Contents

Chapter 1

Introduction to Parrot

DESCRIPTION

This document provides a gentle introduction to the Parrot virtual machine for anyone considering writing code for Parrot by hand, writing a compiler that targets Parrot, getting involved with Parrot development or simply wondering what on earth Parrot is.

WHAT IS PARROT?

Virtual Machines

Parrot is a virtual machine. To understand what a virtual machine is, consider what happens when you write a program in a language such as Perl, then run it with the applicable interpreter (in the case of Perl, the perl executable). First, the program you have written in a high level language is turned into simple instructions, for example *fetch the value of the variable named x*, *add 2 to this value*, *store this value in the variable named y*, etc. A single line of code in a high level language may be converted into tens of these simple instructions. This stage is called *compilation*.

The second stage involves executing these simple instructions. Some languages (for example, C) are often compiled to instructions that are understood by the CPU and as such can be executed by the hardware. Other languages, such as Perl, Python and Java, are usually compiled to CPU-independent instructions. A *virtual machine* (sometimes known as an *interpreter*) is required to execute those instructions.

While the central role of a virtual machine is to efficiently execute instructions, it also performs a number of other functions. One of these is to abstract away the details of the hardware and operating system that a program is running on. Once a program has been compiled to run on a virtual machine, it will run on any platform that the VM has been implemented

on. VMs may also provide security by allowing more fine-grained limitations to be placed on a program, memory management functionality and support for high level language features (such as objects, data structures, types, subroutines, etc).

Design goals

Parrot is designed with the needs of dynamically typed languages (such as Perl and Python) in mind, and should be able to run programs written in these languages more efficiently than VMs developed with static languages in mind (JVM, .NET). Parrot is also designed to provide interoperability between languages that compile to it. In theory, you will be able to write a class in Perl, subclass it in Python and then instantiate and use that subclass in a Tcl program.

Historically, Parrot started out as the runtime for Perl 6. Unlike Perl 5, the Perl 6 compiler and runtime (VM) are to be much more clearly separated. The name *Parrot* was chosen after the 2001 April Fool's Joke which had Perl and Python collaborating on the next version of their languages. The name reflects the intention to build a VM to run not just Perl 6, but also many other languages.

Parrot concepts and jargon

Instruction formats

Parrot can currently accept instructions to execute in four forms. PIR (Parrot Intermediate Representation) 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 (Parrot Assembly) 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 (Parrot Abstract Syntax Tree) 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 (Parrot Bytecode). 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 (for implementing loops, if...then constructs, 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.

Registers and fundamental data types

The Parrot VM is register based. This means that, like a hardware CPU, it has a number of fast-access units of storage called registers. There are 4 types of register in Parrot: integers (I), numbers (N), strings (S) and PMCs (P). There are N of each of these, named I0,I1,..N0.., etc. Integer registers are the same size as a word on the machine Parrot is running on and number registers also map to a native floating point type. The amount of registers needed is determined per subroutine at compile-time.

PMCs

PMC stands for Polymorphic Container. PMCs represent any complex data structure or type, including aggregate data types (arrays, hash tables, 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.

Garbage Collection

Parrot provides garbage collection, meaning that Parrot programs do not need to free memory explicitly; it will be freed when it is no longer in use (that is, no longer referenced) whenever the garbage collector runs.

Obtaining, building and testing Parrot

Where to get Parrot

See `http://www.parrot.org/download` for several ways to get a recent version of parrot.

Building Parrot

The first step to building Parrot is to run the *Configure.pl* program, which looks at your platform and decides how Parrot should be built. This is done by typing:

```
[commandchars=\\\{\}]
  perl Configure.pl
```

Once this is complete, run the `make` program `Configure.pl` prompts you with. When this completes, you will have a working `parrot` executable.

Please report any problems that you encounter while building Parrot so the developers can fix them. You can do this by creating a login and opening a new ticket at `https://github.com/parrot/parrot/issues/new`. Please include the *myconfig* file that was generated as part of the build process and any errors that you observed.

The Parrot test suite

Parrot has an extensive regression test suite. This can be run by typing:

```
[commandchars=\\\{\}]
  make test
```

Substituting make for the name of the make program on your platform. The output will look something like this:

```
[commandchars=\\\{\}]
 C:\textbackslash{}Perl\textbackslash{}bin\textbackslash{}perl.exe t\textbackslash{}harness --gc-debug
   t\textbackslash{}library\textbackslash{}*.t  t\textbackslash{}op\textbackslash{}*.t  t\textbackslash{}pmc\text
   imcc\textbackslash{}t\textbackslash{}*\textbackslash{}*.t  t\textbackslash{}dynpmc\textbackslash{}*.t  t\textb
 t\textbackslash{}library\textbackslash{}dumper.............ok
 t\textbackslash{}library\textbackslash{}getopt_long.........ok
 ...
 All tests successful, 4 test and 71 subtests skipped.
 Files=163, Tests=2719, 192 wallclock secs ( 0.00 cusr +  0.00 csys =  0.00 CPU)
```

It is possible that a number of tests may fail. If this is a small number, then it is probably little to worry about, especially if you have the latest Parrot sources from the Git repository. However, please do not let this discourage you from reporting test failures, using the same method as described for reporting build problems.

SOME SIMPLE PARROT PROGRAMS

Hello world!

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

```
[commandchars=\\\{\}]
  .sub main
      say "Hello world!"
  .end
```

Then run it by typing:

```
[commandchars=\\\{\}]
  parrot hello.pir
```

As expected, this will display the text `Hello world!` on the console, followed by a new line.

Let's take the program apart. `.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.

Using registers

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

```
[commandchars=\\\{\}]
  .sub main
      $S0 = "Hello world!"
      say $S0
  .end
```

PIR does not allow us to set a register directly. We need to prefix the register name with `$` when referring to a register. The compiler will map $S0 to one of the available string registers, for example S0, and set the value. This example also uses the syntactic sugar provided by the `=` operator. `=` is simply a more readable way of using the `set` opcode.

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

```
[commandchars=\\\{\}]
  .sub main
      .local string hello
      hello = "Hello world!"
      say hello
  .end
```

The `.local` directive indicates that the named register is only needed inside the current subroutine (that is, between `.sub` and `.end`). Following `.local` is a type. This can be `int` (for I registers), `float` (for N registers), `string` (for S registers), `pmc` (for P registers) or the name of a PMC type.

PIR vs. PASM

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.

Summing squares

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

```
[commandchars=\\\{\}]
  .sub main
      # State the number of squares to sum.
      .local int maxnum
      maxnum = 10

      # We'll use some named registers. Note that 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 ".\textbackslash{}n"
  .end
```

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

```
[commandchars=\\\{\}]
  .local pmc temp, i
  temp = i * i
```

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

```
[commandchars=\\\{\}]
  .local pmc temp, i
  mul temp, i, i
```

And:

```
[commandchars=\\\{\}]
  .local pmc i, maxnum
  if i <= maxnum goto loop
  # ...
  loop:
```

Is the same as:

```
[commandchars=\\\{\}]
  .local pmc i, maxnum
  le i, maxnum, loop
  # ...
  loop:
```

And:

```
[commandchars=\\\{\}]
  .local pmc temp, total
  total += temp
```

Is the same as:

```
[commandchars=\\\{\}]
  .local pmc  temp, total
  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 selection are implemented in terms of conditional branch statements and labels, as shown above. Assembly programming is one place where using goto is not bad form!

Recursively computing factorial

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

```
[commandchars=\\\{\}]
  .sub factorial
      # Get input parameter.
      .param int n

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

      if n > 1 goto recurse
      result = 1
      goto return

  recurse:
      $I0 = n - 1
      result = factorial($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 = factorial(i)

      print "Factorial of "
      print i
      print " is "
      print f
      print ".\textbackslash{}n"

      inc i
      if i <= 10 goto loop
  .end
```

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:

```
[commandchars=\\\{\}]
  .local int result
  result = factorial($I0)
```

The last line of PIR actually represents a few lines of PASM. The assembler builds a PMC that describes the signature, including which register the arguments are held in. A similar process happens for providing the registers that the return values should be placed in. Finally, the `factorial` sub is invoked.

Right before the `.end` of the `factorial` sub, a `.return` directive is used to specify that the value held in the register named `result` is to be copied to the register that the caller is expecting the return value in.

The call to `factorial` in main works in just the same was as the recursive call to `factorial` within the sub `factorial` 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*.

```
[commandchars=\\\{\}]
  parrot -o factorial.pbc factorial.pir
```

WHERE NEXT?

Documentation

What documentation you read next depends upon what you are looking to do with Parrot. The opcodes reference and built-in PMCs reference are useful to dip into for pretty much everyone. If you intend to write or compile to PIR then there are a number of documents about PIR that are worth a read. For compiler writers, the Compiler FAQ is essential reading. If you want to get involved with Parrot development, the PDDs (Parrot Design Documents) contain some details of the internals of Parrot; a few other documents fill in the gaps. One way of helping Parrot development is to write tests, and there is a document entitled *Testing Parrot* that will help with this.

The Parrot Mailing List

Much Parrot development and discussion takes place on the parrot-dev mailing list. You can subscribe by filling out the form at `http://lists.parrot.org/mailman/listinfo/parrot-dev` or read the NNTP archive at `http://groups.google.com/group/parrot-dev/`.

IRC

The Parrot IRC channel is hosted on `irc://irc.perl.org:6667` and is named `#parrot`. This Perl server network is also called `MagNET` or `Rhizomatic`.

Chapter 2

Overview

DESCRIPTION

This document is an introduction to the structure of and the concepts used by the Parrot shared bytecode compiler/interpreter system. We will primarily concern ourselves with the interpreter, since this is the target platform for which all compiler frontends should compile their code.

THE SOFTWARE CPU

Like all interpreter systems of its kind, the Parrot interpreter is a virtual machine; this is another way of saying that it is a software CPU. However, unlike other VMs, the Parrot interpreter is designed to more closely mirror hardware CPUs.

For instance, the Parrot VM will have a register architecture, rather than a stack architecture. It will also have extremely low-level operations, more similar to Java's than the medium-level ops of Perl and Python and the like.

The reasoning for this decision is primarily that by resembling the underlying hardware to some extent, it's possible to compile down Parrot bytecode to efficient native machine language.

Moreover, many programs in high-level languages consist of nested function and method calls, sometimes with lexical variables to hold intermediate results. Under non-JIT settings, a stack-based VM will be popping and then pushing the same operands many times, while a register-based VM will simply allocate the right amount of registers and operate on them, which can significantly reduce the amount of operations and CPU time.

To be more specific about the software CPU, it will contain a large number of registers. The current design provides for four groups of N registers; each group will hold a different data type: integers, floating-point numbers, strings, and PMCs. (Polymorphic Containers, detailed below.)

Registers will be stored in register frames, which can be pushed and popped onto the register stack. For instance, a subroutine or a block might need its own register frame.

THE OPERATIONS

The Parrot interpreter has a large number of very low level instructions, and it is expected that high-level languages will compile down to a medium-level language before outputting pure Parrot machine code.

Operations will be represented by several bytes of Parrot machine code; the first `INTVAL` will specify the operation number, and the remaining arguments will be operator-specific. Operations will usually be targeted at a specific data type and register type; so, for instance, the `dec_i_c` takes two `INTVAL`s as arguments, and decrements contents of the integer register designated by the first `INTVAL` by the value in the second `INTVAL`. Naturally, operations which act on `FLOATVAL` registers will use `FLOATVAL`s for constants; however, since the first argument is almost always a register **number** rather than actual data, even operations on string and PMC registers will take an `INTVAL` as the first argument.

As in Perl, Parrot ops will return the pointer to the next operation in the bytecode stream. Although ops will have a predetermined number and size of arguments, it's cheaper to have the individual ops skip over their arguments returning the next operation, rather than looking up in a table the number of bytes to skip over for a given opcode.

There will be global and private opcode tables; that is to say, an area of the bytecode can define a set of custom operations that it will use. These areas will roughly map to the subroutines of the original source; each precompiled module will have its own opcode table.

For a closer look at Parrot ops, see *docs/pdds/pdd06_pasm.pod*.

PMCs

PMCs are roughly equivalent to the `SV`, `AV` and `HV` (and more complex types) defined in Perl 5, and almost exactly equivalent to `PythonObject` types in Python. They are a completely abstracted data type; they may be string, integer, code or anything else. As we will see shortly, they can be expected to behave in certain ways when instructed to perform certain operations - such as incrementing by one, converting their value to an integer, and so on.

The fact of their abstraction allows us to treat PMCs as, roughly speaking, a standard API for dealing with data. If we're executing Perl code, we can manufacture PMCs that behave like Perl scalars, and the operations we perform on them will do Perlish things; if we execute Python code, we

can manufacture PMCs with Python operations, and the same underlying bytecode will now perform Pythonic activities.

For documentation on the specific PMCs that ship with Parrot, see the *docs/pmc* directory.

VTABLES

The way we achieve this abstraction is to assign to each PMC a set of function pointers that determine how it ought to behave when asked to do various things. In a sense, you can regard a PMC as an object in an abstract virtual class; the PMC needs a set of methods to be defined in order to respond to method calls. These sets of methods are called **vtables**.

A vtable is, more strictly speaking, a structure which expects to be filled with function pointers. The PMC contains a pointer to the vtable structure which implements its behavior. Hence, when we ask a PMC for its length, we're essentially calling the `length` method on the PMC; this is implemented by looking up the `length` slot in the vtable that the PMC points to, and calling the resulting function pointer with the PMC as argument: essentially,

```
[commandchars=\\\{\}]
    (pmc->vtable->length)(pmc);
```

If our PMC is a string and has a vtable which implements Perl-like string operations, this will return the length of the string. If, on the other hand, the PMC is an array, we might get back the number of elements in the array. (If that's what we want it to do.)

Similarly, if we call the increment operator on a Perl string, we should get the next string in alphabetic sequence; if we call it on a Python value, we may well get an error to the effect that Python doesn't have an increment operator suggesting a bug in the compiler front-end. Or it might use a "super-compatible Python vtable" doing the right thing anyway to allow sharing data between Python programs and other languages more easily.

At any rate, the point is that vtables allow us to separate out the basic operations common to all programming languages - addition, length, concatenation, and so on - from the specific behavior demanded by individual languages. Perl 6 will be Perl by passing Parrot a set of Perlish vtables; Parrot will equally be able to run Python, Tcl, Ruby or whatever by linking in a set of vtables which implement the behaviors of values in those languages. Combining this with the custom opcode tables mentioned above, you should be able to see how Parrot is essentially a language independent base for building runtimes for bytecompiled languages.

One interesting thing about vtables is that you can construct them dynamically. You can find out more about vtables in *docs/vtables.pod*.

STRING HANDLING

Parrot provides a programmer-friendly view of strings. The Parrot string handling subsection handles all the work of memory allocation, expansion, and so on behind the scenes. It also deals with some of the encoding headaches that can plague Unicode-aware languages.

This is done primarily by a similar vtable system to that used by PMCs; each encoding will specify functions such as the maximum number of bytes to allocate for a character, the length of a string in characters, the offset of a given character in a string, and so on. They will, of course, provide a transcoding function either to the other encodings or just to Unicode for use as a pivot.

The string handling API is explained in *docs/strings.pod*.

BYTECODE FORMAT

We have already explained the format of the main stream of bytecode; operations will be followed by arguments packed in such a format as the individual operations require. This makes up the third section of a Parrot bytecode file; frozen representations of Parrot programs have the following structure.

Firstly, a magic number is presented to identify the bytecode file as Parrot code. Next comes the fixup segment, which contains pointers to global variable storage and other memory locations required by the main opcode segment. On disk, the actual pointers will be zeroed out, and the bytecode loader will replace them by the memory addresses allocated by the running instance of the interpreter.

Similarly, the next segment defines all string and PMC constants used in the code. The loader will reconstruct these constants, fixing references to the constants in the opcode segment with the addresses of the newly reconstructed data.

As we know, the opcode segment is next. This is optionally followed by a code segment for debugging purposes, which contains a munged form of the original program file.

The bytecode format is fully documented in *docs/parrotbyte.pod*.

Chapter 3

Submitting bug reports and patches

docs/submissions.pod - Parrot Submission Instructions

DESCRIPTION

How to submit bug reports, patches and new files to Parrot.

HOW TO SUBMIT A BUG REPORT

If you encounter an error while working with Parrot and don't understand what is causing it, create a bug report using the *parrotbug* utility. The simplest way to use it is to run

```
[commandchars=\\\{\}]
    % ./parrotbug
```

in the distribution's root directory, and follow the prompts.

If you know how to fix the problem you encountered, then think about submitting a patch, or (see below) getting commit privileges.

A NOTE ON RANDOM FAILURES

If you encounter errors that appear intermittently, it may be difficult or impossible for Parrot developers to diagnose and solve the problem. It is therefore recommended to control the sources of randomness in Parrot in an attempt to eliminate the intermittency of the bug. There are three common sources of randomness that should be considered.

Pseudo-Random Number Generator

Direct use of a PRNG from within Parrot programs will lead to inconsistent results. If possible, isolate the bug from PRNG use, for

example, by logging the random values which trigger the error and then hard coding them.

Address Space Layout Randomization

Several operating systems provide a security measure known as address space layout randomization. In bugs involving stray pointers, this can cause corruption in random Parrot subsystems. Temporarily disabling this feature may make this problem consistent and therefore debugable.

Hash Seed

Parrot's hash implementation uses randomization of its seed as a precaution against attacks based on hash collisions. The seed used can be directly controlled using `parrot`'s `--hash-seed` parameter. To determine what seeds are causing the error, Parrot can be rebuilt with `DEBUG_HASH_SEED` set to `1`, which will cause `parrot` to output the hash seed being used on every invocation.

HOW TO CREATE A PATCH

Try to keep your patches specific to a single change, and ensure that your change does not break any tests. Do this by running `make test`. If there is no test for the fixed bug, please provide one.

In the following examples, *parrot* contains the Parrot distribution, and *workingdir* contains *parrot*. The name *workingdir* is just a placeholder for whatever the distribution's parent directory is called on your machine.

