

Parrot bit 2: BASIC Parrot

Clinton A. Pierce, clintp@geeksalad.org

Abstract

The Parrot assembly programming environment provides a fun, simple, and back-to-basics experience for the assembly-language programmer, and until recently did not have a complete implementation of any established high-level programming language. Being somewhat bored and looking for a challenge, I created Parrot BASIC—a full-featured interpreted BASIC capable of running an established and time-proven software base.

1 Parrot BASIC

I began work on Parrot BASIC after reading an article on Perl.com about programming in Parrot Assembler (PASM). Assembler’s always been a fun thing for me and I was immediately overcome by waves of nostalgia for programming my Atari 6502 or my first real job involving 8088 assembler, and so I dove in with the sample programs.

First, I attempted was a XML parser in PASM, which was fun while it lasted but over with far too quickly. So I set my sights on something larger—a BASIC interpreter in PASM.

BASIC is a simple language to parse, and the parser would be in PASM. BASIC is also a simple language since it has no namespaces, scopes, or complex data types. I can implement my interpreter in stages and run programs almost immediately which means that BASIC’s large library of programs (games! diversions galore!), will be ready to run.

I also decided that to parse BASIC in something like Perl, to compile to PASM would be cheating. I wanted this to be a fully-interpreted BASIC with an interactive command prompt (“Ready”) all self-contained in one Parrot bytecode file. I based the design of the interpreter roughly on designs for 8k BASIC I saw back in the early 1980s.

2 Working with Parrot

Working with Parrot was an interesting experience. As I would bumble along coding this bit or that I would stumble into parts of the project that did not quite work yet. I would post a message to the internals developer’s list, and a few hours later someone would post a bug fix. Sometimes my problems were my own fault, bugs in Parrot, or I had stumble into pieces of the project that were not quite ready for prime time yet.

Looking back at my mail to the Perl 6 Internals lists (<http://lists.perl.org/showlist.cgi?name=perl6-internals>) and IRC log files ([#parrot](http://rhizomatic.net)), the response I got from the Parrot developers was great. Eventually things got to the point where we could not describe a small and concise bug report so the developers would just fire up BASIC and play a game of wumpus until it crashed to find the bug.

It was not all joy and sunshine, though. Working on a development platform where the developers change coding standards, add opcodes, or subtly change the way existing opcodes work provided more than a few headaches. Overall though, it was a terribly rewarding experience.

3 Following Along

If you want to follow along with the discussion, you can download the Parrot distribution which includes my BASIC interpreter. To get Parrot, go to <http://dev.perl.org> and follow the instructions. Dan Sugalski gave a gentle introduction to the build process in *The Perl Review* 0.3.

After you have built Parrot, change in to the `languages/BASIC` subdirectory and type at the prompt:

```
./basic.pl
```

This is simply a harness which calls the PASM assembler with the correct options and invokes the Parrot interpreter. BASIC will take a few seconds to build and you will soon be treated to a “Ready” prompt. If you quit out of the interpreter (`^C` or `QUIT`) and want to restart it, simply type

```
../../parrot out.pbc
```

and it should start instantly without all of the re-assembly.

4 Guts of the Interpreter

The interpreter code has 5 major subsections.

- The harness, which compiles the PASM and provides the interactive prompt (`basic.pl`)
- The statement dispatcher and BASIC statement code (`basic.pasm`, `instructions.pasm`)
- The BASIC variable storage routines (`basicvar.pasm`)
- The expression evaluator (`expr.pasm`)
- Support libraries (`stackops.pasm`, `alpha.pasm`, `dumpstack.pasm`, `tokenize.pasm`)

5 Coding conventions

I began work on Parrot BASIC while the developers were still debating PASM style, and so the code presented here does not conform to all of the ‘standards’ they later chose. I used callee-save (as opposed to caller-save) and pass arguments to subroutines on the user stack instead of through registers.

I use the stack for argument passing because my assembler programming history is on stack machines (6502, 8088) and I am used to seeing the world through a series of stacks. The one register-based architecture I

used left such a bad impression on me that I may never fully recover, although this was the fault of the OS, not the register concept.

Since I use the stack to pass arguments, I can write self-contained subroutines since I do not expect registers to have anything useful in them. I manage the user stack with these guidelines:

1. Most support library calls pull a fixed number of arguments from the stack and push a fixed number back on. The caller and callee must both be careful to save and restore the same number of arguments in the same order.
2. In some cases, a variable number of elements will be passed between the caller and the callee. In this case, the top item on the stack is an integer which indicates how many items on the stack are being passed. The callee is always expected to return the stack back to this state (with the depth on top).
3. Some subroutines (expression evaluator, tokenizer, peek) take a fixed number of arguments, then deal with the stack using those arguments. In this case, the callee is expected to put the stack back together (depth on top) and then return its arguments on top of that.

