PARROT - QUICK GUIDE

WHAT IS PARROT

When we feed our program into conventional Perl, it is first compiled into an internal representation, or bytecode; this bytecode is then fed into almost separate subsystem inside Perl to be interpreted. So there are two distinct phases of Perl's operation:

- Compilation to bytecode and
- Interpretation of bytecode.

This is not unique to Perl. Other languages following this design include Python, Ruby, Tcl and even Java.

We also know that there is a Java Virtual Machine *JVM* which is a platform independent execution environment that converts Java bytecode into machine language and executes it. If you understand this concept then you will understand Parrot.

Parrot is a virtual machine designed to efficiently compile and execute bytecode for interpreted languages. Parrot is the target for the final Perl 6 compiler, and is used as a backend for Pugs, as well as variety of other languages like Tcl, Ruby, Python etc.

Parrot has been written using most popular language "C".

PARROT INSTALLATION

Before we start, let's download one latest copy of Parrot and install it on our machine.

Parrot download link is available in Parrot CVS Snapshot. Download the latest version of Parrot and to install it follow the following steps:

- Unzip and untar the downloaded file.
- Make sure you already have Perl 5 installed on your machine.
- Now do the following:

```
% cd parrot
% perl Configure.pl
Parrot Configure
Copyright (C) 2001 Yet Another Society
Since you're running this script, you obviously have
Perl 5 -- I'll be pulling some defaults from its configuration.
...
```

- You'll then be asked a series of questions about your local configuration; you can almost always hit return/enter for each one.
- Finally, you'll be told to type - make *test_prog,* and Parrot will successfully build the test interpreter.
- Now you should run some tests; so type 'make test' and you should see a readout like the following:

```
perl t/harness
t/op/basic.....ok,1/2 skipped:label constants unimplemented in
assembler
t/op/string....ok, 1/4 skipped:  I'm unable to write it!
All tests successful, 2 subtests skipped.
Files=2, Tests=6,......
```

By the time you read this, there could be more tests, and some of those which skipped might not skip, but make sure that none of them should fail!

Once you have a parrot executable installed, you can check out the various types of examples given in Parrot 'Examples' section. Also you can check out the examples directory in the parrot repository.

PARROT INSTRUCTIONS FORMAT

Parrot can currently accept instructions to execute in four forms. PIR *ParrotIntermediateRepresentation* is designed to be written by people and generated by compilers. It hides away some low-level details, such as the way parameters are passed to functions.

PASM *ParrotAssembly* is a level below PIR - it is still human readable/writable and can be generated by a compiler, but the author has to take care of details such as calling conventions and register allocation. PAST *ParrotAbstractSyntaxTree* enables Parrot to accept an abstract syntax tree style input - useful for those writing compilers.

All of the above forms of input are automatically converted inside Parrot to PBC *ParrotBytecode*. This is much like machine code, but understood by the Parrot interpreter.

It is not intended to be human-readable or human-writable, but unlike the other forms execution can start immediately without the need for an assembly phase. Parrot bytecode is platform independent.

The instruction set

The Parrot instruction set includes arithmetic and logical operators, compare and branch/jump *forimplementingloops, if. . . thenconstructs, etc.* , finding and storing global and lexical variables, working with classes and objects, calling subroutines and methods along with their parameters, I/O, threads and more.

GARBAGE COLLECTION IN PARROT

Like Java Virtual Machine, Parrot also keep you free from worrying about memory de-allocation.

- Parrot provides garbage collection.
- Parrot programs do not need to free memory explicitly.
- Allocated memory will be freed when it is no longer in use i.e. no longer referenced.
- Parrot Garbage Collector runs periodically to take care of unwanted memory.

PARROT DATATYPES

The Parrot CPU has four basic data types:

- **IV**

  An integer type; guaranteed to be wide enough to hold a pointer.

- **NV**

  An architecture-independent floating-point type.

- **STRING**

  An abstracted, encoding-independent string type.

- **PMC**

  A scalar.

The first three types are pretty much self-explanatory; the final type - Parrot Magic Cookies, are slightly more difficult to understand.

What are PMCs?

PMC stands for Parrot Magic Cookie. PMCs represent any complex data structure or type, including aggregate data types *arrays, hashtables, etc.* . A PMC can implement its own behavior for arithmetic, logical and string operations performed on it, allowing for language-specific behavior to be introduced. PMCs can be built in to the Parrot executable or dynamically loaded when they are needed.

PARROT REGISTERS

The current Perl 5 virtual machine is a stack machine. It communicate values between operations by keeping them on a stack. Operations load values onto the stack, do whatever they need to do and put the result back onto the stack. This is easy to work with, but it is slow.

To add two numbers together, you need to perform three stack pushes and two stack pops. Worse, the stack has to grow at runtime, and that means allocating memory just when you don't want to be allocating it.