```
[commandchars=\\\{\}]
    workingdir
        |
        +--> parrot
                |
                +--> LICENSE
                |
                +--> src
                |
                +--> tools
                |
                +--> ...
```

`git`

If you are working with a git repository of parrot then please submit your patch as a pull request on github. You can find instructions at `http://help.github.com/send-pull-requests/`

Single `diff`

If you are working from a released distribution of Parrot and the change you wish to make affects only one or two files, then you can supply a `diff` for each file. The `diff` should be created in *parrot*. Please be sure to create a unified diff, with `diff -u`.

```
[commandchars=\\\{\}]
    cd parrot
    diff -u docs/submissions.pod docs/submissions.new > submissions.patch
```

Win32 users will probably need to specify `-ub`.

Recursive `diff`

If the change is more wide-ranging, then create an identical copy of *parrot* in *workingdir* and rename it *parrot.new*. Modify *parrot.new* and run a recursive `diff` on the two directories to create your patch. The `diff` should be created in *workingdir*.

```
[commandchars=\\\{\}]
    cd workingdir
    diff -ur --exclude='.git' parrot parrot.new > docs.patch
```

Mac OS X users should also specify `--exclude=.DS_Store`.

`CREDITS`

Each and every patch is an important contribution to Parrot and it's important that these efforts are recognized. To that end, the *CREDITS* file contains an informal list of contributors and their contributions made to Parrot. Patch submitters are encouraged to include a new or updated entry for themselves in *CREDITS* as part of their patch.

The format for entries in *CREDITS* is defined at the top of the file.

HOW TO SUBMIT A PATCH

The preferrred method to submit matches to Parrot is as pull requests via github. Please follow the instructions at `http://help.github.com/send-pull-requests/`.

APPLYING PATCHES

You may wish to apply a patch submitted by someone else before the patch is incorporated into git

For single `diff` patches or `git` patches, copy the patch file to *parrot*, and run:

```
[commandchars=\\\{\}]
    cd parrot
    git apply some.patch
```

For recursive `diff` patches, copy the patch file to *workingdir*, and run:

```
[commandchars=\\\{\}]
    cd workingdir
    git apply some.patch
```

In order to be on the safe side run 'make test' before actually committing the changes.

Configuration of files to ignore

Sometimes new files will be created in the configuration and build process of Parrot. These files should not show up when checking the distribution with

```
[commandchars=\\\{\}]
    git status
```

or

```
[commandchars=\\\{\}]
    perl tools/dev/manicheck.pl
```

In order to keep the two different checks synchronized, the MANIFEST and MANIFEST.SKIP file should be regenerated with:

```
[commandchars=\\\{\}]
    perl tools/dev/mk_manifest_and_skip.pl
```

WHAT HAPPENS NEXT?

If you created a new issue, you will be taken to the issue page and can check on the progress of discussion there. The issue number should be used in all out-of-band correspondence concerning the issue (e.g., in email to the `parrot-dev` mailing list). Otherwise, everyone on the parrot project can see the issue and can comment on it.

A developer with git commit privileges can merge your changes into the main parrot repository, once it is clear that this is the right thing to do. However your pull request may not be processed right away if the changes are large or complex, as we need time for peer review.

A list of open issues can be found here: `https://github.com/parrot/parrot/issues?state=open`

PATCHES FOR THE PARROT WEBSITE

The `http://www.parrot.org` website is hosted in a Drupal CMS. Submit changes through the usual ticket interface in Trac.

GETTING COMMIT PRIVILEGES

If you are interested in getting commit privileges to Parrot, here is the procedure:

1. Obtain a github account at `http://github.com`
2. Submit several high quality patches (and have them committed) via the process described in this document. This process may take weeks or months.

3. Submit a Parrot Contributor License Agreement; this document signifies that you have the authority to license your work to Parrot Foundation for inclusion in their projects. You may need to discuss this with your employer if you contribute to Parrot on work time or with work resources, or depending on your employment agreement.

   `http://www.parrot.org/f\unhbox\voidb@x\hbox{}iles/parrot_cla.pdf`

4. Request commit access via the `parrot-dev` mailing list, or via IRC (#parrot on irc.parrot.org). The existing committers will discuss your request in the next couple of weeks.

   If approved, a metacommitter will update the permissions to allow you to commit to Parrot; see `RESPONSIBLE_PARTIES` for the current list. Welcome aboard!

Thanks for your help!

Chapter 4

Parrot's command line options

```
[commandchars=\\\{\}]
    parrot -R, --runcore <CORE> -O<level> -D<flags> -d<flags> -t<flags>

    parrot -R fast
    parrot -R slow
    parrot -R trace | -t
    parrot -R profiling
    parrot -R subprof
    parrot --gc-debug
    parrot -R jit      I<(currently disabled)>
    parrot -R exec     I<(currently disabled)>
```

DESCRIPTION

This document describes Parrot's runcore, debugging and optimizer options.

ENVIRONMENT

PARROT_RUNTIME

If this environment variable is set, parrot will use this path as its runtime prefix instead of the compiled in path.

PARROT_GC_DEBUG

Turn on the *–gc-debug* flag.

OPTIONS

Assembler/compiler options

-O[level]

Valid optimizer levels: `-O`, `-O1`, `-O2`, `-Op`, `-Oc`

`-O1` enables the pre_optimizer, runs before control flow graph (CFG) is built. It includes strength reduction and rewrites certain if/branch/label constructs.

`-O2` runs afterwards, handles constant propagation, jump optimizations, removal of unused labels and dead code.

`-Op` applies `-O2` to pasm files also.

`-Oc` does tailcall optimizations.

The old options `-Oc` and `-Oj` are currently ineffective.

`-O` defaults to `-O1`.

–help-debug

Print debugging and tracing flag bits summary.

-y, –yydebug

Turn on yydebug in *yacc/bison*. Same as -d0004

-v, –verbose

Turn on compiler verbosity.

-d[=HEXFLAGS]

–**imcc-debug**[=HEXFLAGS]

Turn on compiler debug flags. *See* `parrot --help-debug` for available flag bits.

Runcore Options

These options select the runcore, which is useful for performance tuning and debugging. See ``ABOUTRUNCORES'' for details.

-R, –runcore CORE

Select the runcore. The following cores are available in Parrot, but not all may be available on your system:

```
[commandchars=\\\{\}]
  fast          bare-bones core without bounds-checking or
                context-updating (default)

  slow, bounds  bounds checking core

  trace         bounds checking core with trace info

  profiling     Rudimentary profiling support.
                See L<docs/dev/profiling.pod>

  subprof       Better subroutine-level profilers
  subprof_sub
  subprof_hll
  subprof_ops
                See POD in F<src/runcore/subprof.c>

  gc_debug      Does a full GC on each op.
```

Older currently ignored options include:

```
[commandchars=\\\{\}]
  jit, switch-jit, cgp-jit, switch, cgp, function, exec
```

We do not recommend their use in new code; they will continue working for existing code per our deprecation policy. The options function, cgp, switch, and jit, switch-jit, cgp-jit are currently aliases for fast.

The additional internal `debugger` runcore is used by debugger frontends.

See *src/runcore/cores.c* for details.

-p, –profile

Run with the slow core and print an execution profile.

-t, –trace

Run with the trace core and print trace information to **stderr**. See `parrot --help-debug` for available flag bits.

VM Options

-w, –warnings

Turn on warnings. See `parrot --help-debug` for available flag bits.

-D, –parrot-debug

Turn on interpreter debug flag. See `parrot --help-debug` for available flag bits.

–gc-debug

Turn on GC (Garbage Collection) debugging. This imposes some stress on the GC subsystem and can slow down execution considerably.

-G, –no-gc

This turns off GC. This may be useful to find GC related bugs. Don't use this option for longer running programs: as memory is no longer recycled, it may quickly become exhausted.

–gc-nursery-size=percent

Default: 2

–gc-dynamic-threshold=percent

Default: 75

–gc-min-threshold=MB

Default: 4

–leak-test, –destroy-at-end

Free all memory of the last interpreter. This is useful when running leak checkers.

–numthreads=number

Overrides the automatically detected number of CPU cores to set the number of OS threads. Minimum number: 2

ABOUT RUNCORES

The runcore (or runloop) tells Parrot how to find the C code that implements each instruction. Parrot provides more than one way to do this, partly because no single runcore will perform optimally on all architectures (or even for all problems on a given architecture), and partly because some of the runcores have specific debugging and tracing capabilities.

In the "slow" or "bounds" runcore, each opcode is a separate C function. That's pretty easy in pseudocode:

```
[commandchars=\\\{\}]
    slow_runcore( op ):
        while ( op ):
            op = op_function( op )
            check_for_events()
```

The old GC debugging runcore was similar:

```
[commandchars=\\\{\}]
    gcdebug_runcore( op ):
        while ( op ):
            perform_full_gc_run()
            op = op_function( op )
            check_for_events()
```

Of course, this is much slower, but is extremely helpful for pinning memory corruption problems that affect GC down to single-instruction resolution. See `http://www.oreillynet.com/onlamp/blog/2007/10/debugging_gc_problems_in_parro.html` for more information.

The trace and profile cores are also based on the "slow" core, doing full bounds checking, and also printing runtime information to stderr.

OPERATION TABLE

```
[commandchars=\\\{\}]
 Command Line          Action         Output
 ------------------------------------------
 parrot x.pir          run
 parrot x.pasm         run
 parrot x.pbc          run
 -o x.pasm x.pir       ass            x.pasm
 -o x.pasm y.pasm      ass            x.pasm
 -o x.pbc  x.pir       ass            x.pbc
 -o x.pbc  x.pasm      ass            x.pbc
```

```
 -o x.pbc -r x.pasm    ass/run pasm   x.pbc
 -o x.pbc -r -r x.pasm ass/run pbc    x.pbc
 -o x.o    x.pbc       obj
```

. . . where the possible actions are:

```
[commandchars=\\\{\}]
  run ... yes, run the program
  ass ... assemble sourcefile
  obj ..  produce native (ELF) object file for the EXEC subsystem
```

Chapter 5

PIR Guide

5.1 Introduction

Parrot is a language-neutral virtual machine for dynamic languages such as Ruby, Python, PHP, and Perl. It hosts a powerful suite of compiler tools tailored to dynamic languages and a next generation regular expression engine. Its architecture differs from virtual machines such as the JVM or CLR, with optimizations for dynamic languages, the use of registers instead of stacks, and pervasive continuations used for all flow control.

The name "Parrot" was inspired by Monty Python's Parrot sketch. As an April Fools' Day joke in 2001, Simon Cozens published "Programming Parrot", a fictional interview between Guido van Rossum and Larry Wall detailing their plans to merge Python and Perl into a new language called Parrot (`http://www.perl.com/pub/a/2001/04/01/parrot.htm`).

Parrot Intermediate Representation (PIR) is Parrot's native low-level language. PIR is fundamentally an assembly language, but it has some higher-level features such as operator syntax, syntactic sugar for subroutine and method calls, automatic register allocation, and more friendly conditional syntax. Parrot libraries—including most of Parrot's compiler tools—are often written in PIR. Even so, PIR is more rigid and "close to the machine" than some higher-level languages like C, which makes it a good window into the inner workings of the virtual machine.

Parrot Resources

The starting point for all things related to Parrot is the main website `http://www.parrot.org/`. The site lists additional resources, well as recent news and information about the project.

The Parrot Foundation holds the copyright over Parrot and helps support its development and community.

Documentation

Parrot includes extensive documentation in the distribution. The full documentation for the latest release is available online at `http://docs.parrot.org/`.

Mailing Lists

The primary mailing list for Parrot is *parrot-dev*.[1] If you're interested in developing Parrot, the *parrot-commits* and *parrot-tickets* lists are useful. More information on the Parrot mailing lists, as well as subscription options, is available on the mailing list info page `http://lists.parrot.org/mailman/listinfo`.

The archives for *parrot-dev* are available on Google Groups at `http://groups.google.com/group/parrot-dev` and as NNTP at `nntp://news.gmane.org/gmane.comp.compilers.parrot.devel`.

IRC

Parrot developers and users congregate on IRC at `#parrot` on the `irc://irc.parrot.org` server. It's a good place to ask questions or discuss Parrot in real time.

Issue Tracking & Wiki

Parrot developers track issues using the Github issues system at `https://github.com/parrot/parrot/issues/` Users can submit new tickets and track the status of existing tickets. Github also provides a wiki used in project development and a source code browser.

Parrot Development

Parrot's first release occurred in September 2001. The 1.0 release took place on March 17, 2009. The Parrot project makes releases on the third Tuesday of each month. Two releases a year — occurring every January and July — are "supported" releases intended for production use. The other ten releases are development releases intended for language implementers and testers.

Development proceeds in cycles around releases. Activity just before a release focuses on closing tickets, fixing bugs, reviewing documentation, and preparing for the release. Immediately after the release, larger changes occur: merging branches, adding large features, or removing deprecated features. This allows developers to ensure that changes have sufficient testing time before the next release. These regular releases also encourage feedback from casual users and testers.

[1] parrot-dev@lists.parrot.org

Licensing

The Parrot foundation supports the Parrot development community and holds trademarks and copyrights to Parrot. The project is available under the Artistic License 2.0, allowing free use in commercial and open source/free software contexts.

5.2 Getting Started

The simplest way to install Parrot is to use a pre-compiled binary for your operating system or distribution. Packages are available for many systems, including Debian, Ubuntu, Fedora, Mandriva, FreeBSD, Cygwin, and MacPorts. The Parrot website lists all known packages.[2] A binary installer for Windows is also available from the Parrot Win32 project on SourceForge.[3] If packages aren't available on your system, you can download a source tarball for the latest supported release from `http://www.parrot.org/release/supported`.

You need a C compiler and a make utility to build Parrot from source code—usually `gcc` and `make`, but Parrot can build with standard compiler toolchains on different operating systems. Perl 5.8 is also a prerequiste for configuring and building Parrot.

If you have these dependencies installed, build the core virtual machine and compiler toolkit and run the standard test suite with the commands:

```
[commandchars=\\\{\}]
  $ perl Configure.pl
  $ make
  $ make test
```

By default, Parrot installs to directories *bin/*, *lib/*, and *include/* under */usr/local*. If you have privileges to write to these directories, install Parrot with:

```
[commandchars=\\\{\}]
  $ make install
```

To install Parrot in a different location, use the `--prefix` option to *Configure.pl*:

```
[commandchars=\\\{\}]
    $ perl Configure.pl --prefix=/home/me/parrot
```

Setting the prefix to */home/me/parrot* installs the Parrot executable in */home/me/parrot/bin/parrot*.

If you intend to develop a language on Parrot, install the Parrot developer tools as well:

```
[commandchars=\\\{\}]
  $ make install-dev
```

[2] `http://www.parrot.org/download`

[3] `http://parrotwin32.sourceforge.net/`

Once you've installed Parrot, create a test file called *news.pir*.[4]

```
[commandchars=\\\{\}]
  .sub 'news'
    say "Here is the news for Parrots."
  .end
```

Now run this file with:

```
[commandchars=\\\{\}]
  $ parrot news.pir
```

which will print:

```
[commandchars=\\\{\}]
  Here is the news for Parrots.
```

5.3 Basic Syntax

PIR has a relatively simple syntax. Every line is a comment, a label, a statement, or a directive. Each statement or directive stands on its own line. There is no end-of-line symbol (such as a semicolon in C).

Comments

A comment begins with the `#` symbol, and continues until the end of the line. Comments can stand alone on a line or follow a statement or directive.

```
[commandchars=\\\{\}]
    # This is a regular comment. The PIR
    # interpreter ignores this.
```

PIR also treats inline documentation in Pod format as a comment. An equals sign as the first character of a line marks the start of a Pod block. A `=cut` marker signals the end of a Pod block.

```
[commandchars=\\\{\}]
  =head2

  This is Pod documentation, and is treated like a
  comment. The PIR interpreter ignores this.

  =cut
```

Labels

A label attaches a name to a line of code so other statements can refer to it. Labels can contain letters, numbers, and underscores. By convention, labels use all capital letters to stand out from the rest of the source code. It's fine to put a label on the same line as a statement or directive:

```
[commandchars=\\\{\}]
    GREET: say "'Allo, 'allo, 'allo."
```

Labels on separate lines improve readability, especially when outdented:

```
[commandchars=\\\{\}]
  GREET:
    say "'Allo, 'allo, 'allo."
```

[4] Files containing PIR code use the *.pir* extension.

Statements

A statement is either an opcode or syntactic sugar for one or more opcodes. An opcode is a native instruction for the virtual machine; it consists of the name of the instruction followed by zero or more arguments.