At runtime, PASM uses the user stack to parse BASIC, evaluate expressions, and so on. At the same time, the use stack is also the BASIC runtime stack. I keep the runtime stack at the bottom and the working stack on top of that. For example, after processing:

```
10 FOR I=1 TO 50
20 GOSUB 100
30 NEXT I
100 LET A=90*I
110 RETURN
```

At line 100 the entire user stack would look like:

```
6          # Current stack depth
LET
A
=
90
*
I
10         # Runtime stack depth
GOS
20         # Line number of GOS
0          # Unused
0
0
0
FOR
10         # Line number of FOR
0          # Unused
0
50         # Termination expression
1          # Step
```

When the interpreter is done with line 100, the top item on the stack should be the depth of the runtime stack; each BASIC statement handler is expected to clean up after itself.

Since the callee was saving everything it took a bit of cleverness to call a subroutine that could modify a “global” variable used in the interpreter. Most subroutines are coded something like:

```
CHANGE_PC:  pushi
            # lots of code here, changing things, etc...
            set I23, I0  # Change the PC.
            popi
            ret          # Whoops, PC didn't get changed.
```

Which would have no effect as I23 would be reset on with the popi. Since within a callee-save subroutine I scope everything as though it were a Perl local(), I labeled this as an *unlocal*:

```
CHANGE_PC:  pushi
            # lots of code here, changing things, etc...
            set I23, I0
            save I23     # Preserve it from the popi
            popi
            restore I23 # Put it back.
            ret
```

And now I2—as it was set in the subroutine—would be preserved after the subroutine call. Sometimes this unlocal madness can be quite severe.

6 Support Libraries

A surprising amount of code in any assembler programming is not the actual opcodes themselves, but setting up calls to libraries and dealing with the results. For example, to divide a statement into tokens on the stack for processing I need functions to test for both alphanumeric sequences and whitespace.

The ISALPHA function tests for alpha-numeric characters and is typical of the types of support functions written for BASIC.

```
ISALPHA:
    pushi
    pushs
    restore S1
    ge S1, "A", UPPER    # Yes, this is ASCII.
    branch NONUP        # If you want ISO 8859 or
UPPER:  le S1, "Z", ALPHA # Unicode, you didn't want BASIC.
NONUP:
    ge S1, "a", LOWER
    branch NONLOW
LOWER:  le S1, "z", ALPHA
```

```
NONLOW:
    ge S1, "0", NUMBER
    branch NONUM
NUMBER: le S1, "9", ALPHA
NONUM:  eq S1, "_", ALPHA
        # Not A-Z0-9_
        set I1, 0
        branch LEAVE_ISALPHA
ALPHA:  set I1, 1
LEAVE_ISALPHA:
    save I1
    popi
    pops
    ret
```

The subroutine begins by pushing Parrot's Ix and Sx registers onto their appropriate stacks. This allows the subroutine to use all of the Ix and Sx registers with impunity, as long as it restores them before it returns.

The `restore S1` statement removes a string from the stack that is the character to be tested. After a series of greater-than/less-than tests, the flag variable I1 is either set to 1 (alphabetic) or 0 (non-alphabetic). I push this onto the stack, pop the Ix and Sx registers, and return from the subroutine.

To use the subroutine:

```
save S10
bsr ISALPHA
restore I0
```

Because the subroutine saves the state of the caller first, I do not need any special preparation other than to put the arguments on the stack and pull the results off. The other major library routines are:

- `TOKENIZER` – tokenize a string
- `ISALPHA`, `IS WHITE` – test a character's type
- `STRNCHR` – search for characters within strings
- `ATOI`, `ITOA`, `ISNUM` – convert strings to integers and back[1], test for numeric-ness
- `PAD`, `STRIPSPACE`, `STRIPLEADSPACE` – insert or remove excess whitespace
- `STRSTR` – search a string for a substring
- `PEEK` – fetch a value from an arbitrary depth on the user stack
- `SWAP`, `REPLACE`, `REVERSESTACK` – edit values on the user stack
- `CLEAR` – clear the user stack
- `SORTSTACK`, `NSORTSTACK` – sort the stack alphabetically, numerically.

6.1 Variable Storage

In Parrot BASIC I require storage for three different kinds of data:

- The code for the BASIC program itself
- Numeric variables
- String variables

Before May, I did this with linked lists stored in conventional string registers; unpacking and searching substrings for values in larger string registers. This was slow, but trustworthy.

In May the Parrot team completed initial work on the PerlHash and PerlArray PMC types in Parrot assembler. I scrapped the old, slow variable storage methods and replaced with PMCs. Three PMC registers (P21, P22 and P23) hold a copy of the numeric variables, string variables, and the program code respectively.

I keyed the variable PMC's on the variable name. I create multi-dimensional arrays with compound key of the variable name and the offset. I store the BASIC program itself in P23 and keyed by line number. A parallel structure of type PerlArray keeps an index of the lines in order.

From the PASM BASIC programmer's standpoint, to create a new (numeric) variable it all boils down to:

```
save 42      # Value to store
save "N2"    # Variable to create
bsr NSTORE   # A thin wrapper for VSTORE, which inserts into
```

To fetch it back later:

```
save "N2"    # BASIC variable to fetch
bsr NFETCH
restore NO   # IO now has N2's value (42)
```

With the PerlHash and PerlArray PMC types, storage in PASM got a whole lot simpler to manage.

