

Parroty Bits: Bit 1, The Parrot Vooms!

Dan Sugalski, dan@sidhe.org

Abstract

In which I lure the unsuspecting reader into the wonderful world of Parrot programming. Along the way I will show you some of the features of the Parrot interpreter, and get you well on your way to writing programs that take advantage of its speed and power, before Perl 6 is even finished!

1 Introduction

In my last article I presented a broad overview of Parrot, the interpreter behind Perl 6. This month I show you some programs. All you need to follow along is a working version of Parrot, a C compiler, and Perl 5.005.03 or higher to build it.

Parrot is a bytecode machine, and one of the things it can do is execute bytecode that has been stored on disk. Since there is not a working Perl compiler for Parrot yet, we—the Parrot Development Team—wrote an assembler.

An assembler is a program that takes a symbolic representation of machine code and turns it into actual machine code. Assembly code itself is very low level—each symbol, or mnemonic, corresponds to a single machine or bytecode instruction. That gives me complete control over the bytecode that is fed to the interpreter, which is good for testing, and it means that you do not have to wait for a working Perl parser to start exercising and using Parrot.

By default, Parrot assembly language files have a `.pasm` extension, and compiled bytecode has a `.pbc` extension. To assemble a program, feed it through `assemble.pl`, the Parrot assembler.

```
./assemble.pl file.pasm --output file.pbc
```

To run compiled programs, I use the `parrot` program that was built when I configured and built parrot. I pass it the name of the compiled bytecode file as its parameter.

```
./parrot file.pbc
```

If I feed bad code into the interpreter, it will crash. Generally the language compiler makes sure that bad code is not created, but since I am doing this all by hand it is my responsibility. If things go wrong it may cause a core dump, crash, burn, or otherwise fail in spectacular ways.

1.1 A simple example

I start with the obligatory “Hello, world” program which is simple, straightforward, and to the point.

Code Listing 10: A simple program

```
1 print "Hello, world\n"  
2 end
```

The `print`, as you might expect, prints a string to standard output; `end` marks the end of the program. All programs must exit with `end`—if a program falls off the end of the world, the interpreter will likely crash.

1.2 Swinging without a net

When I create Parrot bytecode by hand, I work without a safety net. Well-formed Parrot programs should never crash the interpreter, and the Perl compiler should never create anything but well-formed Parrot code. Raw assembly, on the other hand, can create all manner of abominable code which is free to kill the interpreter.

This is on purpose—safety is expensive. Every microsecond the interpreter spends making sure some code is correct is a microsecond not spent executing that code. If the code is correct, and any code coming out of a compiler should be, the interpreter should not do this—either the code is correct, or the compiler is broken. If the compiler is broken, all bets are off anyway.

I may want to check the bytecode before executing it if I do not trust it—when I execute code from a remote source for example. The interpreter provides a safe mode during which it checks arguments to bytecode, verifies destinations of branches, and enforces resource limits.

1.3 A bit more detailed: counting from 100 million

Of course, `Hello, world` is not that useful. I want to try something a little more interesting—counting from 100 million down to zero. Code listing 11 shows how I wrote this in Perl.

Code Listing 11: Perl countdown

```
1 $x = 100000000;  
2 do {  
3     $x = $x - 1;  
4 } while ($x);
```

Code listing 12 shows the equivalent Parrot assembly code. It has four lines, and most of them introduce new things.

Code Listing 12: Parrot assembly countdown

```
1     set I0, 100000000  
2 REDO: sub I0, I0, 1  
3     if I0, REDO  
4     end
```

The first line, `set IO, 100000000`, puts the value 100 million into integer register 0. The destination is the first argument. In all cases where something is stored, the destination is the first operand. In this case, it is an integer register, which I discuss later. In line 2, `REDO: sub IO, IO, 1`, subtracts 1 from the contents of integer register 0 and stores the results back in integer register 0. Once again, the destination comes first. The `REDO:` is a label—it gives this line of code a name that I can refer to later. In line 3, `if IO, REDO`, checks integer register 0 is true and, if so, branches to the label `REDO`. As far as Parrot is concerned, an integer is false if it has a value of zero, and is otherwise true. Line 4 exits the program, just like in my “Hello, World” program.

The Parrot version takes 5.78 seconds of user time for me. The Perl version takes 170.9 seconds of user time, though it only takes 169.3 seconds of user time when running with `use integer;`.

1.4 Care and feeding of registers

A register is a sort of named temporary spot, a place that the interpreter can get to very quickly and with a minimum of fuss.

Parrot has a lot of registers, and four register types. In addition to integer registers, which hold a machine-native integer, it has floating point registers which hold machine-native double-precision floating point numbers, string registers which hold pointers to strings, and PMC registers which hold pointers to PMCs, or Parrot Magic Cookies, which correspond to Perl-level variables.

Parrot has 32 of each type of register, numbered from 0 to 31. When I refer to a register, I prefix it with its type—P for PMC registers, S for string registers, I for integer registers, and N for floating point number registers.

Parrot’s register usage is RISC-like, and like most hardware RISC processors such as the Alpha, SPARC, or PPC, it can only do operations on data in its registers. Parrot cannot directly change data stored in an arbitrary named variable—it must first put the PMC for that variable in one of its PMC registers and operate on that.

For comparison, other virtual machines, such as Java’s JVM, .NET, or other interpreters such as Perl 5, Python, and Ruby, are stack-based. They operate on elements that are on top of a stack. If I want to add two numbers together I first push them onto the stack, then call an add operation that takes the two numbers off the stack, adds them together, and puts them back on the top of the stack. Stack machines tend to have more concise bytecode because they do not need to know where things are going—they always go on the top of the stack. On the other hand, they tend to spend a lot of time twiddling with the stack, something Parrot does not have to do nearly as much of.