```
[commandchars=\\\{\}]
  say "Norwegian Blue"
```

PIR also provides higher-level constructs, including symbolic operators:

```
[commandchars=\\\{\}]
  $I1 = 2 + 5
```

These special statement forms are just syntactic sugar for regular opcodes. The `+` symbol corresponds to the `add` opcode, the `-` symbol to the `sub` opcode, and so on. The previous example is equivalent to:

```
[commandchars=\\\{\}]
  add $I1, 2, 5
```

Directives

Directives resemble opcodes, but they begin with a period (`.`). Some directives specify actions that occur at compile time. Other directives represent complex operations that require the generation of multiple instructions. The `.local` directive, for example, declares a named variable.

```
[commandchars=\\\{\}]
  .local string hello
```

Literals

Integers and floating point numbers are numeric literals. They can be positive or negative.

```
[commandchars=\\\{\}]
  $I0 = 42       # positive
  $I1 = -1       # negative
```

Integer literals can also be binary, octal, or hexadecimal:

```
[commandchars=\\\{\}]
  $I1 = 0b01010  # binary
  $I2 = 0o72     # octal
  $I3 = 0xA5     # hexadecimal
```

Floating point number literals have a decimal point and can use scientific notation:

```
[commandchars=\\\{\}]
  $N0 = 3.14
  $N2 = -1.2e+4
```

String literals are enclosed in single or double-quotes.[5]

```
[commandchars=\\\{\}]
  $S0 = "This is a valid literal string"
  $S1 = 'This is also a valid literal string'
```

[5] See the section on `Strings` in Chapter 4 for an explanation of the differences between the quoting types.

Variables

PIR variables can store four different kinds of values—integers, numbers (floating point), strings, and objects. Parrot's objects are called PMCs, for "*P*oly*M*orphic *C*ontainer".

The simplest kind of variable is a register variable. The name of a register variable always starts with a dollar sign (`$`), followed by a single character which specifies the type of the variable—integer (`I`), number (`N`), string (`S`), or PMC (`P`)—and ends with a unique number. You need not predeclare register variables:

```
[commandchars=\\\{\}]
  $S0 = "Who's a pretty boy, then?"
  say $S0
```

PIR also has named variables; the `.local` directive declares them. As with register variables, there are four valid types: `int`, `num`, `string`, and `pmc`. You *must* declare named variables; once declared, they behave exactly the same as register variables.

```
[commandchars=\\\{\}]
  .local string hello
  hello = "'Allo, 'allo, 'allo."
  say hello
```

Constants

The `.const` directive declares a named constant. Named constants are similar to named variables, but the values set in the declaration may never change. Like `.local`, `.const` takes a type and a name. It also requires a literal argument to set the value of the constant.

```
[commandchars=\\\{\}]
  .const int    frog = 4                      # integer
  .const string name = "Superintendent Parrot" # string
  .const num    pi   = 3.14159                # floating point
```

You may use a named constant anywhere you may use a literal, but you must declare the named constant beforehand. This example declares a named string constant `hello` and prints the value:

```
[commandchars=\\\{\}]
  .const string hello = "Hello, Polly."
  say hello
```

Keys

A key is a special kind of constant used for accessing elements in complex variables (such as an array). A key is either an integer or a string; and it's always enclosed in square brackets (`[` and `]`). You do not have to declare literal keys. This code example stores the string "foo" in $P0 as element 5, and then retrieves it.

```
[commandchars=\\\{\}]
  $P0[5] = "foo"
  $S1    = $P0[5]
```

PIR supports multi-part keys. Use a semicolon to separate each part.

```
[commandchars=\\\{\}]
  $P0['my';'key'] = 472
  $I1             = $P0['my';'key']
```

Control Structures

Rather than providing a pre-packaged set of control structures like `if` and `while`, PIR gives you the building blocks to construct your own.[6] The most basic of these building blocks is `goto`, which jumps to a named label.[7] In this code example, the `say` statement will run immediately after the `goto` statement:

```
[commandchars=\\\{\}]
    goto GREET
      # ... some skipped code ...
  GREET:
    say "'Allo, 'allo, 'allo."
```

Variations on the basic `goto` check whether a particular condition is true or false before jumping:

```
[commandchars=\\\{\}]
  if $I0 > 5 goto GREET
```

You can construct any traditional control structure from PIR's built-in control structures.

Subroutines

A PIR subroutine starts with the `.sub` directive and ends with the `.end` directive. Parameter declarations use the `.param` directive; they resemble named variable declarations. This example declares a subroutine named `greeting`, that takes a single string parameter named `hello`:

```
[commandchars=\\\{\}]
  .sub 'greeting'
      .param string hello
      say hello
  .end
```

That's All Folks

You now know everything you need to know about PIR. Everything else you read or learn about PIR will use one of these fundamental language structures. The rest is vocabulary.

[6] PIR has many advanced features, but at heart it **is** an assembly language.

[7] This is not your father's `goto`. It can only jump inside a subroutine, and only to a named label.

Parrot Assembly Language

Parrot Assembly Language (PASM) is another low-level language native to the virtual machine. PASM is a pure assembly language, with none of the syntactic sugar that makes PIR friendly for library development. PASM's primary purpose is to act as a plain English representation of the bytecode format. Its typical use is for debugging, rather than for writing libraries. Use PIR or a higher-level language for development tasks.
PASM files use the *.pasm* file extension.

5.4 Variables

Parrot is a register-based virtual machine. It has four typed register sets—integers, floating-point numbers, strings, and objects. All variables in PIR are one of these four types. When you work with register variables or named variables, you're actually working directly with register storage locations in the virtual machine.

If you've ever worked with an assembly language before, you may immediately jump to the conclusion that `$I0` is the zeroth integer register in the register set, but Parrot is a bit smarter than that. The number of a register variable does not necessarily correspond to the register used internally; Parrot's compiler maps registers as appropriate for speed and memory considerations. The only guarantee Parrot gives you is that you'll always get the same storage location when you use `$I0` in the same subroutine.

Assignment

The most basic operation on a variable is assignment using the `=` operator:

```
[commandchars=\\\{\}]
  $I0 = 42        # set integer variable to the value 42
  $N3 = 3.14159   # set number variable to approximation of pi
  $I1 = $I0       # set $I1 to the value of $I0
```

The `null` opcode sets an integer or number variable to a zero value, and undefines a string or object.

```
[commandchars=\\\{\}]
  null $I0  # 0
  null $N0  # 0.0
  null $S0  # NULL
  null $P0  # PMCNULL
```

Working with Numbers

PIR has an extensive set of instructions that work with integers, floating-point numbers, and numeric PMCs. Many of these instructions have a variant that modifies the result in place:

```
[commandchars=\\\{\}]
  $I0 = $I1 + $I2
  $I0 += $I1
```

The first form of `+` stores the sum of the two arguments in the result variable, `$I0`. The second variant, `+=`, adds the single argument to `$I0` and stores the sum back in `$I0`.

The arguments can be Parrot literals, variables, or constants. If the result is an integer type, like `$I0`, the arguments must also be integers. A number result, like `$N0`, usually requires number arguments, but many numeric instructions also allow the final argument to be an integer. Instructions with a PMC result may accept an integer, floating-point, or PMC final argument:

```
[commandchars=\\\{\}]
  $P0 = $P1 * $P2
  $P0 = $P1 * $I2
  $P0 = $P1 * $N2
  $P0 *= $P1
  $P0 *= $I1
  $P0 *= $N1
```

Unary numeric opcodes

Unary opcodes have a single argument. They either return a result or modify the argument in place. Some of the most common unary numeric opcodes are `inc` (increment), `dec` (decrement), `abs` (absolute value), `neg` (negate):

```
[commandchars=\\\{\}]
  $N0 = abs -5.0  # the absolute value of -5.0 is 5.0
  $I0 = 120
  inc $I1         # 120 incremented by 1 is 121
```

Binary numeric opcodes

Binary opcodes have two arguments and a result. Parrot provides addition (`+` or `add`), subtraction (`-` or `sub`), multiplication (`*` or `mul`), division (`/` or `div`), modulus (`%` or `mod`), and exponent (`pow`) opcodes, as well as `gcd` (greatest common divisor) and `lcm` (least common multiple).

```
[commandchars=\\\{\}]
  $I0 = 12 / 5
  $I0 = 12 % 5
```

Floating-point operations

The most common floating-point operations are `ln` (natural log), `log2` (log base 2), `log10` (log base 10), and `exp` (e^x), as well as a full set of trigonometric opcodes such as `sin` (sine), `cos` (cosine), `tan` (tangent), `sec` (secant), `sinh` (hyperbolic sine), `cosh` (hyperbolic cosine), `tanh` (hyperbolic tangent), `sech` (hyperbolic secant), `asin` (arc sine), `acos` (arc cosine), `atan` (arc tangent), `asec` (arc secant), `exsec` (exsecant), `hav` (haversine), and `vers` (versine). All angle arguments for the trigonometric opcodes are in radians:

```
[commandchars=\\\{\}]
  .loadlib 'trans_ops'

  # ...

  $N0 = sin $N1
  $N0 = exp 2
```

The majority of the floating-point operations have a single argument and a single result. The arguments can generally be either an integer or number, but many of these opcodes require the result to be a number.

Logical and Bitwise Operations

The logical opcodes evaluate the truth of their arguments. They are most useful to make decisions for control flow. Integers and numeric PMCs are false if they're 0 and true otherwise. Strings are false if they're the empty string or a single character "0", and true otherwise. PMCs are true when their `get_bool` vtable function returns a nonzero value.

The `and` opcode returns the first argument if it's false and the second argument otherwise:

```
[commandchars=\\\{\}]
  $I0 = and 0, 1  # returns 0
  $I0 = and 1, 2  # returns 2
```

The `or` opcode returns the first argument if it's true and the second argument otherwise:

```
[commandchars=\\\{\}]
  .loadlib 'bit_ops'

  # ...

  $I0 = or 1, 0  # returns 1
  $I0 = or 0, 2  # returns 2

  $P0 = or $P1, $P2
```

Both `and` and `or` are short-circuiting ops. If they can determine what value to return from the first argument, they'll never evaluate the second. This is significant only for PMCs, as they might have side effects on evaluation.

The `xor` opcode returns the first argument if it is the only true value, returns the second argument if it is the only true value, and returns false if both values are true or both are false:

```
[commandchars=\\\{\}]
  $I0 = xor 1, 0  # returns 1
  $I0 = xor 0, 1  # returns 1
  $I0 = xor 1, 1  # returns 0
  $I0 = xor 0, 0  # returns 0
```

The `not` opcode returns a true value when the argument is false and a false value if the argument is true:

```
[commandchars=\\\{\}]
  $I0 = not $I1
  $P0 = not $P1
```

The bitwise opcodes operate on their values a single bit at a time. `band`, `bor`, and `bxor` return a value that is the logical AND, OR, or XOR of each bit in the source arguments. They each take two arguments.

```
[commandchars=\\\{\}]
  .loadlib 'bit_ops'

  # ...

  $I0 = bor $I1, $I2
  $P0 = bxor $P1, $I2
```

`band`, `bor`, and `bxor` also have variants that modify the result in place.

```
[commandchars=\\\{\}]
  .loadlib 'bit_ops'

  # ...

  $I0 = band $I1
  $P0 = bor $P1
```

`bnot` is the logical NOT of each bit in the source argument.

```
[commandchars=\\\{\}]
  .loadlib 'bit_ops'

  # ...

  $I0 = bnot $I1
```

The logical and arithmetic shift operations shift their values by a specified number of bits:

```
[commandchars=\\\{\}]
  .loadlib 'bit_ops'

  # ...

  $I0 = shl $I1, $I2        # shift $I1 left by count $I2
  $I0 = shr $I1, $I2        # arithmetic shift right
  $P0 = lsr $P1, $P2        # logical shift right
```

Working with Strings

Parrot strings are buffers of variable-sized data. The most common use of strings is to store text data. Strings can also hold binary or other non-textual data, though this is rare.[8] Parrot strings are flexible and powerful, to handle the complexity of human-readable (and computer-representable) text data. String operations work with string literals, variables, and constants, and with string-like PMCs.

Escape Sequences

Strings in double-quotes allow escape sequences using backslashes. Strings in single-quotes only allow escapes for nested quotes:

```
[commandchars=\\\{\}]
  $S0 = "This string is \textbackslash{}n on two lines"
  $S0 = 'This is a \textbackslash{}n one-line string with a slash in it'
```

Table 4.1 shows the escape sequences Parrot supports in double-quoted strings.

Table 5.1: String Escapes

Escape	Meaning
\a	An ASCII alarm character
\b	An ASCII backspace character
\t	A tab
\n	A newline
\v	A vertical tab
\f	A form feed
\r	A carriage return
\e	An escape
\\	A backslash
\'	A quote
\x*NN*	A character represented by 1-2 hexadecimal digits
\x{*NNNNNNNN*}	A character represented by 1-8 hexadecimal digits
\o*NNN*	A character represented by 1-3 octal digits
\u*NNNN*	A character represented by 4 hexadecimal digits
\U*NNNNNNNN*	A character represented by 8 hexadecimal digits
\c*X*	A control character *X*

[8] In general, a custom PMC is more useful.

Heredocs

If you need more flexibility in defining a string, use a heredoc string literal. The `<<` operator starts a heredoc. The string terminator immediately follows. All text until the terminator is part of the string. The terminator must appear on its own line, must appear at the beginning of the line, and may not have any trailing whitespace.

```
[commandchars=\\\{\}]
  $S2 = <<"End_Token"
  This is a multi-line string literal. Notice that
  it doesn't use quotation marks.
  End_Token
```

Concatenating strings

Use the `.` operator to concatenate strings. The following example concatenates the string "cd" onto the string "ab" and stores the result in `$S1`.

```
[commandchars=\\\{\}]
  $S0 = "ab"
  $S1 = $S0 . "cd"  # concatenates $S0 with "cd"
  say $S1           # prints "abcd"
```

Concatenation has a `.=` variant to modify the result in place. In the next example, the `.=` operation appends "xy" onto the string "abcd" in `$S1`.

```
[commandchars=\\\{\}]
  $S1 .= "xy"       # appends "xy" to $S1
  say $S1           # prints "abcdxy"
```

Repeating strings

The `repeat` opcode repeats a string a specified number of times:

```
[commandchars=\\\{\}]
  $S0 = "a"
  $S1 = repeat $S0, 5
  say $S1              # prints "aaaaa"
```

In this example, `repeat` generates a new string with "a" repeated five times and stores it in `$S1`.

Length of a string

The `length` opcode returns the length of a string in characters. This won't be the same as the length in *bytes* for multibyte encoded strings:

```
[commandchars=\\\{\}]
  $S0 = "abcd"
  $I0 = length $S0               # the length is 4
  say $I0
```

`length` has no equivalent for PMC strings.

Substrings

The simplest version of the `substr` opcode takes three arguments: a source string, an offset position, and a length. It returns a substring of the original string, starting from the offset position (0 is the first character) and spanning the length:

```
[commandchars=\\\{\}]
  $S0 = substr "abcde", 1, 2        # $S0 is "bc"
```

This example extracts a two-character string from "abcde" at a one-character offset from the beginning of the string (starting with the second character). It generates a new string, "bc", in the destination register `$S0`.

When the offset position is negative, it counts backward from the end of the string. Thus an offset of -1 starts at the last character of the string.

`substr` no longer has a four-argument form, as in-place string operations have been removed. There is a `replace` operator which will perform the replacement and return a new_string without modifying the old_string. The arguments are new_string, old_string, offset, count and replacement_string. The old_string is copied to the new_string with the replacement_string inserted from offset replacing the content for count characters.

This example replaces the substring "bc" in `$S1` with the string "XYZ", and returns "aXYZde" in `$S0`, `$S1` is not changed:

```
[commandchars=\\\{\}]
  $S1 = "abcde"
  $S0 = replace $S1, 1, 2, "XYZ"
  say $S0                         # prints "aXYZde"
  say $S1                         # prints "abcde"
```

When the offset position in a `replace` is one character beyond the original string length, `replace` appends the replacement string just like the concatenation operator. If the replacement string is an empty string, the opcode removes the characters from the original string in the new string.

```
[commandchars=\\\{\}]
  $S1 = "abcde"
  $S1 = replace $S1, 1, 2, "XYZ"
  say $S1                         # prints "aXYZde"
```

Converting characters

The `chr` opcode takes an integer value and returns the corresponding character in the ASCII character set as a one-character string. The `ord` opcode takes a single character string and returns the integer value of the character at the first position in the string. The integer value of the character will differ depending on the current encoding of the string:

```
[commandchars=\\\{\}]
  $S0 = chr 65              # $S0 is "A"
  $I0 = ord $S0             # $I0 is 65, if $S0 is ASCII/UTF-8
```

`ord` has a two-argument variant that takes a character offset to select a single character from a multicharacter string. The offset must be within the length of the string:

```
[commandchars=\\\{\}]
  $I0 = ord "ABC", 2        # $I0 is 67
```

A negative offset counts backward from the end of the string, so -1 is the last character.

