

Parrot Bits: Bit 0, The Beginning

Dan Sugalski, dan@sidhe.org

Abstract

I take you on a whirlwind tour of the parrot project, the new interpreter engine that Perl 6 will use. I discuss some of the motivations behind designing a new interpreter engine, and some of the ways that Parrot fixes Perl 5's current limitations.

1 What is Parrot?

Parrot is the interpreter engine behind perl 6. but it is not Perl 6. Perl 6 is a language, the revision of perl that Larry Wall is designing. Parrot, on the other hand, is an interpreter system. They are separate for this version of perl, and their development proceeds separately.

Parrot is object-oriented, dynamically typed, threaded, scalable, platform-independent, and language agnostic. On some platforms, notably x86 and Alpha Linux, x86 and Alpha *BSD, and SPARC Solaris, it has a Just In Time (JIT) compiler.

The name "Parrot" comes from the elaborate 2001 April Fool's joke perpetrated on an unsuspecting world by Simon Cozens and a horde of minions. The joke said that Larry and Guido van Rossum, the creator of Python, got together to design a joint language called Parrot. While it was originally a joke, it set Simon's fertile brain in motion. As so often happens, Life imitates Satire, and Parrot was born.

I want to emphasize that Parrot is language neutral. While Parrot must run Perl 6 code, we designed it to run code from other languages, such as Python and Ruby, that are similar to Perl. Not only does this make Parrot a more capable engine, but it also means we can potentially run code written for other languages. Perl 5 already lets you use libraries of C and C++ code, now Parrot can let you use Python and Ruby libraries in your Perl 6 code.

2 Why Parrot?

We started work a new engine because all the ones available to us now do not cut it in one form or another. The perl 5 engine is over seven years old, and has been extended and expanded well beyond its original design, and it shows. Java's JVM is geared towards statically typed languages – perl is weakly typed and late binding. The .NET framework limits us to the platforms that .NET runs on, ruling out many of the current platforms such as the Cray or IBM's big iron systems. Since nothing else meets our needs – not even Scheme – we designed parrot.

Separating the language and interpreter design gives us a number of advantages. Larry Wall, Perl 6's language designer, is free to spend all his time designing the language and leave the implementation to other

people. This is a big help – designing a language the size of Perl 6 is a big task. Designing the language, designing the interpreter, and implementing the interpreter is too big a task for one person.

We can work on the interpreter engine before Larry finishes the language specification, since many of the details of the language do not affect the core design. The interpreter only needs to know the semantics of the language, not the syntax – Parrot needs to know what things Perl 6 does, not how it looks.

A separate interpreter lets us cast a wider net than just Perl. Many of the dynamic languages, including Python and Ruby, have semantics similar to perl. They do pretty much the same stuff perl does and in the same way. Even though they look different, they act the same. Method calls are method calls, and the interpreter does not care whether you use curlyes or white space to denote blocks.

3 Parrot’s goals

Parrot’s design goals are simple.

- Execute programs quickly
- Present a clean extension mechanism
- Present a clean embedding mechanism
- Last for ten years before needing another overhaul

Execution speed is the most important – Parrot is not much use if it is slower than Perl 5. While it is too early to do extensive benchmarks, in our tests Parrot is three times as fast as perl 5, and an optimized version of the test runs about 10 times as fast. That lead will shrink with more substantial benchmarks, but we can readily achieve a target of 20% faster than Perl 5 without counting the JIT or Parrot-to-C compiler, of course, which would be cheating.

The clean extension and embedding interfaces will come as a huge relief to anyone who has had to deal with XS or SWIG. Perl’s current extension and embedding interfaces were something of an afterthought, and uncharted territory at the time. It shows.

Longevity, the last goal, is the trickiest, as we have no real way to tell what the future will bring. All we can be certain of is that it will bring things we have not considered, so we designed Parrot’s to be as flexible as it can be.

You might be surprised that speed is more important than extendibility. Indeed, the current trend in languages is to design things that are clean and pretty inside, and sweep performance issues under the rug until Moore’s Law cleans them up. That is foolish since existing hardware is not getting any faster.

4 Parrot parts

The Parrot interpreter system is broken into four different parts.

4.1 The Parser

The parser takes raw source, whether it is Perl, Python, Ruby, or Scheme, and turns it into an Abstract Syntax Tree, or AST. An AST is a symbolic representation of your program that the compiler and optimizer can manipulate, though not yet anything the interpreter can execute.

4.2 The Compiler

The compiler takes an AST from the parser and translates it into something suitable for the Parrot interpreter to execute. The translation is straightforward, and the target does not have to be the Parrot interpreter. We can, and will, have compiler modules that target .NET and the JVM.

Targeting multiple back ends means your programs are not forced to run on the Parrot interpreter, and provides more cross-language compatibility. You could write part of your Java or C# programs in Perl 6, for example.

We do not throw out the AST after the compiler is done with it. The optimizer and debugger can make good use of it, and you never know when it might come in handy later on.