1.5 Going places

Programs are not that interesting if they do not make decisions and choose to go places. Even my simple example above makes a decision. Parrot has a set of comparison operators, detailed in this comparison table:

if x, dest

 If register X is true, jump to label dest

unless x, dest

If register *X* is false, jump to label *dest*

eq *x*, *y*, *dest*

If the contents of registers *X* and *Y* are the same, jump to label *dest*

ne *x*, *y*, *dest*

If the contents of registers *X* and *Y* are different, jump to label *dest*

gt *x*, *y*, *dest*

If the contents of register *X* is greater than the contents of register *Y*, then jump to label *dest*

lt *x*, *y*, *dest*

If the contents of register *X* are less than the contents of register *Y*, then jump to label *dest*

ge *x*, *y*, *dest*

If the contents of register *X* are greater than or equal to the contents of register *Y*, then jump to label *dest*

le *x*, *y*, *dest*

If the contents of register *X* are less than or equal to the contents of register *Y*, then jump to label *dest*

All of these operators can take any pair of registers, though both registers generally must be the same type. The **if** and **unless** operators use Perl's idea of truth and falsehood for integers, strings, and numbers. If a number is zero, or a string is empty or the single character 0, Parrot considers it false, otherwise true.

Truth and falsehood for a PMC is a slightly trickier thing. Each PMC class is responsible for deciding whether a particular PMC is true or false. This may seem wishy-washy on the part of the interpreter, but I think class authors are in a better position to determine this than Parrot is.

The destination of a comparison is not an absolute location but an offset, a positive or negative integer, that indicates how far forward or back the program should go. While this theoretically limits the size of any one code segment, integers here are 32 bits wide, giving us offsets of up to 2 gigabytes. Parrot can reach code segments larger than 2 gigabytes with multiple hops. Moving forward or backward in a program by a relative amount is called a branch, and this is how the machine transfers control within a routine. A branch works regardless of where your code is, which means the interpreter can load bytecode off of disk and put it wherever is convenient.

In addition to the conditional operators, which will all branch only when their conditions are met, I can unconditionally change the flow of my program with the flow control operators.

The **branch** operator unconditionally branches forward or backward. They are often found at the very end of loops, transferring control back to the start to recheck the loop condition. Code listing 13 shows a branch at the end of the section labelled LOOP. The corresponding Perl program in code listing 14 does the same thing at the end of the while loop.

Code Listing 13: Branching in Parrot

```
1         set I0, 10
2         set I1, 0
3 LOOP:  unless I0, END
4         sub I0, I0, 1
5         add I1, I1, 1
```

```

6         branch LOOP
7     END: print "Done\n"

```

Code Listing 14: Branching in Perl

```

1     my Int $i = 10;
2     my Int $j = 0;
3     while ($i) {
4         $i = $i - 1;
5         $j = $j + 1;
6     }
7     print "Done\n";

```

The `jump` op unconditionally transfers control to an absolute address. The destination is almost never a constant, since I cannot know absolute addresses when I write or assemble my program. Instead a program will generally fetch the address for a named subroutine into a register and jump to it that way.

The `jsr` (Jump to SubRoutine) and `bsr` (Branch to SubRoutine) ops are identical to the `jump` and `branch` ops, respectively, with a single exception. Before transferring control to the destination, they push the address of the next instruction onto the control stack. These are the two ops your programs will use to invoke subroutines.

The `return` op takes the address off the top of the control stack (the one that `jsr` and `bsr` put there) and transfers control there. It is used, as you might expect, to return from a Parrot subroutine.

1.6 Stacks

While Parrot is a register machine, stacks are still handy things to have. Many languages, such as Forth and Scheme, depend heavily on stacks, and having them is convenient when you need a place to stuff something temporarily and a full-fledged entry in a symbol table is more trouble than it is worth.

Because of this, Parrot has stacks. Six of them, in fact—a general-purpose stack for temporary storage, a control stack to store return addresses and the like, and each of the four sets of registers has a private stack.

Parrot uses the register stacks to save the contents of all the registers in one go. Since Parrot uses the registers to do things, they need to be saved across calls to subroutines. With one or two registers, the general stack is fine. Parrot has 128 registers though. Pushing them all one by one is a bit much. So, I created a savestack for each register type so I can save all the registers in one go. The `pushi` opcode, for example, pushes all the integer registers onto the integer register savestack, while the `popi` pops the top of the integer register stack into the integer registers.

Parrot uses the general-purpose stack to save individual registers, and sometimes to pass parameters to subroutines. The values pushed onto the generic stack are typed—if I save an integer register, and then try to restore it into a string register, for example, the interpreter will throw an exception.

When calling a subroutine, the caller is responsible for making sure that its registers are saved off, and subroutines are **not** required to be careful with registers, although subroutines may not leave anything on the stacks. This is a bit different than many systems, which mandate that the callee—the subroutine—is

responsible for saving the registers it uses. Like many other things in Parrot, I did this on purpose. It turns out that the only way to do efficient tail calls and tail recursion is to have the caller save off its registers. Efficient tail calls turn out to be essential for a number of Perl constructs, and of course, languages like Lisp and Scheme make heavy use of them, so it turns out to be a nice win.

2 Wrapping it up

I have not talked about quite a bit of Parrot, including subroutines, lexical and global variables, string operations, or exception handling, for example. Do not worry, Parrot can do that, and I will cover it in a later installment.

3 References

PDD 6, as distributed in the Parrot source tree. (`docs/pdds/pdd06.assembly.pod`) All the valid opcodes are detailed in there.

The Parrot source. CVS checkout instructions and snapshots available from <http://dev.perl.org>.