So Parrot is going to break the established tradition for virtual machines, and use a register architecture, more akin to the architecture of a real hardware CPU. This has another advantage. We can use all the existing literature on how to write compilers and optimizers for register-based CPUs for our software CPU!

Parrot has specialist registers for each type: 32 IV registers, 32 NV registers, 32 string registers and 32 PMC registers. In Parrot assembler, these are named I1...I32, N1...N32, S1...S32, P1...P32 respectively.

Now let's look at some assembler. We can set these registers with the set operator:

```
 set I1, 10
 set N1, 3.1415
 set S1, "Hello, Parrot"
```

All Parrot ops have the same format: the name of the operator, the destination register and then the operands.

PARROT OPERATIONS

There are a variety of operations you can perform. For instance, we can print out the contents of a register or a constant:

```
set I1, 10
print "The contents of register I1 is: "
print I1
print "\n"
```

The above instructions will result in *The contents of register I1 is: 10*

We can perform mathematical operations on registers:

```
# Add the contents of I2 to the contents of I1
add I1, I1, I2
# Multiply I2 by I4 and store in I3
mul I3, I2, I4
# Increment I1 by one
inc I1
# Decrement N3 by 1.5
dec N3, 1.5
```

We can even perform some simple string manipulation:

```
set S1, "fish"
set S2, "bone"
concat S1, S2        # S1 is now "fishbone"
set S3, "w"
substr S4, S1, 1, 7
concat S3, S4        # S3 is now "wishbone"
length I1, S3        # I1 is now 8
```

PARROT BRANCHES

Code gets a little boring without flow control; for starters, Parrot knows about branching and labels. The branch op is equivalent to Perl's goto:

```
         branch TERRY
JOHN:    print "fjords\n"
         branch END
MICHAEL: print " pining"
         branch GRAHAM
TERRY:   print "It's"
         branch MICHAEL
GRAHAM:  print " for the "
         branch JOHN
END:     end
```

It can also perform simple tests to see whether a register contains a true value:

```
       set I1, 12
         set I2, 5
         mod I3, I2, I2
         if I3, REMAIND, DIVISOR
REMAIND: print "5 divides 12 with remainder "
         print I3
         branch DONE
DIVISOR: print "5 is an integer divisor of 12"
DONE:    print "\n"
         end
```

Here's what that would look like in Perl, for comparison:

```
    $i1 = 12;
    $i2 = 5;
    $i3 = $i1 % $i2;
    if ($i3) {
      print "5 divides 12 with remainder ";
      print $i3;
    } else {
      print "5 is an integer divisor of 12";
    }
    print "\n";
    exit;
```

Parrot Operator

We have the full range of numeric comparators: eq, ne, lt, gt, le and ge. Note that you can't use these operators on arguments of disparate types; you may even need to add the suffix _i or _n to the op, to tell it what type of argument you are using, although the assembler ought to divine this for you, by the time you read this.

PARROT PROGRAMMING EXAMPLES

Parrot programing is similar to assembly language programing and you get a chance to work at lower level. Here is the list of programming examples to make you aware of the various aspects of Parrot Programming.

- Classic Hello world!

Classic Hello world!

Create a file called hello.pir that contains the following code:

```
.sub _main
    print "Hello world!\n"
    end
.end
```

Then run it by typing:

```
parrot hello.pir
```

As expected, this will display the text 'Hello world!' on the console, followed by a new line *duetothe*\n.

In this above example, '.sub _main' states that the instructions that follow make up a subroutine named '_main', until a '.end' is encountered. The second line contains the print instruction. In this case, we are calling the variant of the instruction that accepts a constant string. The assembler takes care of deciding which variant of the instruction to use for us. The third line contains the 'end' instruction, which causes the interpreter to terminate.

Using Registers

We can modify hello.pir to first store the string Hello world!\n in a register and then use that register with the print instruction.

```
.sub _main
    set S1, "Hello world!\n"
    print S1
    end
.end
```

Here we have stated exactly which register to use. However, by replacing S1 with $S1 we can delegate the choice of which register to use to Parrot. It is also possible to use an = notation instead of writing the set instruction.

```
.sub _main
    $S0 = "Hello world!\n"
    print $S0
    end
.end
```

To make PIR even more readable, named registers can be used. These are later mapped to real numbered registers.

```
.sub _main
    .local string hello
    hello = "Hello world!\n"
    print hello
    end
.end
```

The '.local' directive indicates that the named register is only needed inside the current compilation unit *thatis, between. suband. end*. Following '.local' is a type. This can be int *forIregisters*, float *forNregisters*, string *forSregisters*, pmc *forPregisters* or the name of a PMC type.

