;
: 6STARS ( - )  6 STARS ;
```

Definitions may have multiple exits, with either the word EXIT or the shorthand double semicolon (; ;). The single semicolon is used only to mark the end of a definition.

High-level Forth words are maintained in the dictionary and automatically map the Forth word name to the name of the corresponding Python procedure.

Recursion

Recursion is allowed, but be careful. The word must already be defined, at least with a dummy definition. In the following example, FAC will first be defined with a dummy definition, but that will be replaced by the second definition of FAC. Also, “tail call optimization” is not performed, so you could run out of stack eventually.

```
: FAC DUP ;  ( just a dummy definition)

: FAC ( n - n!)
  DUP 1 >
  IF ( n) DUP 1 - ( n n-1) FAC * THEN ;
```

Kernel

As with a traditional Forth, there is a *kernel*, which is just enough Forth to know how to LOAD the rest of the Forth system from a Forth text file.

The kernel is a Python (Python 3) program named `pygmykernel.py`. Running this in a terminal gives you an interactive loop (a REPL) with a prompt where you can type Forth words and load Forth source code files.

In effect, Python is the assembly language for Pygmy Forth. Thus, "primitives" (CODE words) are defined as Python procedures.

Although there is no need to define small constants (such as 0, 1, 2, 3) as Forth words, here is how they could be defined as CODE words. The new word 0 would push the integer zero to the Forth data stack:

```
CODE 0    dpush(0)   END-CODE
```

See any of the CODE words in `pygmy.fth` for further examples. Note that there must be at least two spaces between the name and the start of the Python code, or the Python code must start on a new line and be indented.

When `pygmykernel.py` begins, it tries to load `pygmy.fth` from the directory that `pygmykernel.py` is in. Failing that, it tries to load `pygmy.fth` from the current directory.

After loading `pygmy.fth`, it attempts to load any additional Forth files passed to it on the command line. For example,

```
$ pygmykernel.py stars.fth
```

The above would load `pygmy.fth` and then load `stars.fth`.

After loading the source files, it runs the interactive loop (the REPL) known in Forth as QUIT. The word QUIT is defined in `pygmy.fth` but you could redefine it either by editing `pygmy.fth` directly (not recommended) or by defining it again in a file that is loaded later. Later definitions replace earlier definitions.

By redefining QUIT, you can run an application (in which case, depending upon your code, perhaps no interactive prompt will appear).

Turnkey Applications

See the previous section about the kernel and how to run Pygmy Forth. Since you can load additional files just by placing them on the command line, you can load your application (let's call it `myapp.fth`) wherein you have redefined `QUIT` to run your application instead of running a REPL.

```
$ pygmy myapp.fth
```

or you can create a shell script named `myapp` or `myapp.sh` (or, on Windows, perhaps an equivalent batch file) that looks something like this:

```
#!/bin/sh
/usr/bin/python3 \
  /usr/local/bin/pygmykernel.py myapp.fth
```

The backslash at the end of the second line indicates the command is continued on the third line. You would ordinarily write lines two and three on a single line (without the backslash).

Of course, you will need to make the shell script executable, e.g.,

```
$ chmod +x myapp
```

If you are running something other than Linux/Unix (or even if you aren't), you might Google for "how to make a turnkey Python app for Windows". One thing that turned up that looks interesting is http://www.pyinstaller.org/index.html.

Hiding/Protecting/Encrypting Your Source Code

Let's not do that.

Numeric Bases

By default, input and output are in decimal but hexadecimal is also recognized.

Traditionally, Forth allowed nearly any arbitrary base, at least up to base 36 (10 digits plus 26 letters of the alphabet).

In microprocessor work, binary, octal, decimal, and hexadecimal might all be useful at times, but for PC/desktop applications, I think decimal and hexadecimal are all that will be needed.

An initial dollar sign indicates a hexadecimal number on input. To display a number in hexadecimal, print it with the word `.H`.