4.3 The Optimizer

The optimizer takes the bytecode the compiler module produces, along with the AST it was generated from, and makes the bytecode more efficient. Writing an optimizer for Perl is tricky, since Perl's dynamic nature makes an awful lot of things un-optimizable. Many of the classic optimizations assume that data is static, and its behavior is known at compile time, which is definitely not the case with Perl. We can still do things like constant folding (replacing expressions like "5 + 5" with 10) though.

Perl 5's optimizer is hobbled by a few things that we avoid with Parrot. Perl 5 has an integrated compile and run cycle with no way to save the resulting compiled program to disk. There is no point in spending 30 seconds to optimize a program if the optimizations save only 5 seconds.

Parrot, on the other hand, can load bytecode from disk, skipping the compilation phase entirely. While it still may take 30 seconds to optimize your program, and still save only 5 seconds per run, when you save it to disk you only need to run it 7 times to make the time spent worthwhile. When running a program from source, the way Perl 5 does, we will use a minimal set of optimizations by default.

Perl 5's optimizer assumes that programs are small. A few dozen lines of code and a couple of seconds to run through some external data file. There is little point in optimizing a program like that.

We know that modern programs are much larger than that, and might run for hours or days. Thirty seconds or a minute of extra startup time is well worth shaving even 1% off a program that runs for eight hours. Parrot lets you enable full optimizations even with compile-and-go style programs, though that will not be the default.

Finally, you can add type specifiers, such as "Int", to variables in Perl 6, which gives the optimizer more hints as to what your program does so it can optimize better.

4.4 The Interpreter

This is where the really interesting things happen. The interpreter module takes bytecode, either from the compiler/optimizer or loaded from disk, and executes it. Most of our development effort is focused here.

Parrot's interpreter is a register-based design that resembles a hardware CPU more than a traditional interpreter. The core design is predicated on the idea that memory is plentiful but using it is slow, CPU pipelines are deep, and L1 caches are reasonably large. This simplifies code generation and JIT compilation and optimizes better than the traditional stack-based approach. we will cover it in detail in later articles.

The core variable type in the interpreter is a Parrot Magic Cookie, or PMC, which I will discuss in much more detail in another article. We designed PMCs assuming that programs will tie them, share them between interpreters, or overload their operations. We made sure that the cost of overloading operators, thread-safing variable access, or calling tied methods or subroutines would only be paid by the variables that are actually overloaded, shared, or tied.

With perl 5, every time your program does anything with a variable – read from it, write to it, do any sort of math operation with it, turn it to a string, integer, or float – the interpreter has to first check if there is custom code attached to the variable. That is a lot of time spent checking, and wasted time at that, since most programs have a vanishingly small number of tied or overloaded variables.

The interpreter also has support for more complex programming tools, such as coroutines, continuations, and exceptions, as well as threading support built in from the ground up.

5 What works today

The interpreter engine is currently at about the level of your average CPU (about equivalent to an Alpha or PPC CPU). We can do basic math and string operations, flow control, input/output, and stack operations. The core of a regular expression engine is also functional. It is Turing complete, for those folks who care about such things. Operations on simple types – strings, integers, and floating point numbers – work, as do operations on more complex variable types.

The bytecode loader, interpreter core, and assembler work. We can write Parrot programs in a sort of high-level assembly language, turn those programs into bytecode and store it on disk, load that bytecode off disk, and execute it.

Normal perl scalars work as they should. You can create a string variable, turn it into a number to do math with it, then turn it back to a string for printing.

Arrays and hashes mostly work. We can store integers, floats, and strings into arrays and hashes, and we can fetch them back out again. By the time you see this, we should be able to store perl variables in hashes and arrays and fetch them back out again.

We have several small language parsers, including a Perl subset and a Scheme parser. We also have Jako, a new C-style language being developed in conjunction with the interpreter. Our miniperl so far handles simple control structures, most math operations, and scalar variables.

Part of the garbage collection and memory allocation system works. We can allocate memory and we know where its being used.

6 What works soon

The garbage collector does not yet find or collect garbage.

We do not yet have either lexical or global variables you can look up by name. This is waiting on functional hashes – once we have those, globals and lexicals are simple.

Methods and subroutines do not yet work

The parser, compiler, and optimizer are all written in perl, and need perl 5 to work.

7 Where can I get more information?

You can check out the parrot source through anonymous CVS:

```
cvs -d :pserver:anonymous@cvs.perl.org:/cvs/public login
[press enter at the password prompt]
cvs -d :pserver:anonymous@cvs.perl.org:/cvs/public co parrot
```

You can also get a repository snapshot which we generate every six hours from <http://cvs.perl.org/snapshots/parrot/parrot-latest.tar.gz>. The source should build on most Unices and Windows (both native and in CygWin) and, if we are lucky, VMS.

The Perl 6 internals list is the canonical place for Parrot development. Send mail to perl6-internals-subscribe@perl.org to subscribe. Additional information's on the web at www.parrotcode.org and dev.perl.org, and you are welcome to drop by #parrot on IRC, on irc.rhizomatic.net.

And, of course, There is always next issue.