```
[commandchars=\\\{\}]
  $I0 = ord "ABC", -1       # $I0 is 67
```

Formatting strings

The `sprintf` opcode generates a formatted string from a series of values. It takes two arguments: a string specifying the format, and an array PMC containing the values to be formatted. The format string and the result can be either strings or PMCs:

```
[commandchars=\\\{\}]
  $S0 = sprintf $S1, $P2
  $P0 = sprintf $P1, $P2
```

The format string is similar to C's `sprintf` function with extensions for Parrot data types. Each format field in the string starts with a `%` and ends with a character specifying the output format. Table 4.2 lists the available output format characters.

Each format field supports several specifier options: *flags*, *width*, *precision*, and *size*. Table 4.3 lists the format flags.

The *width* is a number defining the minimum width of the output from a field. The *precision* is the maximum width for strings or integers, and the number of decimal places for floating-point fields. If either *width* or *precision* is an asterisk (`*`), it takes its value from the next argument in the PMC.

The *size* modifier defines the type of the argument the field takes. Table 4.4 lists the size flags. The values in the aggregate PMC must have a type compatible with the specified *size*.

```
[commandchars=\\\{\}]
  $S0 = sprintf "int %#Px num %+2.3Pf\textbackslash{}n", $P2
  say $S0       # prints "int 0x2a num +10.000"
```

The format string of this `sprintf` example has two format fields. The first, `%#Px`, extracts a PMC argument (`P`) from the aggregate `$P2` and formats it as a hexadecimal integer (`x`) with a leading 0x (`#`). The second format field, `%+2.3Pf`, takes a PMC argument (`P`) and formats it as a floating-point number (`f`) with a minimum of two whole digits and a maximum of three decimal places (`2.3`) and a leading sign (`+`).

The test files *t/op/string.t* and *t/op/sprintf.t* have many more examples of format strings.

Table 5.2: Format characters

Format	Meaning
%c	A single character.
%d	A decimal integer.
%i	A decimal integer.
%u	An unsigned integer.
%o	An octal integer.
%x	A hex integer, preceded by 0x (when # is specified).
%X	A hex integer with a capital X (when # is specified).
%b	A binary integer, preceded by 0b (when # is specified).
%B	A binary integer with a capital B (when # is specified).
%p	A pointer address in hex.
%f	A floating-point number.
%e	A floating-point number in scientific notation (displayed with a lowercase "e").
%E	The same as %e, but displayed with an uppercase E.
%g	The same as %e or %f, whichever fits best.
%G	The same as %g, but displayed with an uppercase E.
%s	A string.

Table 5.3: Format flags

Flag	Meaning
0	Pad with zeros.
¡space¿	Pad with spaces.
+	Prefix numbers with a sign.
-	Align left.
#	Prefix a leading 0 for octal, 0x for hex, or force a decimal point.

Joining strings

The `join` opcode joins the elements of an array PMC into a single string. The first argument separates the individual elements of the PMC in the final string result.

```
[commandchars=\\\{\}]
  $P0 = new "ResizablePMCArray"
  push $P0, "hi"
  push $P0, 0
  push $P0, 1
  push $P0, 0
  push $P0, "parrot"
```

Table 5.4: Size flags

Character	Meaning
h	short integer or single-precision float
l	long
H	huge value (long long or long double)
v	Parrot INTVAL or FLOATVAL
O	opcode_t pointer
P	PMC
S	String

```
  $S0 = join "__", $P0
  say $S0                # prints "hi__0__1__0__parrot"
```

This example builds a Array in $P0 with the values ‘‘hi’’, 0, 1, 0, and ‘‘parrot’’. It then joins those values (separated by the string ‘‘__’’) into a single string stored in $S0.

Splitting strings

Splitting a string yields a new array containing the resulting substrings of the original string.

This example splits the string “abc” into individual characters and stores them in an array in $P0. It then prints out the first and third elements of the array.

```
[commandchars=\\\{\}]
  $P0 = split "", "abc"
  $P1 = $P0[0]
  say $P1                # 'a'
  $P1 = $P0[2]
  say $P1                # 'c'
```

Testing for substrings

The index opcode searches for a substring within a string. If it finds the substring, it returns the position where the substring was found as a character offset from the beginning of the string. If it fails to find the substring, it returns -1:

```
[commandchars=\\\{\}]
  $I0 = index "Beeblebrox", "eb"
  say $I0                          # prints 2
  $I0 = index "Beeblebrox", "Ford"
  say $I0                          # prints -1
```

index also has a three-argument version, where the final argument defines an offset position for starting the search.

```
[commandchars=\\\{\}]
  $I0 = index "Beeblebrox", "eb", 3
  say $I0                          # prints 5
```

This example finds the second "eb" in "Beeblebrox" instead of the first, because the search skips the first three characters in the string.

Bitwise Operations

The numeric bitwise opcodes also have string variants for AND, OR, and XOR: `bors`, `bands`, and `bxors`. These take string or string-like PMC arguments and perform the logical operation on each byte of the strings to produce the result string. Remember that in-place string operations are no longer available.

```
[commandchars=\\\{\}]
  .loadlib 'bit_ops'

  # ...

  $P0 = bors $P1
  $P0 = bands $P1
  $S0 = bors $S1, $S2
  $P0 = bxors $P1, $S2
```

The bitwise string opcodes produce meaningful results only when used with simple ASCII strings, because Parrot performs bitwise operations per byte.

Copy-On-Write

Strings use copy-on-write (COW) optimizations. A call to `$S1 = $S0` doesn't immediately make a copy of `$S0`, it only makes both variables point to the same string. Parrot doesn't make a copy of the string until one of two strings is modified.

```
[commandchars=\\\{\}]
  $S0 = "Ford"
  $S1 = $S0
  $S1 = "Zaphod"
  say $S0                # prints "Ford"
  say $S1                # prints "Zaphod"
```

Modifying one of the two variables causes Parrot to create a new string. This example preserves the existing value in `$S0` and assigns the new value to the new string in `$S1`. The benefit of copy-on-write is avoiding the cost of copying strings until the copies are necessary.

Encodings and Charsets

Years ago, strings only needed to support the ASCII character set (or charset), a mapping of 128 bit patterns to symbols and English-language characters. This worked as long as everyone using a computer read and wrote English and only used a small handful of punctuation symbols. In

other words, it was woefully insufficient. A modern string system must manage charsets in order to make sense out of all the string data in the world. A modern string system must also handle different encodings—ways to represent various charsets in memory and on disk.

Every string in Parrot has an associated encoding and character set. The default format is 8-bit ASCII, which is almost universally supported. Double-quoted string constants can have an optional prefix specifying the string's format.[9] Parrot tracks information about encoding and charset internally, and automatically converts strings when necessary to preserve these characteristics. Strings constants may have prefixes of the form `format:`.

```
[commandchars=\\\{\}]
  $S0 = utf8:"Hello UTF-8 Unicode World!"
  $S1 = utf16:"Hello UTF-16 Unicode World!"
  $S2 = ascii:"This is 8-bit ASCII"
  $S3 = binary:"This is raw, unformatted binary data"
```

Parrot supports the formats `ascii`, `binary`, `iso-8859-1` (Latin 1), `utf8`, `utf16`, `ucs2`, and `ucs4`.

The `binary` format treats the string as a buffer of raw unformatted binary data. It isn't really a string per se, because binary data contains no readable characters. This exists to support libraries which manipulate binary data that doesn't easily fit into any other primitive data type.

When Parrot operates on two strings (as in concatenation or comparison), they must both use the same character set and encoding. Parrot will automatically upgrade one or both of the strings to the next highest compatible format as necessary. ASCII strings will automatically upgrade to UTF-8 strings if needed, and UTF-8 will upgrade to UTF-16. All of these conversions happen inside Parrot, so the programmer doesn't need to worry about the details.

Working with PMCs

Polymorphic Containers (PMCs) are the basis for complex data types and object-oriented behavior in Parrot. In PIR, any variable that isn't a low-level integer, number, or string is a PMC. PMC variables act much like the low-level variables, but you have to instantiate a new PMC object before you use it. The `new` opcode creates a new PMC object of the specified type.

```
[commandchars=\\\{\}]
  $P0 = new 'String'
  $P0 = "That's a bollard and not a parrot"
  say $P0
```

This example creates a `String` object, stores it in the PMC register variable `$P0`, assigns it the value "That's a bollard and not a parrot", and prints it.

Every PMC has a type that indicates what data it can store and what behavior it supports. The `typeof` opcode reports the type of a PMC. When the result is a string variable, `typeof` returns the name of the type:

[9] As you might suspect, single-quoted strings do not support this.

```
[commandchars=\\\{\}]
  $P0 = new "String"
  $S0 = typeof $P0              # $S0 is "String"
  say $S0                       # prints "String"
```

When the result is a PMC variable, `typeof` returns the `Class` PMC for that object type.

Scalars

In most of the examples shown so far, PMCs duplicate the behavior of integers, numbers, and strings. Parrot provides a set of PMCs for this exact purpose. `Integer`, `Float`, and `String` are thin overlays on Parrot's low-level integers, numbers, and strings.

A previous example showed a string literal assigned to a PMC variable of type `String`. Direct assignment of a literal to a PMC works for all the low-level types and their PMC equivalents:

```
[commandchars=\\\{\}]
  $P0 = new 'Integer'
  $P0 = 5

  $P1 = new 'String'
  $P1 = "5 birds"

  $P2 = new 'Float'
  $P2 = 3.14
```

You may also assign non-constant low-level integer, number, or string registers directly to a PMC. The PMC handles the conversion from the low-level type to its own internal storage.[10]

```
[commandchars=\\\{\}]
  $I0 = 5
  $P0 = new 'Integer'
  $P0 = $I0

  $S1 = "5 birds"
  $P1 = new 'String'
  $P1 = $S1

  $N2 = 3.14
  $P2 = new 'Float'
  $P2 = $N2
```

The `box` opcode is a handy shortcut to create the appropriate PMC object from an integer, number, or string literal or variable.

```
[commandchars=\\\{\}]
  $P0 = box 3    # $P0 is an "Integer"

  $P1 = box $S1  # $P1 is a "String"

  $P2 = box 3.14 # $P2 is a "Float"
```

[10] This conversion of a simpler type to a more complex type is "boxing".

In the reverse situation, when assigning a PMC to an integer, number, or string variable, the PMC also has the ability to convert its value to the low-level type.[11]

```
[commandchars=\\\{\}]
  $P0 = box 5
  $S0 = $P0           # the string "5"
  $N0 = $P0           # the number 5.0
  $I0 = $P0           # the integer 5

  $P1 = box "5 birds"
  $S1 = $P1           # the string "5 birds"
  $I1 = $P1           # the integer 5
  $N1 = $P1           # the number 5.0

  $P2 = box 3.14
  $S2 = $P2           # the string "3.14"
  $I2 = $P2           # the integer 3
  $N2 = $P2           # the number 3.14
```

This example creates `Integer`, `Float`, and `String` PMCs, and shows the effect of assigning each one back to a low-level type.

Converting a string to an integer or number only makes sense when the contents of the string are a number. The `String` PMC will attempt to extract a number from the beginning of the string, but otherwise will return a false value.

Type Conversions

Parrot also handles conversions between the low-level types where possible, converting integers to strings (`$S0 = $I1`), numbers to strings (`$S0 = $N1`), numbers to integers (`$I0 = $N1`), integers to numbers (`$N0 = $I1`), and even strings to integers or numbers (`$I0 = $S1` and `$N0 = $S1`).

Aggregates

PMCs can define complex types that hold multiple values, commonly called aggregates. Two basic aggregate types are ordered arrays and associative arrays. The primary difference between these is that ordered arrays use integer keys for indexes and associative arrays use string keys.

Aggregate PMCs support the use of numeric or string keys. PIR also offers a extensive set of operations for manipulating aggregate data types.

[11] The reverse of "boxing" is "unboxing".

Ordered Arrays

Parrot provides several ordered array PMCs, differentiated by whether the array should store booleans, integers, numbers, strings, or other PMCs, and whether the array should maintain a fixed size or dynamically resize for the number of elements it stores.

The core array types are `FixedPMCArray`, `ResizablePMCArray`, `FixedIntegerArray`, `ResizableIntegerArray`, `FixedFloatArray`, `ResizableFloatArray`, `FixedStringArray`, `ResizableStringArray`, `FixedBooleanArray`, and `ResizableBooleanArray`. The array types that start with "Fixed" have a fixed size and do not allow elements to be added outside their allocated size. The "Resizable" variants automatically extend themselves as more elements are added.[12] The array types that include "String", "Integer", or "Boolean" in the name use alternate packing methods for greater memory efficiency.

Parrot's core ordered array PMCs all have zero-based integer keys. Extracting or inserting an element into the array uses PIR's standard key syntax, with the key in square brackets after the variable name. An lvalue key sets the value for that key. An rvalue key extracts the value for that key in the aggregate to use as the argument value:

```
[commandchars=\\\{\}]
  $P0    = new "ResizablePMCArray" # create a new array object
  $P0[0] = 10                      # set first element to 10
  $P0[1] = $I31                    # set second element to $I31
  $I0    = $P0[0]                  # get the first element
```

Setting the array to an integer value directly (without a key) sets the number of elements of the array. Assigning an array directly to an integer retrieves the number of elements of the array.

```
[commandchars=\\\{\}]
  $P0 = 2    # set array size
  $I1 = $P0  # get array size
```

This is equivalent to using the `elements` opcode to retrieve the number of items currently in an array:

```
[commandchars=\\\{\}]
  elements $I0, $P0 # get element count
```

Some other useful instructions for working with ordered arrays are `push`, `pop`, `shift`, and `unshift`, to add or remove elements. `push` and `pop` work on the end of the array, the highest numbered index. `shift` and `unshift` work on the start of the array, adding or removing the zeroth element, and renumbering all the following elements.

```
[commandchars=\\\{\}]
  push $P0, 'banana' # add to end
  $S0 = pop $P0      # fetch from end

  unshift $P0, 74    # add to start
  $I0 = shift $P0    # fetch from start
```

[12] With some additional overhead for checking array bounds and reallocating array memory.

Associative Arrays

An associative array is an unordered aggregate that uses string keys to identify elements. You may know them as "hash tables", "hashes", "maps", or "dictionaries". Parrot provides one core associative array PMC, called `Hash`. String keys work very much like integer keys. An lvalue key sets the value of an element, and an rvalue key extracts the value of an element. The string in the key must always be in single or double quotes.

```
[commandchars=\\\{\}]
  new $P1, "Hash"          # create a new associative array
  $P1["key"] = 10          # set key and value
  $I0        = $P1["key"]  # get value for key
```

Assigning a `Hash` PMC (without a key) to an integer result fetches the number of elements in the hash.[13]

```
[commandchars=\\\{\}]
  $I1 = $P1         # number of entries
```

The `exists` opcode tests whether a keyed value exists in an aggregate. It returns 1 if it finds the key in the aggregate and 0 otherwise. It doesn't care if the value itself is true or false, only that an entry exists for that key:

```
[commandchars=\\\{\}]
  new $P0, "Hash"
  $P0["key"] = 0
  exists $I0, $P0["key"] # does a value exist at "key"?
  say $I0                # prints 1
```

The `delete` opcode removes an element from an associative array:

```
[commandchars=\\\{\}]
  delete $P0["key"]
```

Iterators

An iterator extracts values from an aggregate PMC one at a time. Iterators are most useful in loops which perform an action on every element in an aggregate. The `iter` opcode creates a new iterator from an aggregate PMC. It takes one argument, the PMC over which to iterate:

```
[commandchars=\\\{\}]
  $P1 = iter $P2
```

The `shift` opcode extracts the next value from the iterator.

```
[commandchars=\\\{\}]
      $P5 = shift $P1
```

Evaluating the iterator PMC as a boolean returns whether the iterator has reached the end of the aggregate:

```
[commandchars=\\\{\}]
      if $P1 goto iter_repeat
```

[13] You may not set a `Hash` PMC directly to an integer value.

Parrot provides predefined constants for working with iterators. `.ITERATE_FROM_START` and `.ITERATE_FROM_END` constants select whether an ordered array iterator starts from the beginning or end of the array. These two constants have no effect on associative array iterators, as their elements are unordered.

Load the iterator constants with the `.include` directive to include the file *iterator.pasm*. To use them, set the iterator PMC to the value of the constant:

```
[commandchars=\\\{\}]
      .include "iterator.pasm"

      # ...

      $P1 = .ITERATE_FROM_START
```

With all of those separate pieces in one place, this example loads the iterator constants, creates an ordered array of "a", "b", "c", creates an iterator from that array, and then loops over the iterator using a conditional `goto` to checks the boolean value of the iterator and another unconditional `goto`:

```
[commandchars=\\\{\}]
      .include "iterator.pasm"
      $P2 = new "ResizablePMCArray"
      push $P2, "a"
      push $P2, "b"
      push $P2, "c"

      $P1 = iter $P2
      $P1 = .ITERATE_FROM_START

  iter_loop:
      unless $P1 goto iter_end
      $P5 = shift $P1
      say $P5                       # prints "a", "b", "c"
      goto iter_loop
  iter_end:
```

Associative array iterators work similarly to ordered array iterators. When iterating over associative arrays, the `shift` opcode extracts keys instead of values. The key looks up the value in the original hash PMC.