```
$45 EMIT   --> E
65 .H      --> 41
```

Numbers

Pygmy Forth accepts integers and floats and characters as numbers, e.g.,

```
13 .        --> 13
13.75 .     --> 13.75
'E .        --> 69
'E EMIT     --> E
```

Strings and Characters

```
" Hello" .           --> Hello
" Hello" EMIT        --> H
" Hello" COUNT TYPE --> Hello
." Hello"            --> Hello
```

Unlike traditional Forths, in Pygmy Forth, a character is just a string with a length of one. Consider the Forth BL. It is defined like this:

```
CODE BL    dpush(' ')    END-CODE
```

It returns a string containing a space rather than the ASCII code for a space. However, 'A, etc., put a number on the stack (the ASCII code for the character), so there may be some inconsistency here.

Text Files versus Block Files

There are more or less three types of files for Forth source code:

- classic block files
- pseudo block files
- text files

I can see Pygmy Forth supporting all three types, but to begin with, it supports loading just from pseudo block files and from text files.

Pygmy for DOS has a simple, pleasant, block editor. I used it happily for years.

However, these days, most of my life is spent in Emacs. Anything other than Emacs is fairly painful to me. I'm sure others feel the same about Vi or their favorite text editor.

Some people like to keep Forth source code in plain text files and others like to keep it in blocks.

The new Pygmy Forth lets you use either or both, except, the block files supported at this time are really pseudo block files.

Text Files

The new Pygmy Forth can load directly from text files like this:

```
" mysource.fth" LOAD
```

Note that the file doesn't need to be opened or closed explicitly.

Pseudo Block Files

Pygmy Forth can also load from pseudo block files like this

```
" myapp.blk" OPEN
1 LOAD
13 LOAD
" anotherapp.blk" OPEN
300 LOAD
```

A pseudo block file needs to be opened before accessing it. It does not need to be closed explicitly. It remains open until another pseudo block file is opened (or until the program exits).

Pseudo block files are really plain text files with a special comment line that marks the beginning of each block.

This lets you edit text files as if they were block files, but with no requirements on the size of the blocks. They are "logical" blocks rather than physical 1024-byte blocks.

A special comment is used to indicate the start of a block. Here is an example:

```
( block 7)
```

The comment must start at the left margin with an opening parenthesis followed by one or more spaces, followed by either the word "block" or the word "shadow", followed by one or more spaces and then the block number. Eventually, the comment line must end with a closing parenthesis.

Here are some examples:

```
( block 3 miscellaneous)
(  block 7)
(  shadow 7)
( block 375 some description)
```

Shadow blocks, used for documentation, are optional and are ignored by LOAD.

Pygmy-mode and Emacs

Since a pseudo block file is a plain text file, you can use any text editor you wish. However, an editor with outlining capabilities comes in handy so you can see all the block comment lines with the contents hidden, or display just one block at a time and page down and page up to move from block to block.

Pygmy Forth comes with an Emacs mode (pygmy-mode) to help edit pseudo block files.

I first tried something like this with Riscy Pygness (for ARM chips) (`forthblocks.el`), but I wasn't completely happy with that first attempt.

I tried again for Pygmy Forth. I am much happier with pygmy-mode: (see the included file `pygmy-mode.el`).

The mode also has a command to renumber blocks which goes through the text file and renumbers all the blocks consecutively. Other keystrokes collapse or expand the blocks (to and from outline mode) or move to the next or previous block or shadow block.

Because the file is a text file, and the block boundaries are marked by special comment lines, each block can be as short or long as you like.

The word LOAD distinguishes between block files and text files by whether you give it a number or a file name:

```
1492 LOAD
```

loads block number 1492 (assuming the file had already been opened)

```
" utilities.fth" LOAD
```

loads the text file named `utilities.fth` from the current directory

```
" /home/frank/app1/utilities.fth" LOAD
```

loads a text file from an absolute path

Pseudo block files need to be opened before accessing them. At most a single pseudo block file can be active at any one time. So, the 1492 example above should really be

```
" columbus-navigation-aids.scr" OPEN
1492 LOAD
```

To use pygmy-mode, just copy `pygmy-mode.el` to somewhere in your Emacs load path, or, as shown below, give the full path to `pygmy-mode.el`, which assumes you put the file into your home directory.

Then add this to your `.emacs` file