Summing squares

This example introduces some more instructions and PIR syntax. Lines starting with a # are comments.

```
  .sub _main
      # State the number of squares to sum.
      .local int maxnum
      maxnum = 10

      # Some named registers we'll use.
      # Note how we can declare many
      # registers of the same type on one line.
      .local int i, total, temp
      total = 0

      # Loop to do the sum.
      i = 1
  loop:
      temp = i * i
      total += temp
      inc i
      if i <= maxnum goto loop

      # Output result.
      print "The sum of the first "
      print maxnum
      print " squares is "
      print total
      print ".\n"
      end
  .end
```

PIR provides a bit of syntactic sugar that makes it look more high level than assembly. For example:

```
  temp = i * i
```

Is just another way of writing the more assembly-ish:

```
  mul temp, i, i
```

And:

```
 if i <= maxnum goto loop
```

Is the same as:

```
le i, maxnum, loop
```

And:

```
  total += temp
```

Is the same as:

```
add total, temp
```

As a rule, whenever a Parrot instruction modifies the contents of a register, that will be the first register when writing the instruction in assembly form.

As is usual in assembly languages, loops and selections are implemented in terms of conditional branch statements and labels, as shown above. Assembly programming is one place where using goto is not a bad form!

Fibonacci Numbers

The Fibonacci series is defined like this: take two numbers, 1 and 1. Then repeatedly add together the last two numbers in the series to make the next one: 1, 1, 2, 3, 5, 8, 13, and so on. The Fibonacci number fib*n* is the n'th number in the series. Here's a simple Parrot assembler program that finds the first 20 Fibonacci numbers:

```
# Some simple code to print some Fibonacci numbers

        print   "The first 20 fibonacci numbers are:\n"
        set     I1, 0
        set     I2, 20
        set     I3, 1
        set     I4, 1
REDO:   eq      I1, I2, DONE, NEXT
NEXT:   set     I5, I4
        add     I4, I3, I4
        set     I3, I5
        print   I3
        print   "\n"
        inc     I1
        branch  REDO
DONE:   end
```

This is the equivalent code in Perl:

```
        print "The first 20 fibonacci numbers are:\n";
        my $i = 0;
        my $target = 20;
        my $a = 1;
        my $b = 1;
        until ($i == $target) {
           my $num = $b;
           $b += $a;
           $a = $num;
           print $a,"\n";
           $i++;
        }
```

NOTE: As a fine point of interest, one of the shortest and certainly the most beautiful ways of printing out a Fibonacci series in Perl is perl -le '*b* = 1; *print*a+=*bwhileprint*b+=$a'.

Recursively computing factorial

In this example we define a factorial function and recursively call it to compute factorial.

```
 .sub _fact
      # Get input parameter.
      .param int n

      # return (n > 1 ? n * _fact(n - 1) : 1)
      .local int result

      if n > 1 goto recurse
      result = 1
      goto return

  recurse:
      $I0 = n - 1
      result = _fact($I0)
      result *= n

  return:
      .return (result)
  .end


  .sub _main :main
      .local int f, i

      # We'll do factorial 0 to 10.
      i = 0
  loop:
      f = _fact(i)

      print "Factorial of "
      print i
      print " is "
      print f
      print ".\n"

      inc i
      if i <= 10 goto loop

      # That's it.
      end
  .end
```

Let's look at the _fact sub first. A point that was glossed over earlier is why the names of subroutines, all start with an underscore! This is done simply as a way of showing that the label is global rather than scoped to a particular subroutine. This is significant as the label is then visible to other subroutines.

The first line, .param int n, specifies that this subroutine takes one integer parameter and that we'd like to refer to the register it was passed in by the name n for the rest of the sub.

Much of what follows has been seen in previous examples, apart from the line reading:

```
result = _fact($I0)
```

This single line of PIR actually represents quite a few lines of PASM. First, the value in register $I0 is moved into the appropriate register for it to be received as an integer parameter by the _fact function. Other calling related registers are then set up, followed by _fact being invoked. Then, once _fact returns, the value returned by _fact is placed into the register given the name result.

Right before the .end of the _fact sub, a .return directive is used to ensure the value held in the register; named result is placed into the correct register for it to be seen as a return value by the code calling the sub.

The call to _fact in main works in just the same way as the recursive call to _fact within the sub _fact itself. The only remaining bit of new syntax is the :main, written after .sub _main. By default, PIR assumes that execution begins with the first sub in the file. This behavior can be changed by marking the sub to start in with :main.

Compiling to PBC

To compile PIR to bytecode, use the -o flag and specify an output file with the extension .pbc.

```
 parrot -o factorial.pbc factorial.pir
```

PIR vs. PASM

PIR can be turned into PASM by running:

```
parrot -o hello.pasm hello.pir
```

The PASM for the final example looks like this:

```
  _main:
      set S30, "Hello world!\n"
      print S30
      end
```

PASM does not handle register allocation or provide support for named registers. It also does not have the .sub and .end directives, instead replacing them with a label at the start of the instructions

Processing math: 100%