```
[commandchars=\\\{\}]
      $P2      = new "Hash"
      $P2["a"] = 10
      $P2["b"] = 20
      $P2["c"] = 30

      $P1      = iter $P2

  iter_loop:
      unless $P1 goto iter_end
      $S5 = shift $P1          # the key "a", "b", or "c"
      $I9 = $P2[$S5]           # the value 10, 20, or 30
      say $I9
      goto iter_loop
  iter_end:
```

This example creates an associative array `$P2` that contains three keys "a", "b", and "c", assigning them the values 10, 20, and 30. It creates an iterator

($P1) from the associative array using the `iter` opcode, and then starts a loop over the iterator. At the start of each loop, the `unless` instruction checks whether the iterator has any more elements. If there are no more elements, `goto` jumps to the end of the loop, marked by the label `iter_end`. If there are more elements, the `shift` opcode extracts the next key. Keyed assignment stores the integer value of the element indexed by the key in `$I9`. After printing the integer value, `goto` jumps back to the start of the loop, marked by `iter_loop`.

Multi-level Keys

Aggregates can hold any data type, including other aggregates. Accessing elements deep within nested data structures is a common operation, so PIR provides a way to do it in a single instruction. Complex keys specify a series of nested data structures, with each individual key separated by a semicolon.

```
[commandchars=\\\{\}]
  $P0           = new "Hash"
  $P1           = new "ResizablePMCArray"
  $P1[2]        = 42
  $P0["answer"] = $P1

  $I1 = 2
  $I0 = $P0["answer";$I1]
  say $I0
```

This example builds up a data structure of an associative array containing an ordered array. The complex key `["answer''; $I1]` retrieves an element of the array within the hash. You can also set a value using a complex key:

```
[commandchars=\\\{\}]
  $P0["answer";0] = 5
```

The individual keys are integer or string literals, or variables with integer or string values.

Copying and Cloning

PMC registers don't directly store the data for a PMC, they only store a pointer to the structure that stores the data. As a result, the `=` operator doesn't copy the entire PMC, it only copies the pointer to the PMC data. If you later modify the copy of the variable, it will also modify the original.

```
[commandchars=\\\{\}]
  $P0 = new "String"
  $P0 = "Ford"
  $P1 = $P0
  $P1 = "Zaphod"
  say $P0               # prints "Zaphod"
  say $P1               # prints "Zaphod"
```

In this example, `$P0` and `$P1` are both pointers to the same internal data structure. Setting `$P1` to the string literal "Zaphod", it overwrites the previous value "Ford". Both `$P0` and `$P1` refer to the `String` PMC "Zaphod".

The `clone` opcode makes a deep copy of a PMC, instead of copying the pointer like `=` does.

```
[commandchars=\\\{\}]
  $P0 = new "String"
  $P0 = "Ford"
  $P1 = clone $P0
  $P0 = "Zaphod"
  say $P0        # prints "Zaphod"
  say $P1        # prints "Ford"
```

This example creates an identical, independent clone of the PMC in `$P0` and puts it in `$P1`. Later changes to `$P0` have no effect on the PMC in `$P1`.[14]

To assign the *value* of one PMC to another PMC that already exists, use the `assign` opcode:

```
[commandchars=\\\{\}]
  $P0 = new "Integer"
  $P1 = new "Integer"
  $P0 = 42
  assign $P1, $P0    # note: $P1 must exist already
  inc $P0
  say $P0            # prints 43
  say $P1            # prints 42
```

This example creates two `Integer` PMCs, `$P1` and `$P2`, and gives the first one the value 42. It then uses `assign` to pass the same integer value on to `$P1`. Though `$P0` increments, `$P1` doesn't change. The result for `assign` must have an existing object of the right type in it, because `assign` neither creates a new duplicate object (as does `clone`) or reuses the source object (as does `=`).

Properties

PMCs can have additional values attached to them as "properties" of the PMC. Most properties hold extra metadata about the PMC.

The `setprop` opcode sets the value of a named property on a PMC. It takes three arguments: the PMC on which to set a property, the name of the property, and a PMC containing the value of the property.

```
[commandchars=\\\{\}]
  setprop $P0, "name", $P1
```

The `getprop` opcode returns the value of a property. It takes two arguments: the name of the property and the PMC from which to retrieve the property value.

```
[commandchars=\\\{\}]
  $P2 = getprop $P0, "name"
```

This example creates a `String` object in `$P0` and an `Integer` object with the value 1 in `$P1`. `setprop` sets a property named "eric" on the object in `$P0` and gives the property the value of `$P1`. `getprop` retrieves the value of the property "eric" on `$P0` and stores it in `$P2`.

[14] With low-level strings, the copies created by `clone` are copy-on-write exactly the same as the copy created by `=`.

```
[commandchars=\\\{\}]
  $P0 = new "String"
  $P0 = "Half-a-Bee"
  $P1 = new "Integer"
  $P1 = 1

  setprop $P0, "eric", $P1  # set a property on $P0
  $P2 = getprop $P0, "eric" # retrieve a property from $P0

  say $P2                   # prints 1
```

Parrot stores PMC properties in an associative array where the name of the property is the key.

`delprop` deletes a property from a PMC.

```
[commandchars=\\\{\}]
  delprop $P1, "constant" # delete property
```

You can fetch a complete hash of all properties on a PMC with `prophash`:

```
[commandchars=\\\{\}]
  $P0 = prophash $P1 # set $P0 to the property hash of $P1
```

Fetching the value of a non-existent property returns an `Undef` PMC.

Vtable Functions

You may have noticed that a simple operation sometimes has a different effect on different PMCs. Assigning a low-level integer value to a `Integer` PMC sets its integer value of the PMC, but assigning that same integer to an ordered array sets the size of the array.

Every PMC defines a standard set of low-level operations called vtable functions. When you perform an assignment like:

```
[commandchars=\\\{\}]
   $P0 = 5
```

. . . Parrot calls the `set_integer_native` vtable function on the PMC referred to by register `$P0`.

Parrot has a fixed set of vtable functions, so that any PMC can stand in for any other PMC; they're polymorphic.[15] Every PMC defines some behavior for every vtable function. The default behavior is to throw an exception reporting that the PMC doesn't implement that vtable function. The full set of vtable functions for a PMC defines the PMC's basic interface, but PMCs may also define methods to extend their behavior beyond the vtable set.

[15] Hence the name "Polymorphic Container".

Namespaces

Parrot performs operations on variables stored in small register sets local to each subroutine. For more complex tasks,[16] it's also useful to have variables that live beyond the scope of a single subroutine. These variables may be global to the entire program or restricted to a particular library. Parrot stores long-lived variables in a hierarchy of namespaces.

The opcodes `set_global` and `get_global` store and fetch a variable in a namespace:

```
[commandchars=\\\{\}]
  $P0 = new "String"
  $P0 = "buzz, buzz"
  set_global "bee", $P0
  # ...
  $P1 = get_global "bee"
  say $P1                        # prints "buzz, buzz"
```

The first two statements in this example create a `String` PMC in `$P0` and assign it a value. In the third statement, `set_global` stores that PMC as the named global variable `bee`. At some later point in the program, `get_global` retrieves the global variable by name, and stores it in `$P1` to print.

Namespaces can only store PMC variables. Parrot boxes all primitive integer, number, or string values into the corresponding PMCs before storing them in a namespace.

The name of every variable stored in a particular namespace must be unique. You can't have store both an `Integer` PMC and an array PMC both named "bee", stored in the same namespace.[17]

Namespace Hierarchy

A single global namespace would be far too limiting for most languages or applications. The risk of accidental collisions—where two libraries try to use the same name for some variable—would be quite high for larger code bases. Parrot maintains a collection of namespaces arranged as a tree, with the `parrot` namespace as the root. Every namespace you declare is a child of the `parrot` namespace (or a child of a child. . . .).

The `set_global` and `get_global` opcodes both have alternate forms that take a key name to access a variable in a particular namespace within the tree. This code example stores a variable as `bill` in the Duck namespace and retrieves it again:

```
[commandchars=\\\{\}]
  set_global ["Duck"], "bill", $P0
  $P1 = get_global ["Duck"], "bill"
```

[16] . . . and for most high-level languages that Parrot supports.

[17] You may wonder why anyone would want to do this. We wonder the same thing, but Perl 5 does it all the time. The Perl 6 implementation on Parrot includes type sigils in the names of the variables it stores in namespaces so each name is unique, e.g. `$bee`, `@bee`. . . .

The key name for the namespace can have multiple levels, which correspond to levels in the namespace hierarchy. This example stores a variable as `bill` in the Electric namespace under the General namespace in the hierarchy.

```
[commandchars=\\\{\}]
  set_global ["General";"Electric"], "bill", $P0
  $P1 = get_global ["General";"Electric"], "bill"
```

The `set_global` and `get_global` opcode operate on the currently selected namespace. The default top-level namespace is the "root" namespace. The `.namespace` directive allows you to declare any namespace for subsequent code. If you select the General Electric namespace, then store or retrieve the `bill` variable without specifying a namespace, you will work with the General Electric bill, not the Duck bill.

```
[commandchars=\\\{\}]
  .namespace ["General";"Electric"]
  #...
  set_global "bill", $P0
  $P1 = get_global "bill"
```

Passing an empty key to the `.namespace` directive resets the selected namespace to the root namespace. The brackets are required even when the key is empty.

```
[commandchars=\\\{\}]
  .namespace [ ]
```

When you need to be absolutely sure you're working with the root namespace regardless of what namespace is currently active, use the `set_root_global` and `get_root_global` opcodes instead of `set_global` and `get_global`. This example sets and retrieves the variable `bill` in the Dollar namespace, which is directly under the root namespace:

```
[commandchars=\\\{\}]
  set_root_global ["Dollar"], "bill", $P0
  $P1 = get_root_global ["Dollar"], "bill"
```

To prevent further collisions, each high-level language running on Parrot operates within its own virtual namespace root. The default virtual root is `parrot`, and the `.HLL` directive (for *H*igh-*L*evel *L*anguage) selects an alternate virtual root for a particular high-level language:

```
[commandchars=\\\{\}]
  .HLL 'ruby'
```

The `set_hll_global` and `get_hll_global` opcodes are like `set_root_global` and `get_root_global`, except they always operate on the virtual root for the currently selected HLL. This example stores and retrieves a `bill` variable in the Euro namespace, under the Dutch HLL namespace root:

```
[commandchars=\\\{\}]
  .HLL 'Dutch'
  #...
  set_hll_global ["Euro"], "bill", $P0
  $P1 = get_hll_global ["Euro"], "bill"
```

NameSpace PMC

Namespaces are just PMCs. They implement the standard vtable functions and a few extra methods. The `get_namespace` opcode retrieves the currently selected namespace as a PMC object:

```
[commandchars=\\\{\}]
  $P0 = get_namespace
```

The `get_root_namespace` opcode retrieves the namespace object for the root namespace. The `get_hll_namespace` opcode retrieves the virtual root for the currently selected HLL.

```
[commandchars=\\\{\}]
  $P0 = get_root_namespace
  $P0 = get_hll_namespace
```

Each of these three opcodes can take a key argument to retrieve a namespace under the currently selected namespace, root namespace, or HLL root namespace:

```
[commandchars=\\\{\}]
  $P0 = get_namespace ["Duck"]
  $P0 = get_root_namespace ["General";"Electric"]
  $P0 = get_hll_namespace ["Euro"]
```

Once you have a namespace object you can use it to retrieve variables from the namespace instead of using a keyed lookup. This example first looks up the Euro namespace in the currently selected HLL, then retrieves the `bill` variable from that namespace:

```
[commandchars=\\\{\}]
  $P0 = get_hll_namespace ["Euro"]
  $P1 = get_global $P0, "bill"
```

Namespaces also provide a set of methods to provide more complex behavior than the standard vtable functions allow. The `get_name` method returns the name of the namespace as a `ResizableStringArray`:

```
[commandchars=\\\{\}]
  $P3 = $P0.'get_name'()
```

The `get_parent` method retrieves a namespace object for the parent namespace that contains this one:

```
[commandchars=\\\{\}]
  $P5 = $P0.'get_parent'()
```

The `get_class` method retrieves any Class PMC associated with the namespace:

```
[commandchars=\\\{\}]
  $P6 = $P0.'get_class'()
```

The `add_var` and `find_var` methods store and retrieve variables in a namespace in a language-neutral way:

```
[commandchars=\\\{\}]
  $P0.'add_var'("bee", $P3)
  $P1 = $P0.'find_var'("bee")
```

The `find_namespace` method looks up a namespace, just like the `get_namespace` opcode:

```
[commandchars=\\\{\}]
  $P1 = $P0.'find_namespace'("Duck")
```

The `add_namespace` method adds a new namespace as a child of the namespace object:

```
[commandchars=\\\{\}]
  $P0.'add_namespace'($P1)
```

The `make_namespace` method looks up a namespace as a child of the namespace object and returns it. If the requested namespace doesn't exist, `make_namespace` creates a new one and adds it under that name:

```
[commandchars=\\\{\}]
  $P1 = $P0.'make_namespace'("Duck")
```

Aliasing

Just like regular assignment, the various operations to store a variable in a namespace only store a pointer to the PMC. If you modify the local PMC after storing in a namespace, those changes will also appear in the stored global. To store a true copy of the PMC, `clone` it before you store it.

Leaving the global variable as an alias for a local variable has its advantages. If you retrieve a stored global into a register and modify it:

```
[commandchars=\\\{\}]
  $P1 = get_global "feather"
  inc $P1
```

...you modify the value of the stored global, so you don't need to call `set_global` again.

5.5 Control Structures

The semantics of control structures in high-level languages vary broadly. Rather than dictating one particular set of semantics for control structures, or attempting to provide multiple implementations of common control structures to fit the semantics of all major target languages, PIR provides a simple set of conditional and unconditional branch instructions.[18]

[18] In fact, all control structures in all languages ultimately compile down to conditional and unconditional branches, so you're just getting a peek into the inner workings of your software.

Conditionals and Unconditionals

An unconditional branch always jumps to a specified label. PIR has only one unconditional branch instruction, `goto`. In this example, the first `say` statement never runs because the `goto` always skips over it to the label `skip_all_that`:

```
[commandchars=\\\{\}]
      goto skip_all_that
      say "never printed"

  skip_all_that:
      say "after branch"
```

A conditional branch jumps to a specified label only when a particular condition is true. The condition may be as simple as checking the truth of a particular variable or as complex as a comparison operation.

In this example, the `if/goto` skips to the label `maybe_skip` only if the value stored in `$I0` is true. If `$I0` is false, it will print "might be printed" and then print "after branch":

```
[commandchars=\\\{\}]
      if $I0 goto maybe_skip
      say "might be printed"
  maybe_skip:
      say "after branch"
```

Boolean Truth

Parrot's `if` and `unless` instructions evaluate a variable as a boolean to decide whether to jump. In PIR, an integer is false if it's 0 and true if it's any non-zero value. A number is false if it's 0.0 and true otherwise. A string is false if it's the empty string (`''`) or a string containing only a zero (`'0'`), and true otherwise. Evaluating a PMC as a boolean calls the vtable function `get_bool` to check if it's true or false, so each PMC is free to determine what its boolean value should be.

Comparisons

In addition to a simple check for the truth of a variable, PIR provides a collection of comparison operations for conditional branches. These jump when the comparison is true.

This example compares `$I0` to `$I1` and jumps to the label `success` if `$I0` is less than `$I1`:

```
[commandchars=\\\{\}]
      if $I0 < $I1 goto success
      say "comparison false"
  success:
      say "comparison true"
```

The full set of comparison operators in PIR are `==` (equal), `!=` (not equal), `<` (less than), `<=` (less than or equal), `>` (greater than), and `>=` (greater than or equal).

Complex Conditions

PIR disallows nested expressions. You cannot embed a statement within another statement. If you have a more complex condition than a simple truth test or comparison, you must build up your condition with a series of instructions that produce a final, single truth value.

This example performs two operations, addition and multiplication, then uses `and` to check if the results of both operations were true. The `and` opcode stores a boolean value (0 or 1) in the integer variable `$I2`; the code uses this value in an ordinary truth test:

```
[commandchars=\\\{\}]
    $I0 = 4 + 5
    $I1 = 63 * 0
    $I2 = and $I0, $I1

    if $I2 goto true
    say "maybe printed"
  true:
```

If/Else Construct

`if control structure` High-level languages often use the keywords *if* and *else* for simple conditional control structures. These control structures perform an action when a condition is true and skip the action when the condition is false. PIR's `if` instruction can build up simple conditionals.

This example checks the truth of the condition `$I0`. If `$I0` is true, it jumps to the `do_it` label, and runs the body of the conditional construct. If `$I0` is false, it continues on to the next statement, a `goto` instruction that skips over the body of the conditional to the label `dont_do_it`:

```
[commandchars=\\\{\}]
    if $I0 goto do_it
    goto dont_do_it
  do_it:
    say "in the body of the if"
  dont_do_it:
```

The control flow of this example may seem backwards. In a high-level language, *if* often means *"if the condition is true, run the next few lines of code"*. In an assembly language, it's often more straightforward to write *"if the condition is true, **skip** the next few lines of code"*. Because of the reversed logic, you may find it easier to build a simple conditional construct using the `unless` instruction instead of `if`.