```
(autoload
    'pygmy-mode
    "~/pygmy-mode.el"
  "Major mode to edit Forth pseudo block files." t)

(add-to-list
   'auto-mode-alist
   '("\\.scr\\'" . pygmy-mode))
(add-to-list
   'auto-mode-alist
   '("\\.blk\\'" . pygmy-mode))
```

(**Warning**, you should be able to copy and paste the above to your `.emacs` file from the HTML version of this manual. However, you if you copy and paste from the PDF version of this manual, the single and double quotation marks must be edited to change them from curly quotes to straight quotes.)

The `add-to-list` forms associate the file extensions `.scr` and `.blk` with pygmy-mode so that it will be started automatically when you open a file with one of those extensions. Edit the `add-to-list` forms to use suit your file naming conventions. (Currently, I use `.fth` to indicate a text file and `.blk` and `.scr` to indicate pseudo block files.)

To see the documentation for pygmy-mode, open a file in pygmy-mode then press `C-h m`. This runs the Emacs `describe-mode` command.

Pygmy-mode recognizes the special block marker comment lines and treats them as headings. See the pygmy-mode documentation mentioned above or see the file `stars.scr` for examples.

The keystrokes C-c C-n and C-c C-p move to the next or previous block. S-TAB cycles through the three visibility settings. C-v and M-v (and PgDn and PgUp) move to the next or previous block and also "narrow" the buffer to show just the single block. To "unnarrow" the buffer, press S-TAB.

Stacks

Because words, both CODE and high-level, are compiled down to Python procedures, there is not much need for the return stack except for its use as a "third hand" and for nested LOADs. PUSH and POP move an item to and from the return stack. R@ copies the top return stack item to the data stack. PUSHing and POPing addresses to affect flow control will not work.

Furthermore, stack items, unlike with previous Forths, no longer have a "width" and are not necessarily numbers. For example, a stack item could be a string itself rather than the address of a string. In practice, though, this makes little difference. COUNT, TYPE, etc., still work.

```
" This is a string" COUNT TYPE
```

Loops

As with my previous Forths, FOR … NEXT is the main looping mechanism.

```
: DOSOMETHING ( -) 10 FOR SOMETHING NEXT ;
DOSOMETHING
```

The above will do SOMETHING 10 times. The index counts down, so

```
: CD ( -)  3 FOR I . NEXT ;
CD
```

would print

```
2 1 0
```

Other looping commands are BEGIN … UNTIL, BEGIN … AGAIN, and BEGIN … WHILE … REPEAT.

Files

In addition to Python (version 3 or later), two files are required to run Pygmy Forth.

`pygmykernel.py`
: This defines enough of the Forth mechanism so it can load the rest of the system from `pygmy.fth`.

`pygmy.fth`
: This is loaded by `pygmykernel.py` to flesh out the Forth system.

The shell script `pygmy.sh` is also provided as an example of a shorter way to start Pygmy Forth. It may need to be edited to adjust it to your system. On a Microsoft Windows system, you could write a similar batch file.

CODE words

The Pygmy Forth kernel (`pygmykernel.py`) is written in Python. This has the advantage of running anywhere that Python can run, which is everywhere (at least Linux, Unix, other Unix-like systems, Microsoft Windows, Apple Mac OS—if you get Pygmy running on something else, including tablets and phones, I'd love to hear about it).

The Forth primitives (CODE words), instead of being written in assembly language, are written in Python. In effect, Python *is* the assembly language for Pygmy Forth.

Some utility Python functions (defined in `pygmykernel.py`) are useful for accessing the stacks and printing a list:

`dpush()`
: pushes its arguments to the data stack

`dpop()`
: pops one or more items off the data stack

`tos()`
: returns the top item on the data stack

`rpush()`
: pushes its arguments to the return stack

`rpop()`
: pops one or more items off the return stack

`dotList()`
: prints a list (used by `.S`, `.RS`, and `WORDS`)

Here is an example of a CODE word (from `pygmy.fth`):

