PY65 – Python compiler for the 6502 microprocessor.

July 14, 2018

This is the first public release of my experimental Python cross-compiler.

It is available at https://github.com/BillGee1/PY65

These are 32-bit Windows console applications. I have been told they run in Wine. I also have a 16-bit MS-DOS version for use with something like DOSBox.

Language features implemented:

- Dynamic typing
- Types: integer, string, True, False, function (partial)
- Automatic memory management (using reference counting)
- Variable precision integers
- Integer operators +, -, *, //, %, &, ^, |
- Variable length strings
- String operators +, *
- Relational operators ==, !=, <, <=, >, >=
- if else
- while else break continue
- print function, including sep and end keyword arguments
- input function
- hardware access functions: peek, poke
- randint function

Usage:

py65 [/reinit] eggs
    generates eggs.a65 assembly language source file
    listing in eggs.lst, read this in case of compilation error
asm65 eggs
    generates eggs.s19 binary hex file
    listing in eggs.lst
em6502 eggs.s19
    symdeb style debugger
    g<enter> to run
    q<enter> to quit
    ?<enter> for help

Adapting to your system

The tools currently generate a binary image in Motorola srec hex format. The intent in the future is to directly generate the appropriate executable binary image format for the various 6502 computers.

The run-time library is contained in PYRTL.A65. It is configured for the virtual 6502 computer provided by EM6502.

Currently, the compiler “includes” portions of the run-time library source code into the programs it generates.

To adapt the run-time library to your system, save a copy of the original file. Then make changes to ADAPT.I65 as appropriate.

Lines beginning with “;;<”, “;;>” and “;;r” are markers for the compiler during file inclusion. Do not change those.

This behavior will change in the future when a linker is implemented.

ORIGIN

This symbol specifies the first memory location used by the program. Your system may have reserved memory locations above the stack; in that case, adjust the origin higher.

Note that the program will use some memory in the zero page regardless of this setting.

MEMEND

This symbol specifies the last memory location used by the program.

INCH

INCH returns a single character of input. The distribution version is for a virtual ACIA at the base address of $E000. You may modify this subroutine to match the address of your serial port, a different kind of port such as a UART or call your system ROM or operating system input routines.

The first time INCH is called, it randomizes the random number generator. If you modify INCH to call your ROM or operating system for input, it is highly preferable that you use a non-blocking port status check so that this randomization process continues to work.

OUTCH

OUTCH takes a single character of output. The distribution version is for a virtual ACIA at the base address of $E000. You may modify this subroutine to match the address of your serial port, a different kind of port such as a UART or call your system ROM or operating system output routine.

Program termination

EM6502 emulates a very simple 6502 computer consisting of the processor, RAM and a serial port. There is no operating system. Program termination consists of attempting to execute an invalid instruction $FF to pass control to the debugger in the emulator.

Place a jmp to reenter your ROM monitor or operating system in the subroutine EXIT_PY.

Technical details

Dynamic typing

In Python, variables are nothing more than a reference to an object. A pointer, in other words. A variable may refer to an object of any type.

Variables must be assigned before usage. They may not and cannot be declared; a variable is created by assignment.

A variable takes on the type of the object assigned to it. For example, this is valid Python code:

```
a = True
print(a)
a = False
print(a)
a = 1
print(a)
a = 'Hello world.'
print(a)
a = print
a('Wow!  You can do that?')
```

Variable precision integers

Integers in Python automatically grow as needed. Be aware that large integers can be slow to process.

Hardware access

Python does not have traditional pointers and Python variables do not behave like variables in other traditional programming languages. Hence, peek and poke functions are the only reasonable way to access arbitrary memory locations and I/O ports.

peek(address)

poke(address, value)

You must import them before using them:

```
from hardware import peek, poke
```

FAQS

Why compile Python and why the 6502?

I started to learn Python in early 2017. It quickly dawned on me that Python is not unlike a modernized BASIC. It is interactive and it is easy for a beginner to start programming.

I thought, “Python is popular and Arduino is popular. But there is no way to program an Arduino in Python. I can fix that”

Then the Vintage Computer Committee put a Commodore 128 on display. “I will initially target that machine since an Arduino UNO has only 2K of RAM and I am not sure whether that will be enough.” I now believe the 2K of RAM will not be much of a problem.

These machines are too small to host a full Python interpreter, so I have to implement a subset of the language. I find it is much more interesting to write a compiler than an interpreter.

Will PY65 eventually handle all of the features of Python?

No. For example, it will never allow executing an arbitrary string as code since that requires an entire Python interpreter to be present. Some features are just not practical on small machines.

But I intend to eventually implement all of the beloved Pythonic types, tuples, lists, dictionaries.

I am trying to call randint but the assembler cannot find it.

Just like in Python, you must import it before you can use it.

```
From random import randint
```

Can I define my own functions?

Eventually, but that is not implemented yet.

Why are the generated programs so large?

The S19 (Motorola srec) format represents each byte of machine code as two ASCII characters, so the file size is about twice the size of the binary image in memory.

Compounding the problem is my current lack of a linker, so the entire run-time library is included with every program.

That plus dynamic typing, automatic memory management and variable precision integers are complicated and require much code to implement.

Finally, like C, Python suffers from the “printf” problem. The print function must be able to handle every possible type it may encounter.

What other computers or processors may be supported in the future?

Because I already have floating point code written for the AVR (Arduino UNO) and the 6800/6809 (SWTPC running FLEX), those are likely next in line. Another possibility is the Intel x86 where I can require floating point hardware be present. WebAssembly is also an interesting option.

Why does my program quit with an out of memory error when I run it a second time without reloading it?

In Python, a variable is nothing more than a reference to an object, a pointer, if you will. An uninitialized variable cannot be used until something is assigned to it first. That something is often dynamically allocated in cases of variable-precision integers or compound objects resulting from a calculation. Before something can be assigned to a variable, the previous reference must be deleted and the associated data freed if it was allocated. In this particular case, variables are left referring to allocated data the first time the program is run. Freeing an object allocated in the prior run of the program corrupted the heap resulting in the out of memory condition.

But wait, there’s more. “print” is a variable whose initialized contents is a reference to a subroutine which formats and displays the value of data. It is perfectly valid in Python to assign another function to “print” to alter the way it works. When a program is restarted, “print” should revert to its original behavior. This means another list is required to enumerate variables and restore their original values.

A program can be made rerunable by adding a list of variables to be zeroed and another list of variables to be set to arbitrary values when a program starts. Because this behavior is not always needed, it is only generated when the “/reinit” compiler option is specified.