```
[commandchars=\\\{\}]
    unless $I0 goto dont_do_it
    say "in the body of the if"
  dont_do_it:
```

This example produces the same output as the previous example, but the logic is simpler. When `$I0` is true, `unless` does nothing and the body of

the conditional runs. When `$I0` is false, `unless` skips over the body of the conditional by jumping to `dont_do_it`.

`else control structure` An *if/else* control structure is easier to build using the `if` instruction than `unless`. To build an *if/else*, insert the body of the else right after the first `if` instruction.

This example checks if `$I0` is true. If so, it jumps to the label `true` and runs the body of the *if* construct. If `$I0` is false, the `if` instruction does nothing, and the code continues to the body of the *else* construct. When the body of the else has finished, the `goto` jumps to the end of the *if/else* control structure by skipping over the body of the *if* construct:

```
[commandchars=\\\{\}]
    if $I0 goto true
    say "in the body of the else"
    goto done
  true:
    say "in the body of the if"
  done:
```

Switch Construct

A *switch* control structure selects one action from a list of possible actions by comparing a single variable to a series of values until it finds one that matches. The simplest way to achieve this in PIR is with a series of `unless` instructions:

```
[commandchars=\\\{\}]
    $S0 = 'a'

  option1:
    unless $S0 == 'a' goto option2
    say "matched: a"
    goto end_of_switch

  option2:
    unless $S0 == 'b' goto default
    say "matched: b"
    goto end_of_switch

  default:
    say "I don't understand"

  end_of_switch:
```

This example uses `$S0` as the *case* of the switch construct. It compares that case against the first value `a`. If they match, it prints the string "matched: a", then jumps to the end of the switch at the label `end_of_switch`. If the first case doesn't match `a`, the `goto` jumps to the label `option2` to check the second option. The second option compares the case against the value `b`. If they match, it prints the string "matched: b", then jumps to the end of the switch. If the case doesn't match the second option, the `goto` goes on to the default case, prints "I don't understand", and continues to the end of the switch.

Do-While Loop

A *do-while* loop runs the body of the loop once, then checks a condition at the end to decide whether to repeat it. A single conditional branch can build this style of loop:

```
[commandchars=\\\{\}]
    $I0 = 0                 # counter

  redo:                     # start of loop
    inc $I0
    say $I0
    if $I0 < 10 goto redo   # end of loop
```

This example prints the numbers 1 to 10. The first time through, it executes all statements up to the `if` instruction. If the condition evaluates as true (`$I0` is less than 10), it jumps to the `redo` label and runs the loop body again. The loop ends when the condition evaluates as false.

Here's a slightly more complex example that calculates the factorial `5!`:

```
[commandchars=\\\{\}]
      .local int product, counter

      product = 1
      counter = 5

  redo:                         # start of loop
      product *= counter
      dec counter
      if counter > 0 goto redo  # end of loop

      say product
```

Each time through the loop it multiplies `product` by the current value of the `counter`, decrements the counter, and jumps to the start of the loop. The loop ends when `counter` has counted down to 0.

While Loop

A *while* loop tests the condition at the start of the loop instead of at the end. This style of loop needs a conditional branch combined with an unconditional branch. This example also calculates a factorial, but with a *while* loop:

```
[commandchars=\\\{\}]
      .local int product, counter
      product = 1
      counter = 5

  redo:                         # start of loop
      if counter <= 0 goto end_loop
      product *= counter
      dec counter
      goto redo
  end_loop:                     # end of loop

      say product
```

This code tests the counter `counter` at the start of the loop to see if it's less than or equal to 0, then multiplies the current product by the counter and decrements the counter. At the end of the loop, it unconditionally jumps back to the start of the loop and tests the condition again. The loop ends when the counter `counter` reaches 0 and the `if` jumps to the `end_loop` label. If the counter is a negative number or zero before the loop starts the first time, the body of the loop will never execute.

For Loop

A *for* loop is a counter-controlled loop with three declared components: a starting value, a condition to determine when to stop, and an operation to step the counter to the next iteration. A *for* loop in C looks something like:

```
[commandchars=\\\{\}]
  for (i = 1; i <= 10; i++) \{
    ...
  \}
```

where `i` is the counter, `i = 1` sets the start value, `i <= 10` checks the stop condition, and `i++` steps to the next iteration. A *for* loop in PIR requires one conditional branch and two unconditional branches.

```
[commandchars=\\\{\}]
  loop_init:
    .local int counter
    counter = 1

  loop_test:
    if counter <= 10 goto loop_body
    goto loop_end

  loop_body:
    say counter

  loop_continue:
    inc counter
    goto loop_test

  loop_end:
```

The first time through the loop, this example sets the initial value of the counter in `loop_init`. It then goes on to test that the loop condition is met in `loop_test`. If the condition is true (`counter` is less than or equal to 10) it jumps to `loop_body` and executes the body of the loop. If the condition is false, it will jump straight to `loop_end` and the loop will end. The body of the loop prints the current counter then goes on to `loop_continue`, which increments the counter and jumps back up to `loop_test` to continue on to the next iteration. Each iteration through the loop tests the condition and increments the counter, ending the loop when the condition is false. If the condition is false on the very first iteration, the body of the loop will never run.

5.6 Subroutines

Subroutines in PIR are roughly equivalent to the subroutines or methods of a high-level language. They're the most basic building block of code reuse in PIR. Each high-level language has different syntax and semantics for defining and calling subroutines, so Parrot's subroutines need to be flexible enough to handle a broad array of behaviors.

A subroutine declaration starts with the `.sub` directive and ends with the `.end` directive. This example defines a subroutine named `hello` that prints a string "Hello, Polly.":

```
[commandchars=\\\{\}]
  .sub 'hello'
      say "Hello, Polly."
  .end
```

The quotes around the subroutine name are optional as long as the name of the subroutine uses only plain alphanumeric ASCII characters. You must use quotes if the subroutine name uses Unicode characters, characters from some other character set or encoding, or is otherwise an invalid PIR identifier.

A subroutine call consists of the name of the subroutine to call followed by a list of (zero or more) arguments in parentheses. You may precede the call with a list of (zero or more) return values. This example calls the subroutine `fact` with two arguments and assigns the result to `$I0`:

```
[commandchars=\\\{\}]
  $I0 = 'fact'(count, product)
```

Modifiers

A modifier is an annotation to a basic subroutine declaration[19] that selects an optional feature. Modifiers all start with a colon (`:`). A subroutine can have multiple modifiers.

When you execute a PIR file as a program, Parrot normally runs the first subroutine it encounters, but you can mark any subroutine as the first one to run with the `:main` modifier:

```
[commandchars=\\\{\}]
  .sub 'first'
      say "Polly want a cracker?"
  .end

  .sub 'second' :main
      say "Hello, Polly."
  .end
```

This code prints "Hello, Polly." but not "Polly want a cracker?". The `first` subroutine is first in the source code, but `second` has the `:main` modifier. Parrot will never call `first` in this program. If you remove the `:main` modifier, the code will print "Polly want a cracker?" instead.

[19] or parameter declaration

The `:load` modifier tells Parrot to run the subroutine when it loads the current file as a library. The `:init` modifier tells Parrot to run the subroutine only when it executes the file as a program (and *not* as a library). The `:immediate` modifier tells Parrot to run the subroutine as soon as it gets compiled. The `:postcomp` modifier also runs the subroutine right after compilation, but only if the subroutine was declared in the main program file (when *not* loaded as a library).

By default, Parrot stores all subroutines in the namespace currently active at the point of their declaration. The `:anon` modifier tells Parrot not to store the subroutine in the namespace. The `:nsentry` modifier stores the subroutine in the currently active namespace with a different name. For example, Parrot will store this subroutine in the current namespace as `bar`, not `foo`:

```
[commandchars=\\\{\}]
  .sub 'foo' :nsentry('bar')
    #...
  .end
```

Chapter 7 on *"Classes and Objects"* explains other subroutine modifiers.

Parameters and Arguments

The `.param` directive defines the parameters for the subroutine and creates local named variables for them (similar to `.local`):

```
[commandchars=\\\{\}]
  .param int c
```

The `.return` directive returns control flow to the calling subroutine. To return results, pass them as arguments to `.return`.

```
[commandchars=\\\{\}]
  .return($P0)
```

This example implements the factorial algorithm using two subroutines, `main` and `fact`:

```
[commandchars=\\\{\}]
  # factorial.pir
  .sub 'main' :main
     .local int count
     .local int product
     count   = 5
     product = 1

     $I0 = 'fact'(count, product)

     say $I0
  .end

  .sub 'fact'
     .param int c
     .param int p

  loop:
```

```
    if c <= 1 goto fin
    p = c * p
    dec c
    branch loop
  fin:
    .return (p)
  .end
```

This example defines two local named variables, `count` and `product`, and assigns them the values 1 and 5. It calls the `fact` subroutine with both variables as arguments. The `fact` subroutine uses the `.param` directive to retrieve these parameters and the `.return` directive to return the result. The final printed result is 120.

Positional Parameters

The default way of matching the arguments passed in a subroutine call to the parameters defined in the subroutine's declaration is by position. If you declare three parameters—an integer, a number, and a string:

```
[commandchars=\\\{\}]
  .sub 'foo'
    .param int a
    .param num b
    .param string c
    # ...
  .end
```

. . . then calls to this subroutine must also pass three arguments—an integer, a number, and a string:

```
[commandchars=\\\{\}]
  'foo'(32, 5.9, "bar")
```

Parrot will assign each argument to the corresponding parameter in order from first to last. Changing the order of the arguments or leaving one out is an error.

Named Parameters

Named parameters are an alternative to positional parameters. Instead of passing parameters by their position in the string, Parrot assigns arguments to parameters by their name. Consequently you may pass named parameters in any order. Declare named parameters with the `:named` modifier.

This example declares two named parameters in the subroutine `shoutout`—`name` and `years`—each declared with the `:named` modifier and followed by the name to use when pass arguments. The string name can match the parameter name (as with the `name` parameter), but it can also be different (as with the `years` parameter):

```
[commandchars=\\\{\}]
 .sub 'shoutout'
    .param string name :named("name")
    .param string years :named("age")
```

```
    $S0  = "Hello " . name
    $S1  = "You are " . years
    $S1 .= " years old"
    say $S0
    say $S1
 .end
```

Pass named arguments to a subroutine as a series of name/value pairs, with the elements of each pair separated by an arrow =>.

```
[commandchars=\\\{\}]
 .sub 'main' :main
    'shoutout'("age" => 42, "name" => "Bob")
 .end
```

The order of the arguments does not matter:

```
[commandchars=\\\{\}]
 .sub 'main' :main
    'shoutout'("name" => "Bob", "age" => 42)
 .end
```

Optional Parameters

Another alternative to the required positional parameters is optional parameters. Some parameters are unnecessary for certain calls. Parameters marked with the `:optional` modifier do not produce errors about invalid parameter counts if they are not present. A subroutine with optional parameters should gracefully handle the missing argument, either by providing a default value or by performing an alternate action that doesn't need that value.

Checking the value of the optional parameter isn't enough to know whether the call passed such an argument, because the user might have passed a null or false value intentionally. PIR also provides an `:opt_flag` modifier for a boolean check whether the caller passed an argument:

```
[commandchars=\\\{\}]
  .param string name     :optional
  .param int    has_name :opt_flag
```

When an integer parameter with the `:opt_flag` modifier immediately follows an `:optional` parameter, it will be true if the caller passed the argument and false otherwise.

This example demonstrates how to provide a default value for an optional parameter:

```
[commandchars=\\\{\}]
    .param string name     :optional
    .param int    has_name :opt_flag

    if has_name goto we_have_a_name
    name = "default value"
  we_have_a_name:
```

When the `has_name` parameter is true, the `if` control statement jumps to the `we_have_a_name` label, leaving the `name` parameter unmodified. When `has_name` is false (when the caller passed no argument for `name`) the `if` statement does nothing. The next line sets the `name` parameter to a default value.

The `:opt_flag` parameter never takes an argument from the passed-in argument list. It's purely for bookkeeping within the subroutine.

Optional parameters can be positional or named parameters. Optional parameters must appear at the end of the list of positional parameters after all the required parameters. An optional parameter must immediately precede its `:opt_flag` parameter whether it's named or positional:

```
[commandchars=\\\{\}]
  .sub 'question'
    .param int value     :named("answer") :optional
    .param int has_value :opt_flag
    #...
  .end
```

You can call this subroutine with a named argument or with no argument:

```
[commandchars=\\\{\}]
  'question'("answer" => 42)
  'question'()
```

Aggregating Parameters

Another alternative to a sequence of positional parameters is an aggregating parameter which bundles a list of arguments into a single parameter. The `:slurpy` modifier creates a single array parameter containing all the provided arguments:

```
[commandchars=\\\{\}]
  .param pmc args :slurpy
  $P0 = args[0]           # first argument
  $P1 = args[1]           # second argument
```

As an aggregating parameter will consume all subsequent parameters, you may use an aggregating parameter with other positional parameters only after all other positional parameters:

```
[commandchars=\\\{\}]
  .param string first
  .param int second
  .param pmc the_rest :slurpy

  $P0 = the_rest[0]           # third argument
  $P1 = the_rest[1]           # fourth argument
```

When you combine `:named` and `:slurpy` on a parameter, the result is a single associative array containing the named arguments passed into the subroutine call:

```
[commandchars=\\\{\}]
  .param pmc all_named :slurpy :named

  $P0 = all_named['name']             # 'name' => 'Bob'
  $P1 = all_named['age']              # 'age'  => 42
```

Flattening Arguments

A flattening argument breaks up a single argument to fill multiple parameters. It's the complement of an aggregating parameter. The `:flat` modifier splits arguments (and return values) into a flattened list. Passing an array PMC to a subroutine with `:flat`:

```
[commandchars=\\\{\}]
  $P0 = new "ResizablePMCArray"
  $P0[0] = "Bob"
  $P0[1] = 42
  'foo'($P0 :flat)
```

. . . allows the elements of that array to fill the required parameters:

```
[commandchars=\\\{\}]
  .param string name  # Bob
  .param int age      # 42
```

Arguments on the Command Line

Arguments passed to a PIR program on the command line are available to the `:main` subroutine of that program as strings in a `ResizableStringArray` PMC. If you call a program *args.pir*, passing it three arguments:

```
[commandchars=\\\{\}]
  $ parrot args.pir foo bar baz
```

. . . they will be accessible at index 1, 2, and 3 of the PMC parameter.[20]

```
[commandchars=\\\{\}]
  .sub 'main' :main
    .param pmc all_args
    $S1 = all_args[1]   # foo
    $S2 = all_args[2]   # bar
    $S3 = all_args[3]   # baz
    # ...
  .end
```

Because `all_args` is a `ResizableStringArray` PMC, you can loop over the results, access them individually, or even modify them.

Compiling and Loading Libraries

In addition to running PIR files on the command-line, you can also load a library of pre-compiled bytecode directly into your PIR source file. The `load_bytecode` opcode takes a single argument: the name of the bytecode file to load. If you create a file named *foo_file.pir* containing a single subroutine:

```
[commandchars=\\\{\}]
  # foo_file.pir
  .sub 'foo_sub'              # .sub stores a global sub
     say "in foo_sub"
  .end
```

[20] Index 0 is unused.

. . . and compile it to bytecode using the `-o` command-line switch:

```
[commandchars=\\\{\}]
  $ parrot -o foo_file.pbc foo_file.pir
```

. . . you can then load the compiled bytecode into *main.pir* and directly call the subroutine defined in *foo_file.pir*:

```
[commandchars=\\\{\}]
  # main.pir
  .sub 'main' :main
    load_bytecode "foo_file.pbc"    # compiled foo_file.pir
    foo_sub()
  .end
```

The `load_bytecode` opcode also works with source files, as long as Parrot has a compiler registered for that type of file:

```
[commandchars=\\\{\}]
  # main2.pir
  .sub 'main' :main
    load_bytecode "foo_file.pir"  # PIR source code
    foo_sub()
  .end
```

Sub PMC

Subroutines are a PMC type in Parrot. You can store them in PMC registers and manipulate them just as you do with other PMCs. Parrot stores subroutines in namespaces; retrieve them with the `get_global` opcode:

```
[commandchars=\\\{\}]
  $P0 = get_global "my_sub"
```

To find a subroutine in a different namespace, first look up the appropriate the namespace object, then use that as the first parameter to `get_global`:

```
[commandchars=\\\{\}]
  $P0 = get_namespace ["My";"Namespace"]
  $P1 = get_global $P0, "my_sub"
```

You can invoke a Sub object directly:

```
[commandchars=\\\{\}]
  $P0(1, 2, 3)
```

You can get or even *change* its name:

```
[commandchars=\\\{\}]
  $S0 = $P0               # Get the current name
  $P0 = "my_new_sub"      # Set a new name
```

You can get a hash of the complete metadata for the subroutine:

```
[commandchars=\\\{\}]
  $P1 = inspect $P0
```

. . . which contains the fields:

- pos_required

  The number of required positional parameters

- pos_optional

  The number of optional positional parameters

- named_required

  The number of required named parameters

- named_optional

  The number of optional named parameters

- pos_slurpy

  True if the sub has an aggregating parameter for positional args

- named_slurpy

  True if the sub has an aggregating parameter for named args

Instead of fetching the entire inspection hash, you can also request individual pieces of metadata:

```
[commandchars=\\\{\}]
  $P1 = inspect $P0, "pos_required"
```

The `arity` method on the sub object returns the total number of defined parameters of all varieties:

```
[commandchars=\\\{\}]
  $I0 = $P0.'arity'()
```

The `get_namespace` method on the sub object fetches the namespace PMC which contains the Sub:

```
[commandchars=\\\{\}]
  $P1 = $P0.'get_namespace'()
```

Evaluating a Code String

One way of producing a code object during a running program is by compiling a code string. In this case, it's a bytecode segment object.

The first step is to fetch a compiler object for the target language:

```
[commandchars=\\\{\}]
  $P1 = compreg "PIR"
```

Parrot registers a compiler for PIR by default, so it's always available. The following example fetches a compiler object for PIR and places it in the named variable `compiler`. It then generates a code object from a string by calling `compiler` as a subroutine and places the resulting bytecode segment object into the named variable `generated` and then invokes it as a subroutine:

```
[commandchars=\\\{\}]
  .local pmc compiler, generated
  .local string source
  source    = ".sub foo\textbackslash{}n$S1 = 'in eval'\textbackslash{}nprint $S1\textbackslash{}n.end"
  compiler  = compreg "PIR"
  generated = compiler(source)
  generated()
  say "back again"
```

You can register a compiler or assembler for any language inside the Parrot core and use it to compile and invoke code from that language.

In the following example, the `compreg` opcode registers the subroutine-like object `$P10` as a compiler for the language "MyLanguage":

```
[commandchars=\\\{\}]
  compreg "MyLanguage", $P10
```

Lexicals

Variables stored in a namespace are global variables. They're accessible from anywhere in the program if you specify the right namespace path. High-level languages also have lexical variables which are only accessible from the local section of code (or *scope*) where they appear, or in a section of code embedded within that scope.[21] In PIR, the section of code between a `.sub` and a `.end` defines a scope for lexical variables.

While Parrot stores global variables in namespaces, it stores lexical variables in lexical pads[22]. Each lexical scope has its own pad. The `store_lex` opcode stores a lexical variable in the current pad. The `find_lex` opcode retrieves a variable from the current pad:

```
[commandchars=\\\{\}]
  $P0 = new "Integer"       # create a variable
  $P0 = 10                  # assign value to it
  store_lex 'foo', $P0      # store with lexical name "foo"
  # ...
  $P1 = find_lex 'foo'      # get the lexical "foo" into $P1
  say $P1                   # prints 10
```

The `.lex` directive defines a local variable that follows these scoping rules:

```
[commandchars=\\\{\}]
      .local pmc foo
      .lex 'foo', foo
```

Limitation: For now use only single-quotes for lexical variable names! Double-quoted lexical names in `.lex` are treated as already escaped, i.e. only as single-quoted and are not unescaped.

So never use this:

```
[commandchars=\\\{\}]
    .lex "foo\textbackslash{}\textbackslash{}o", $P3       # wrong, parsed as 'foo\textbackslash{}\textbackslash{
    $P1 = box 'ok 2'
    store_lex "foo\textbackslash{}\textbackslash{}o", $P1  # Lexical 'foo\textbackslash{}o' not found
    $P2 = find_lex "foo\textbackslash{}\textbackslash{}o"
```

[21] A scope is roughly equivalent to a block in C.

[22] Think of a pad like a box to hold a collection of lexical variables.

LexPad and LexInfo PMCs

Parrot uses two different PMCs to store information about a subroutine's lexical variables: the `LexPad` PMC and the `LexInfo` PMC. Neither of these PMC types are usable directly from PIR code; Parrot uses them internally to store information about lexical variables.

`LexInfo` PMCs store information about lexical variables at compile time. Parrot generates this read-only information during compilation to represent what it knows about lexical variables. Not all subroutines get a `LexInfo` PMC by default; subroutines need to indicate to Parrot that they require a `LexInfo` PMC. One way to do this is with the `.lex` directive. Of course, the `.lex` directive only works for languages that know the names of their lexical variables at compile time. Languages where this information is not available can mark the subroutine with `:lex` instead.

`LexPad` PMCs store run-time information about lexical variables. This includes their current values and type information. Parrot creates a new `LexPad` PMC for subs that have a `LexInfo` PMC already. It does so for each invocation of the subroutine, which allows for recursive subroutine calls without overwriting lexical variables.

The `get_lexinfo` method on a sub retrieves its associated `LexInfo` PMC:

```
[commandchars=\\\{\}]
  $P0 = get_global "MySubroutine"
  $P1 = $P0.'get_lexinfo'()
```

The `LexInfo` PMC supports a few introspection operations. The `elements` opcode retrieves the number of elements it contains. String key access operations retrieve entries from the `LexInfo` PMC as if it were an associative array.

```
[commandchars=\\\{\}]
  $I0 = elements $P1    # number of lexical variables
  $P0 = $P1["name"]     # lexical variable "name"
```

There is no easy way to retrieve the current `LexPad` PMC in a given subroutine, but they are of limited use in PIR.

Nested Scopes

PIR has no separate syntax for blocks or lexical scopes; subroutines define lexical scopes in PIR. Because PIR disallows nested `.sub`/`.end` declarations, it needs a way to identify which lexical scopes are the parents of inner lexical scopes. The `:outer` modifier declares a subroutine as a nested inner lexical scope of another existing subroutine. The modifier takes one argument, the name of the outer subroutine:

```
[commandchars=\\\{\}]
  .sub 'foo'
    # defines lexical variables
  .end
```

```
  .sub 'bar' :outer('foo')
    # can access foo's lexical variables
  .end
```

Sometimes a name alone isn't sufficient to uniquely identify the outer subroutine. The `:subid` modifier allows the outer subroutine to declare a truly unique name usable with `:outer`:

```
[commandchars=\\\{\}]
  .sub 'foo' :subid('barsouter')
    # defines lexical variables
  .end

  .sub 'bar' :outer('barsouter')
    # can access foo's lexical variables
  .end
```

The `get_outer` method on a `Sub` PMC retrieves its `:outer` sub.

```
[commandchars=\\\{\}]
  $P1 = $P0.'get_outer'()
```

If there is no `:outer` sub, this will return a null PMC. The `set_outer` method on a `Sub` object sets the `:outer` sub:

```
[commandchars=\\\{\}]
  $P0.'set_outer'($P1)
```

Scope and Visibility

High-level languages such as Perl, Python, and Ruby allow nested scopes, or blocks within blocks that have their own lexical variables. This construct is common even in C:

```
[commandchars=\\\{\}]
  \{
      int x = 0;
      int y = 1;
      \{
          int z = 2;
          /* x, y, and z are all visible here */
      \}

      /* only x and y are visible here */
  \}
```

In the inner block, all three variables are visible. The variable `z` is only visible inside that block. The outer block has no knowledge of `z`. A naïve translation of this code to PIR might be:

```
[commandchars=\\\{\}]
  .param int x
  .param int y
  .param int z
  x = 0
  y = 1
  z = 2
  #...
```

This PIR code is similar, but the handling of the variable `z` is different: `z` is visible throughout the entire current subroutine. It was not visible throughout the entire C function. A more accurate translation of the C scopes uses `:outer` PIR subroutines instead:

```
[commandchars=\\\{\}]
  .sub 'MyOuter'
      .local pmc x, y
      .lex 'x', x
      .lex 'y', y
      x = new 'Integer'
      x = 10
      'MyInner'()
      # only x and y are visible here
      say y                              # prints 20
  .end

  .sub 'MyInner' :outer('MyOuter')
      .local pmc x, new_y, z
      .lex 'z', z
      find_lex x, 'x'
      say x                             # prints 10
      new_y = new 'Integer'
      new_y = 20
      store_lex 'y', new_y
  .end
```

The `find_lex` and `store_lex` opcodes don't just access the value of a variable directly in the scope where it's declared, they interact with the `LexPad` PMC to find lexical variables within outer lexical scopes. All lexical variables from an outer lexical scope are visible from the inner lexical scope.

Note that you can only store PMCs—not primitive types—as lexicals.

Multiple Dispatch

Multiple dispatch subroutines (or *multis*) have several variants with the same name but different sets of parameters. The set of parameters for a subroutine is its *signature*. When a multi is called, the dispatch operation compares the arguments passed in to the signatures of all the variants and invokes the subroutine with the best match.

Parrot stores all multiple dispatch subs with the same name in a namespace within a single PMC called a `MultiSub`. The `MultiSub` is an invokable list of subroutines. When a multiple dispatch sub is called, the `MultiSub` PMC searches its list of variants for the best matching candidate.

The `:multi` modifier on a `.sub` declares a `MultiSub`:

```
[commandchars=\\\{\}]
  .sub 'MyMulti' :multi()
      # does whatever a MyMulti does
  .end
```

Each variant in a `MultiSub` must have a unique type or number of parameters declared, so the dispatcher can calculate a best match. If you had two

variants that both took four integer parameters, the dispatcher would never be able to decide which one to call when it received four integer arguments.

The `:multi` modifier takes one or more arguments defining the *multi signature*. The multi signature tells Parrot what particular combination of input parameters the multi accepts:

```
[commandchars=\\\{\}]
  .sub 'Add' :multi(I, I)
    .param int x
    .param int y
     $I0 = x + y
    .return($I0)
  .end

  .sub 'Add' :multi(N, N)
    .param num x
    .param num y
    $N0 = x + y
    .return($N0)
  .end

  .sub 'Start' :main
    $I0 = Add(1, 2)      # 3
    $N0 = Add(3.14, 2.0) # 5.14
    $S0 = Add("a", "b")  # ERROR! No (S, S) variant!
  .end
```

Multis can take I, N, S, and P types, but they can also use _ (underscore) to denote a wildcard, and a string which names a PMC type:

```
[commandchars=\\\{\}]
  .sub 'Add' :multi(I, I)          # two integers
    #...
  .end

  .sub 'Add' :multi(I, 'Float')    # integer and Float PMC
    #...
  .end

  .sub 'Add' :multi('Integer', _)  # Integer PMC and wildcard
    #...
  .end
```

When you call a `MultiSub`, Parrot will try to take the most specific best-match variant, but will fall back to more general variants if it cannot find a perfect match. If you call `Add` with `(1, 2)`, Parrot will dispatch to the `(I, I)` variant. If you call it with `(1, ''hi'')`, Parrot will match the `(I, _)` variant, as the string in the second argument doesn't match `I` or `Float`. Parrot can also promote one of the I, N, or S values to an Integer, Float, or String PMC.

To make the decision about which multi variant to call, Parrot calculates the *Manhattan Distance* between the argument signature and the parameter signature of each variant. Every difference between each element counts as one step. A difference can be a promotion from a primitive type to a PMC, the conversion from one primitive type to another, or the matching of an argument to a _ wildcard. After Parrot calculates the distance to each

variant, it calls the one with the lowest distance. Notice that it's possible to define a variant that is impossible to call: for every potential combination of arguments there is a better match. This is uncommon, but possible in systems with many multis and a limited number of data types.

Continuations

Continuations are subroutines that take snapshots of control flow. They are frozen images of the current execution state of the VM. Once you have a continuation, you can invoke it to return to the point where the continuation was first created. It's like a magical timewarp that allows the developer to arbitrarily move control flow back to any previous point in the program.

Continuations are like any other PMC; create one with the `new` opcode:

```
[commandchars=\\\{\}]
  $P0 = new 'Continuation'
```

The new continuation starts in an undefined state. If you attempt to invoke a new continuation without initializing it, Parrot will throw an exception. To prepare the continuation for use, assign it a destination label with the `set_addr` opcode:

```
[commandchars=\\\{\}]
    $P0 = new 'Continuation'
    set_addr $P0, my_label

  my_label:
    # ...
```

To jump to the continuation's stored label and return the context to the state it was in *at the point of its creation*, invoke the continuation:

```
[commandchars=\\\{\}]
  $P0()
```

Even though you can use the subroutine call notation `$P0()` to invoke the continuation, you cannot pass arguments or obtain return values.

Continuation Passing Style

Parrot uses continuations internally for control flow. When Parrot invokes a subroutine, it creates a continuation representing the current point in the program. It passes this continuation as an invisible parameter to the subroutine call. To return from that subroutine, Parrot invokes the continuation to return to the point of creation of that continuation. If you have a continuation, you can invoke it to return to its point of creation any time you want.

This type of flow control—invoking continuations instead of performing bare jumps—is called Continuation Passing Style (CPS).

Tailcalls

Many subroutines set up and call another subroutine and then return the result of the second call directly. This is a tailcall, and is an important opportunity for optimization. Here's a contrived example in pseudocode:

```
[commandchars=\\\{\}]
  call add_two(5)

  subroutine add_two(value)
    value = add_one(value)
    return add_one(value)
```

In this example, the subroutine `add_two` makes two calls to `add_one`. The second call to `add_one` is the return value. `add_one` gets called; its result gets returned to the caller of `add_two`. Nothing in `add_two` uses that return value directly.

A simple optimization is available for this type of code. The second call to `add_one` can return to the same place that `add_two` returns; it's perfectly safe and correct to use the same return continuation that `add_two` uses. The two subroutine calls can share a return continuation.

PIR provides the `.tailcall` directive to identify similar situations. Use it in place of the `.return` directive. `.tailcall` performs this optimization by reusing the return continuation of the parent subroutine to make the tailcall:

```
[commandchars=\\\{\}]
  .sub 'main' :main
      .local int value
      value = add_two(5)
      say value
  .end

  .sub 'add_two'
      .param int value
      .local int val2
      val2 = add_one(value)
      .tailcall add_one(val2)
  .end

  .sub 'add_one'
      .param int a
      .local int b
      b = a + 1
      .return (b)
  .end
```

This example prints the correct value `7`.

Coroutines

Coroutines are similar to subroutines except that they have an internal list of *states* for each `.yield` call. In addition to performing a normal `.return` to return control flow back to the caller and destroy the execution environment of the subroutine, coroutines may also perform a `.yield` operation. `.yield`

returns a value to the caller like `.return` can, but it does not destroy the execution state of the coroutine. The next call to the coroutine continues execution from the point of the last `.yield`, not at the beginning of the coroutine.

Inside a coroutine continuing from a `.yield`, the entire execution environment is the same as it was when the coroutine `.yield`ed. This means that the parameter values don't change, even if the next invocation of the coroutine had different arguments passed in.

Coroutines look like ordinary subroutines. They do not require any special modifier or any special syntax to mark them as being a coroutine. What sets them apart is the use of the `.yield` directive. `.yield` plays several roles:

- Identifies coroutines

  When Parrot sees a `.yield`, it knows to create a Coroutine PMC object instead of a `Sub` PMC.

- Creates a continuation

  `.yield` creates a continuation in the coroutine and stores the continuation object in the coroutine object for later resuming from the point of the `.yield`.

- Returns a value

  `.yield` can return a value [23] to the caller. It is basically the same as a `.return` in this regard.

Here is a simple coroutine example:

```
[commandchars=\\\{\}]
  .sub 'MyCoro'
    .yield(1)
    .yield(2)
    .yield(3)
    .return(4)
  .end

  .sub 'main' :main
    $I0 = MyCoro()    # 1
    $I0 = MyCoro()    # 2
    $I0 = MyCoro()    # 3
    $I0 = MyCoro()    # 4
    $I0 = MyCoro()    # 1
    $I0 = MyCoro()    # 2
    $I0 = MyCoro()    # 3
    $I0 = MyCoro()    # 4
    $I0 = MyCoro()    # 1
    $I0 = MyCoro()    # 2
    $I0 = MyCoro()    # 3
    $I0 = MyCoro()    # 4
  .end
```

[23] . . . or many values, or no values.

This contrived example demonstrates how the coroutine stores its state. When Parrot encounters the `.yield`, the coroutine stores its current execution environment. At the next call to the coroutine, it picks up where it left off.

Native Call Interface

The Native Call Interface (NCI) is a special version of the Parrot calling conventions for calling functions in shared C libraries with a known signature. This is a simplified version of the first test in *t/pmc/nci.t*:

```
[commandchars=\\\{\}]
    .local pmc library
    library = loadlib "libnci_test"         # library object
    say "loaded"

    .local pmc ddfunc
    ddfunc = dlfunc library, "nci_dd", "dd" # function object
    say "dlfunced"

    .local num result
    result = ddfunc( 4.0 )                  # call the function

    ne result, 8.0, nok_1
    say "ok 1"
    end
  nok_1:
    say "not ok 1"

    #...
```

This example shows two new opcodes: `loadlib` and `dlfunc`. The `loadlib` opcode obtains a handle for a shared library. It searches for the shared library in the current directory, in *runtime/parrot/dynext*, and in a few other configured directories. It also tries to load the provided filename unaltered and with appended extensions like *.so* or *.dll*. Which extensions it tries depends on the operating system Parrot is running on.

The `dlfunc` opcode gets a function object from a previously loaded library (second argument) of a specified name (third argument) with a known function signature (fourth argument). The function signature is a string where the first character is the return value and the rest of the parameters are the function parameters. Table 6-1 lists the characters used in NCI function signatures.

5.7 Classes and Objects

Many of Parrot's core classes—such as `Integer`, `String`, or `ResizablePMCArray`—are written in C, but you can also write your own classes in PIR. PIR doesn't have the shiny syntax of high-level object-oriented languages, but it provides the necessary features to construct well-behaved objects every bit as powerful as those of high-level object systems.

Table 5.5: Function signature letters

Character	Register	C type
v	-	void (no return value)
c	I	char
s	I	short
i	I	int
l	I	long
f	N	float
d	N	double
t	S	char *
p	P	void * (or other pointer)
I	-	Parrot_Interp *interpreter
C	-	a callback function pointer
D	-	a callback function pointer
Y	P	the subroutine C or D calls into
Z	P	the argument for Y

Parrot developers often use the word "PMCs" to refer to the objects defined in C classes and "objects" to refer to the objects defined in PIR. In truth, all PMCs are objects and all objects are PMCs, so the distinction is a community tradition with no official meaning.

Class Declaration

The `newclass` opcode defines a new class. It takes a single argument, the name of the class to define.