```
CODE PUSH  rpush(dpop())  END-CODE
```

Note, there must be at least two spaces (or a line feed followed by a space) between the name (PUSH) and the start of the Python code. Otherwise, you will get the error:

```
The body of code word XXXX must be indented
```

The above code for PUSH pops an item from the data stack and pushes that item to the return stack.

Everything between the new word's name and `END-CODE` is written in Python.

The above definition can be written on a single line, but more complicated definitions may need to be written on multiple lines like this:

```
CODE PUSH
  rpush(dpop())
END-CODE
```

or

```
CODE EMIT
    x = dpop()
    if isinstance (x, str):
        print(x[:1], end='')
    else:
        print(chr(x), end='')
  END-CODE
```

Note that Python indentation rules must be observed.

If the Python code consists of multiple *expressions* (not statements), they can be written on a single line if separated by semicolons, like this:

```
CODE +
   a,b = dpop(2); dpush(a+b)
 END-CODE
```

or

```
CODE +  a,b = dpop(2); dpush(a+b)  END-CODE
```

Unlike traditional Forth, you cannot use Forth-style comments between CODE and END-CODE. Forth-style comments may be placed outside, though, like this:

```
( a b - a+b)
CODE +  a,b = dpop(2); dpush(a+b)  END-CODE
```

or, with proper indentation, Python comments may be used, like this:

```
CODE +
   # ( a b - a+b)
   a,b = dpop(2); dpush(a+b)
END-CODE
```

If the Python code consists of multiple *statements* (rather than expressions), then multiple lines must be used, like this:

```
CODE IF
  global _tab
  assemble ("if dpop():")
  _tab += 1
 END-CODE
```

Note that Python procedures defined as CODE words are not allowed to have parameters and, if they return values, those values are ignored. Instead, they must take their arguments from the data stack and return any values by placing them on the data stack. The functions `dpush()`, `dpop()`, `tos()`, etc., help with that.

```
a = dpop()
```

otherwise, dpop(*n*) (with $n > 1$) removes *n* items from the data stack and returns a tuple. If the data stack consists of (4, 5, 6, 7, 8) (8 is on top), then the top 3 items can be popped off and assigned to three Python variables with

```
a,b,c = dpop(3)
```

which sets a to 6, b to 7, c to 8, and leaves the data stack as (4, 5) with 5 on top.

The following would push 3 items to the data stack, leaving 9 on top.

```
dpush (7, 8, 9)
```

The functions `rpop()` and `rpush()` work the same way except on the return stack.

High-level Forth words

These are colon definitions and VARIABLEs.

Here are some examples:

```
: 0= ( n - f)  0 = ;
```

```
: STAR  ( -) '* EMIT ;
```

```
: STARS ( n -) FOR STAR NEXT ;
```

```
VARIABLE F1
F1 @ .    --> 0
13 F1 !
F1 @ .    --> 13
" hello" F1 !
F1 @ .    --> hello
F1 @ COUNT TYPE  --> hello
```

To define a constant, use a colon definition, e.g.,

```
: X  35 ;
: Y  72 ;
```

Compiling

Each Forth word is compiled as a Python procedure.

As always, a Forth word name may contain any non-white-space character.

When possible, the Forth word name is used as the Python procedure name. For example, the Forth word QUIT becomes the Python procedure QUIT.

However, when the Forth word is not a valid Python procedure name, the name is adjusted. For example, the Forth word + becomes the Python procedure `Plus` and the Forth word `;` becomes the Python procedure `Semi`.

The Forth dictionary maps the Forth word name to the name of the Python procedure. (You do not need to keep track of the Python procedure names.)

Forth Word Name Limitations

One caution: it *is* possible to step on a Python function. Generally, you are safe if you use upper case for the Forth word names. That is, OPEN would not step on the Python `open` function.

To help avoid such problems, a warning is displayed whenever a Python name is redefined.

True and False Values

Traditionally in Forth, false is represented by the integer zero and true is represented by any other integer. This is still true in Pygmy Forth but with an extension. The Python values `True` and `False` are also recognized by Pygmy Forth. (The Forth words TRUE or FALSE put the Python values `True` or `False` on the stack.)

So, we can define 0= like this