7 Expression Evaluation

Another large component of the interpreter is the expression evaluator (`expr.pasm`). Its job is to take an expression on the stack and replace it with a value. Take the statement:

```
IF X < Y$*3/-2 THEN GOTO 50
```

The tokenizer will leave this on the stack as:

```
13      # depth
IF
X
<
Y
$
*
3
/
-
2
THEN
GOTO
50
[runtime stack below this]
```

The IF-handling subroutine will remove the IF and decrement the stack depth. Next, it pushes a stop-word onto the stack and calls EVAL_EXPR. The stop-word tells the expression evaluator when to stop.

```
restore I5      # Depth
restore S0      # "IF"
dec I5
save I5         # Put the depth back
save "THEN"     # Process until "THEN" is seen
bsr EVAL_EXPR  # True or false?
restore S0      # The return value...
```

The stack after EVAL_EXPR will contain everything from the stop-word down, and on top it will have the return value from the expression evaluation.

Expression evaluation happens in three steps:

1. The expression stack is “cooked” a bit: unary minus is glued onto its argument again (they were separated during tokenizing) and \$ is tacked onto the previous item so that string variable names are whole again. Functions (and subscripts) are re-written from RND(3*2) to RND! ~ (3 * 2) so that ~ becomes something that looks like a bind operator.
2. Next, the expression is converted from Infix notation to Postfix notation. The precedence of the operators is fixed here, parenthesis removed, and evaluation order determined. To pull this off, I had to implement a second “stack” in a string register. The expression in the IF statement above in RPN looks like: X Y\$ 3 * -2 / < . This becomes our input queue for the next step.
3. Last, the input queue is emptied one element at a time and evaluated. The nice thing about Postfix is you simply push things onto a stack from an input queue. When you find an operator on the queue, pull two items from the stack, operate on them, put one item back. When the input queue is exhausted you’re done evaluating, whatever is on the stack is your result. The RPN given would be evaluated as follows: push X on the stack, push Y\$ on the stack, push 3 on the stack, to process * pull the top two items, Y\$ and 3, multiply them and put the result back. And continue.

When functions are encountered, they pull all of the necessary items from the expression stack and put their own value back (if any). Variables (non-subscripted) are substituted during step 3 above. If there's a subscript they fall into the function-evaluation routines, it recognizes that they're not a function, and performs a variable lookup.

All the while, strings are quietly interchanged for numbers and vice-versa. This is why we can multiply Y\$ and 3 in the example above.

7.1 Running a Program

A BASIC program runs terribly simple. The interpreter fetches a line of the program, tokenizes it, grabs the command to be run—the second token—and determines if it is a valid command (`basic.pasm`). Next, it jumps to the appropriate subroutine in `instructions.pasm` and then removes the rest of the tokens from the stack and does whatever they say. It cleans up the stack, and fetches the next line. Repeat as necessary.

For example, I implemented the GOTO (or GO TO) statement with this (somewhat over-commented) code:

```
# GOTO EXPR
# GO TO EXPR
I_GOTO: pushi
        pushes
        restore I0      # Line Number (supplied by basic.pasm)
        restore I5      # Depth
        restore S0      # Keyword "GO" or "GOTO"
        dec I5
        eq S0, "GOTO", I_NOTGO_TO
        restore S0      # Hope this is a "TO"
        dec I5
I_NOTGO_TO:
        save I5          # Stack's kosher now.
        save "REM"       # Stop-word
        bsr EVAL_EXPR    # Leaves result string on stack
        bsr ATOI         # Convert that to an integer
        restore I23      # Adjust PROGRAM COUNTER.
        bsr CLEAR        # Clean up stack
        restore I0       # Dummy value (depth of 0)
END_I_GOTO:
        save I23
        popi
        pops
        restore I23
        ret
```

The whole purpose of this routine is to set I23—the BASIC line number counter—to the new line number which is an expression.

8 Conclusion

Parrot is a charming CPU to code for, and porting BASIC to it was a gentle experience. Parrot programmers should be able to port other languages than can have small implementations (but often do not) like FORTH, Lisp, or Pascal.

9 About the Author

Clinton Pierce is the author of *Teach Yourself Perl in 24 Hours* and *The Perl Developer's Dictionary*. Currently he's writing financial software in Perl and C. For more information see <http://geeksalad.org>.

10 References

“Parrot: Some Assembly Required” by Simon Cozens
<http://www.perl.com/pub/a/2001/09/18/parrot.html>

Perl 6 Internals developer's list
<http://lists.perl.org/showlist.cgi?name=perl6-internals>

“Parrot Bits: Bit 1, The Parrot Vooms!”, *The Perl Review* 0.3, Dan Sugalski
http://www.theperlreview.com/Issues/The_Perl_Review_0_3.pdf