```
( n - f)
CODE 0=
   dpush (dpop() == 0)
 END-CODE
```

rather than

```
( n - f)
CODE 0=
   if dpop() == 0:
      dpush (1)
   else:
      dpush (0)
 END-CODE
```

In practice, we define = as a CODE word then define 0= as a high-level word, like this:

```
( a b - f)
CODE =
   a = dpop()
   b = dpop()
   dpush(a == b)
 END-CODE
: 0= ( n - f)  0 =  ;
```

or

```
( a b - f)
CODE =
  a,b = dpop(2)
  dpush(a == b)
 END-CODE
: 0= ( n - f)  0 =  ;
```

The Forth data stack can hold more than just integers. It can also hold `True`, `False`, strings, and floats.

Forth `IF`, `UNTIL`, and `WHILE` recognize `0`, `False`, and the empty string as false, and anything else as true.

VARIABLEs

A Forth variable is defined by the word VARIABLE, like this

```
VARIABLE STATUS
```

When the variable, e.g., STATUS, is executed, it puts its name (as a string) on the stack. The words @ and ! use this string to access the variable in the variables dictionary. Since we are not concerned the width of a stack item, there is no need for C@, C!, etc.

I'm not sure you would need to do it, but here is an example of how you could access the value of a variable within a CODE word:

```
VARIABLE STAT#
CODE CHECK-STATUS
   if variables['STAT#']:
      print ("Uh-oh, we've got trouble")
   else
      print ("Fortunately, status is OK")
 END-CODE
```

Unlike with traditional Forths, a Pygmy Forth variable does not have a fixed size and does not represent a memory address.

Constants

Constants are defined as colon definitions, e.g.,

```
: WIDTH 30 ;
```

Alternatively, constants could be defined as CODE definitions like the following, but there is no reason not to use the simpler colon definition.

```
: CODE WIDTH   dpush(30)  END-CODE
```

Unlike traditional Forths, there is no need to define small integers as constants. The only reason to define a constant is for naming clarity.

Glossary

In Pygmy Forth, WORDS will display the words in whichever vocabulary is current. COMPILER switches to the COMPILER vocabulary. FORTH switches back to the FORTH vocabulary.

The following sections list the words in the two vocabularies, along with their stack effects.

The FORTH words have a single stack effect.

The COMPILER words have two stack effects:

The first is the stack effect at compile time and the second is the stack effect at run time.

For example, consider the word ". It marks the beginning of a string. It is defined both in FORTH and in COMPILER.

The version in FORTH collects the string up to the ending quotation mark and pushes the string to the data stack, so its stack effect is (- s).

The version in COMPILER collects the string up to the ending quotation mark and lays down some code, so its stack effect at compile time is (-). At run-time, the laid-down code pushes the string to the data stack, so its stack effect at run-time is (- s).

In the stack effect comments,

- f stands for flag (something treated as a true or false value)
- n, n1, n2, etc., stand for numbers
- s stands for a string
- # stands for a count or length
- x, a, b, etc., stand for anything, or something appropriate for the context
- n | s stands for either a number or a string
- c | s stands for either a character code or a string

The hyphen separates the stack picture when the word begins to execute from the stack picture after the word finishes executing.

To see the definitions, look in the files pygmykernel.py and pygmy.fth.

FORTH vocabulary

! (value v -) "store" the value into the variable v

" (- s) put a string on the stack, e.g., " This is a string"

((-) ignore the input stream up through the closing parenthesis, i.e., start a comment

***** (a b - a*b)

+ (a b - a+b)

- (a b - a-b)

. (x -) print and remove top of stack

." (-) print the string, e.g., ." Hello"

.H (n -) print n in hexadecimal

.RS
(-) display the return stack

.S (-) display the data stack

/ (a b - a/b) divide a by b

0= (n - f) f is True if n is zero, else False

1+ (n - n+1) increment n by 1

2DROP
(a b -)

2DUP
(a b - a b a b)

; (-) marks the end of a colon definition

< (n1 n2 - f) f is True if n1 is less than n2, else False

<= (n1 n2 - f) f is True if n1 is less than or equal to n2, else False

= (a b - f) f is True if a equals b, else False

> (n1 n2 - f) f is True if n1 is greater than n2, else False

? (v -) fetch and print the value of variable v

@ (v - value) "fetch" the value of the variable v

ABORT
(s -) print the string and return to the interactive QUIT loop

AND
(f f - f) f is True if both inputs are true, else False

BL (- blank) put a space on the stack

BLOCK
(n - s) put the contents of pseudo block n on the stack as a string

BYE
(-) exit Pygmy Forth

CODE
(-) define a primitive in Python up to the END-CODE marker

COMPILER
(-) change context to COMPILER vocabulary

COUNT
(s - s #) put the length of the string on the stack

CR (-) print a carriage return

DROP
(x -)

DUP
(a - a a)

EMIT
(c|s -) print a character

END-CODE
(-) marks the end of a CODE definition

FALSE
(- False) push the Python False to the stack

FORTH
`( -)` change context to FORTH vocabulary

LOAD
`( n|s -)` load the block or file

NOT
`( f - f)` f is True if input is false and vice versa

OPEN
`( s -)` open a pseudo block file named s

OR `( f f - f)` f is True if either input is true, else False

OVER
`( a b - a b a)`

POP
`( - x)` move top of return stack to data stack

PUSH
`( x -)` move top of data stack to return stack

QUIT
`( -)` the main interactive loop (REPL)

R@ `( - x)` copy top of return stack to data stack

SWAP
`( a b - b a)`

THRU
`( first last -)` load a range of pseudo blocks

TRUE
`( - True)` push the Python True to the stack

TYPE
`( s # -)` print the first # characters of the string

VARIABLE
`( -)` define a variable, taking the name from the input stream, e.g., VARIABLE STATUS

WORD
`( s1 - s2)` collect input up to s1 and put it on the stack as s2

WORDS
`( -)` display words of the current vocabulary

XOR
`( f f -)` f is True if exactly one input is true, else False

: `( -)` start a high-level Forth definition

COMPILER vocabulary

" (-) (- s) compile a string that will be pushed to stack at run-time

((-) (-) ignore the input string up through the next closing parenthesis, i.e., a comment

." (-) (-) compile a string that will be printed at run-time

;; (-) (-) exit the Forth word at run-time, synonym for EXIT

AGAIN
(-) (-) end a BEGIN … AGAIN structure

BEGIN
(-) (-) begin a BEGIN … UNTIL or BEGIN … AGAIN structure

ELSE
(-) (-) begin the false part of an IF … ELSE … THEN structure

EXIT
(-) (-) exit the Forth word at run-time

FOR
(-) (n -) begin a FOR … NEXT structure

I (-) (- n) at run-time, push the loop index to the data stack

IF (-) (f -) begin an IF … THEN or IF … ELSE … THEN structure

NEXT
(-) (-) end a FOR … NEXT structure

REPEAT
(-) (-) end a BEGIN … WHILE … REPEAT structure

THEN
(-) (-) end the IF … THEN or IF … ELSE … THEN structure

UNTIL
(-) (f -) end a BEGIN … UNTIL structure

WHILE
(-) (f -) begin the conditional part of a BEGIN … WHILE … REPEAT structure

Afterword

My mind is not yet settled on whether the current implementation is the best approach, or even how useful it might be. Certainly, it has some quirks and lacks some features. Also, it has not been tested very thoroughly yet. I need to get some experience with it, and, hopefully, some feedback from *you* about your experiences with it.

I hope to put together a mailing list of Pygmy correspondents, etc., so I can mail out the occasional notice. If you would like to be on the list, just drop me a note at frank@pygmy.utoh.org.

– Frank
http://pygmy.utoh.org
frank@pygmy.utoh.org