```
[commandchars=\\\{\}]
    $P0 = newclass 'Foo'
```

Just as with Parrot's core classes, the `new` opcode instantiates a new object of a named class.

```
[commandchars=\\\{\}]
  $P1 = new 'Foo'
```

In addition to a string name for the class, `new` can also instantiate an object from a class object or from a keyed namespace name.

```
[commandchars=\\\{\}]
  $P0 = newclass 'Foo'
  $P1 = new $P0

  $P2 = new ['Bar';'Baz']
```

Attributes

The `addattribute` opcode defines a named attribute—or *instance variable*—in the class:

```
[commandchars=\\\{\}]
  $P0 = newclass 'Foo'
  addattribute $P0, 'bar'
```

The `setattribute` opcode sets the value of a declared attribute. You must declare an attribute before you may set it. The value of an attribute is always a PMC, never an integer, number, or string.[24]

```
[commandchars=\\\{\}]
    $P6 = box 42
    setattribute $P1, 'bar', $P6
```

The `getattribute` opcode fetches the value of a named attribute. It takes an object and an attribute name as arguments and returns the attribute PMC:

```
[commandchars=\\\{\}]
    $P10 = getattribute $P1, 'bar'
```

Because PMCs are containers, you may modify an object's attribute by retrieving the attribute PMC and modifying its value. You don't need to call `setattribute` for the change to stick:

```
[commandchars=\\\{\}]
    $P10 = getattribute $P1, 'bar'
    $P10 = 5
```

Instantiation

With a created class, we can use the `new` opcode to instantiate an object of that class in the same way we can instantiate a new PMC.

```
[commandchars=\\\{\}]
    $P0 = newclass "Foo"
    $P1 = new $P0
```

Or, if we don't have the class object handy, we can do it by name too:

```
[commandchars=\\\{\}]
    $P1 = new "Foo"
```

PMCs have two VTABLE interface functions for dealing with instantiating a new object: `init` and `init_pmc`. The former is called when a new PMC is created, the later is called when a new PMC is created with an initialization argument.

In order to iterate over the init_pmc args, you must also provide a get_iter method.

[24] Though it can be an `Integer`, `Number`, or `String` PMC.

```
[commandchars=\\\{\}]
    .namespace ["Foo"]
    .sub 'init' :vtable
        say "Creating a new Foo"
    .end

    .sub 'init_pmc' :vtable
        .param pmc args
        print "Creating a new Foo with argument "
        say args
    .end

    .sub 'get_iter' :vtable
        .param pmc self
        .local pmc args, iter
        args = new ['FixedPMCArray'], 1
        iter = new ['ArrayIterator'], args
        .return(iter)
    .end

    .namespace[]
    .sub 'main' :main
        $P0 = newclass 'Foo'    # instantiate class
        $P1 = new ['Foo']       # init object
        $P2 = new ['Foo'], $P1  # init_pmc
    .end
```

Methods

Methods in PIR are subroutines stored in the class object. Define a method with the `.sub` directive and the `:method` modifier:

```
[commandchars=\\\{\}]
  .sub half :method
    $P0 = getattribute self, 'bar'
    $P1 = $P0 / 2
    .return($P1)
  .end
```

This method returns the integer value of the `bar` attribute of the object divided by two. Notice that the code never declares the named variable `self`. Methods always make the invocant object—the object on which the method was invoked—available in a local variable called `self`.

The `:method` modifier adds the subroutine to the class object associated with the currently selected namespace, so every class definition file must contain a `.namespace` declaration. Class files for languages may also contain an `.HLL` declaration to associate the namespace with the appropriate high-level language:

```
[commandchars=\\\{\}]
  .HLL 'php'
  .namespace [ 'Foo' ]
```

Method calls in PIR use a period (`.`) to separate the object from the method name. The method name is either a literal string in quotes or a string variable. The method call looks up the method in the invocant object using the string name:

```
[commandchars=\\\{\}]
    $P0 = $P1.'half'()

    $S2 = 'double'
    $P0 = $P1.$S2()
```

You can also pass a method object to the method call instead of looking it up by string name:

```
[commandchars=\\\{\}]
    $P2 = get_global 'triple'
    $P0 = $P1.$P2()
```

Parrot always treats a PMC used in the method position as a method object, so you can't pass a `String` PMC as the method name.

Methods can have multiple arguments and multiple return values just like subroutines:

```
[commandchars=\\\{\}]
  ($P0, $S1) = $P2.'method'($I3, $P4)
```

The `can` opcode checks whether an object has a particular method. It returns 0 (false) or 1 (true):

```
[commandchars=\\\{\}]
    $I0 = can $P3, 'add'
```

Inheritance

The `subclass` opcode creates a new class that inherits methods and attributes from another class. It takes two arguments: the name of the parent class and the name of the new class:

```
[commandchars=\\\{\}]
    $P3 = subclass 'Foo', 'Bar'
```

`subclass` can also take a class object as the parent class instead of a class name:

```
[commandchars=\\\{\}]
    $P3 = subclass $P2, 'Bar'
```

The `addparent` opcode also adds a parent class to a subclass. This is especially useful for multiple inheritance, as the `subclass` opcode only accepts a single parent class:

```
[commandchars=\\\{\}]
  $P4 = newclass 'Baz'
  addparent $P3, $P4
  addparent $P3, $P5
```

To override an inherited method in the child class, define a method with the same name in the subclass. This example code overrides `Bar`'s `who_am_i` method to return a more meaningful name:

```
[commandchars=\\\{\}]
  .namespace [ 'Bar' ]

  .sub 'who_am_i' :method
    .return( 'I am proud to be a Bar' )
  .end
```

Object creation for subclasses is the same as for ordinary classes:

```
[commandchars=\\\{\}]
    $P5 = new 'Bar'
```

Calls to inherited methods are just like calls to methods defined in the class:

```
[commandchars=\\\{\}]
    $P1.'increment'()
```

The `isa` opcode checks whether an object is an instance of or inherits from a particular class. It returns 0 (false) or 1 (true):

```
[commandchars=\\\{\}]
    $I0 = isa $P3, 'Foo'
    $I0 = isa $P3, 'Bar'
```

Overriding Vtable Functions

The `Object` PMC is a core PMC written in C that provides basic object-like behavior. Every object instantiated from a PIR class inherits a default set of vtable functions from `Object`, but you can override them with your own PIR subroutines.

The `:vtable` modifier marks a subroutine as a vtable override. As it does with methods, Parrot stores vtable overrides in the class associated with the currently selected namespace:

```
[commandchars=\\\{\}]
  .sub 'init' :vtable
    $P6 = new 'Integer'
    setattribute self, 'bar', $P6
    .return()
  .end
```

Subroutines acting as vtable overrides must either have the name of an actual vtable function or include the vtable function name in the `:vtable` modifier:

```
[commandchars=\\\{\}]
  .sub foozle :vtable('init')
    # ...
  .end
```

You must call methods on objects explicitly, but Parrot calls vtable functions implicitly in multiple contexts. For example, creating a new object with `$P3 = new 'Foo'` will call `init` with the new `Foo` object.

As an example of some of the common vtable overrides, the `=` operator (or `set` opcode) calls `Foo`'s vtable function `set_integer_native` when its left-hand side is a `Foo` object and the argument is an integer literal or integer variable:

```
[commandchars=\\\{\}]
    $P3 = 30
```

The `+` operator (or `add` opcode) calls `Foo`'s `add` vtable function when it adds two `Foo` objects:

```
[commandchars=\\\{\}]
    $P3 = new 'Foo'
    $P3 = 3
    $P4 = new 'Foo'
    $P4 = 1774

    $P5 = $P3 + $P4
    # or:
    add $P5, $P3, $P4
```

The `inc` opcode calls `Foo`'s `increment` vtable function when it increments a `Foo` object:

```
[commandchars=\\\{\}]
    inc $P3
```

Parrot calls `Foo`'s `get_integer` and `get_string` vtable functions to retrieve an integer or string value from a `Foo` object:

```
[commandchars=\\\{\}]
    $I10 = $P5  # get_integer
    say $P5     # get_string
```

Introspection

Classes defined in PIR using the `newclass` opcode are instances of the `Class` PMC. This PMC contains all the meta-information for the class, such as attribute definitions, methods, vtable overrides, and its inheritance hierarchy. The opcode `inspect` provides a way to peek behind the curtain of encapsulation to see what makes a class tick. When called with no arguments, `inspect` returns an associative array containing data on all characteristics of the class that it chooses to reveal:

```
[commandchars=\\\{\}]
  $P1 = inspect $P0
  $P2 = $P1['attributes']
```

When called with a string argument, `inspect` only returns the data for a specific characteristic of the class:

```
[commandchars=\\\{\}]
  $P0 = inspect $P1, 'parents'
```

Table 7-1 shows the introspection characteristics supported by `inspect`.

5.8 I/O

Parrot handles all I/O in Parrot with a set of PMCs. The `FileHandle` PMC takes care of reading from and writing to files and file-like streams. The `Socket` PMC takes care of network I/O.

Table 5.6: Class Introspection

Characteristic	Description
`attributes`	Information about the attributes the class will instantiate in its objects. An a
`flags`	An `Integer` PMC containing any integer flags set on the class object.
`methods`	A list of methods provided by the class. An associative array where the keys
`name`	A `String` PMC containing the name of the class.
`namespace`	The `NameSpace` PMC associated with the class.
`parents`	An array of `Class` objects that this class inherits from directly (via `subclass`
`roles`	An array of `Role` objects composed into the class.
`vtable_overrides`	A list of vtable overrides defined by the class. An associative array where the

FileHandle Opcodes

The `open` opcode opens a new filehandle. It takes a string argument, which is the path to the file:

```
[commandchars=\\\{\}]
  .loadlib 'io_ops'

  # ...

  $P0 = open 'my/file/name.txt'
```

By default, it opens the filehandle as read-only, but an optional second string argument can specify the mode for the file. The modes are `r` for read, `w` for write, `a` for append, and `p` for pipe:[25]

```
[commandchars=\\\{\}]
  .loadlib 'io_ops'

  # ...

  $P0 = open 'my/file/name.txt', 'a'

  $P0 = open 'myfile.txt', 'r'
```

You can combine modes; a handle that can read and write uses the mode string `rw`. A handle that can read and write but will not overwrite the existing contents uses `ra` instead.

The `close` opcode closes a filehandle when it's no longer needed. Closing a filehandle doesn't destroy the object, it only makes that filehandle object available for opening a different file.[26]

```
[commandchars=\\\{\}]
  .loadlib 'io_ops'
```

[25] These are the same as the C language read-modes, so may be familiar.

[26] It's generally not a good idea to manually close the standard input, standard output, or standard error filehandles, though you can recreate them.

```
  # ...

  close $P0
```

The `print` opcode prints a string argument or the string form of an integer, number, or PMC to a filehandle:

```
[commandchars=\\\{\}]
  print $P0, 'Nobody expects'
```

It also has a one-argument variant that always prints to standard output:

```
[commandchars=\\\{\}]
  print 'the Spanish Inquisition'
```

The `say` opcode also prints to standard output, but it appends a trailing newline to whatever it prints. Another opcode worth mentioning is the `printerr` opcode, which prints an argument to the standard error instead of standard output:

```
[commandchars=\\\{\}]
  say 'Turnip'

  # ...

  .loadlib 'io_ops'

  # ...

  printerr 'Blancmange'
```

The `read` and `readline` opcodes read values from a filehandle. `read` takes an integer value and returns a string with that many characters (if possible). `readline` reads a line of input from a filehandle and returns the string without the trailing newline:

```
[commandchars=\\\{\}]
  .loadlib 'io_ops'

  $S0 = read $P0, 10

  $S0 = readline $P0
```

The `read` opcode has a one-argument variant that reads from standard input:

```
[commandchars=\\\{\}]
  .loadlib 'io_ops'

  # ...

  $S0 = read 10
```

The `getstdin`, `getstdout`, and `getstderr` opcodes fetch the filehandle objects for the standard streams: standard input, standard output, and standard error:

```
[commandchars=\\\{\}]
  .loadlib 'io_ops'

  # ...

  $P0 = getstdin    # Standard input handle
  $P1 = getstdout   # Standard output handle
  $P2 = getstderr   # Standard error handle
```

Once you have the filehandle for one of the standard streams, you can use it just like any other filehandle object:

```
[commandchars=\\\{\}]
  .loadlib 'io_ops'

  # ...

  $P0 = getstdout
  print $P0, 'hello'
```

This following example reads data from the file *myfile.txt* one line at a time using the `readline` opcode. As it loops over the lines of the file, it checks the boolean value of the read-only filehandle `$P0` to test whether the filehandle has reached the end of the file:

```
[commandchars=\\\{\}]
  .loadlib 'io_ops'

  .sub 'main'
    $P0 = getstdout
    $P1 = open 'myfile.txt', 'r'
    loop_top:
      $S0 = readline $P1
      print $P0, $S0
      if $P1 goto loop_top
    close $P1
  .end
```

FileHandle Methods

The methods available on a filehandle object are mostly duplicates of the opcodes, though sometimes they provide more options. Behind the scenes many of the opcodes call the filehandle's methods anyway, so the choice between the two is more a matter of style preference than anything else.

open

The `open` method opens a stream in an existing filehandle object. It takes two optional string arguments: the name of the file to open and the open mode.

```
[commandchars=\\\{\}]
  $P0 = new 'FileHandle'
  $P0.'open'('myfile.txt', 'r')
```

The `open` opcode internally creates a new filehandle PMC and calls its `open` method on it. The opcode version is shorter to write, but it also creates a new PMC for every call, while the method can reopen an existing filehandle PMC with a new file.

When reopening a filehandle, Parrot will reuse the previous filename associated with the filehandle unless you provide a different filename. The same goes for the mode.

close

The `close` method closes the filehandle. This does not destroy the filehandle object; you can reopen it with the `open` method later.

```
[commandchars=\\\{\}]
  $P0.'close'()
```

is_closed

The `is_closed` method checks if the filehandle is closed. It returns true if the filehandle has been closed or was never opened, and false if it is currently open:

```
[commandchars=\\\{\}]
  $I0 = $P0.'is_closed'()
```

print

The `print` method prints a given value to the filehandle. The argument can be an integer, number, string, or PMC.

```
[commandchars=\\\{\}]
  $P0.'print'('Hello!')
```

read

The `read` method reads a specified number of bytes from the filehandle object and returns them in a string.

```
[commandchars=\\\{\}]
  $S0 = $P0.'read'(10)
```

If the remaining bytes in the filehandle are fewer than the requested number of bytes, returns a string containing the remaining bytes.

readline

The `readline` method reads an entire line up to a newline character or the end-of-file mark from the filehandle object and returns it in a string.

```
[commandchars=\\\{\}]
  $S0 = $P0.'readline'()
```

readline_interactive

The `readline_interactive` method is useful for command-line scripts. It writes the single argument to the method as a prompt to the screen, then reads back a line of input.

```
[commandchars=\\\{\}]
  $S0 = $P0.'readline_interactive'('Please enter your name:')
```

readall

The `readall` method reads an entire file. If the filehandle is closed, it will open the file given by the passed in string argument, read the entire file, and then close the filehandle.

```
[commandchars=\\\{\}]
  $S0 = $P0.'readall'('myfile.txt')
```

If the filehandle is already open, `readall` will read the contents of the file, and won't close the filehandle when it's finished. Don't pass the name argument when working with a file you've already opened.

```
[commandchars=\\\{\}]
  $S0 = $P0.'readall'()
```

mode

The `mode` method returns the current file access mode for the filehandle object.

```
[commandchars=\\\{\}]
  $S0 = $P0.'mode'()
```

encoding

The `encoding` method sets or retrieves the string encoding behavior of the filehandle.

```
[commandchars=\\\{\}]
 $P0.'encoding'('utf8')
 $S0 = $P0.'encoding'()
```

See ``EncodingsandCharsets'' in Chapter 4 for more details on the encodings supported in Parrot.

buffer_type

The `buffer_type` method sets or retrieves the buffering behavior of the filehandle object. The argument or return value is one of: `unbuffered` to disable buffering, `line-buffered` to read or write when the filehandle encounters a line ending, or `full-buffered` to read or write bytes when the buffer is full.

```
[commandchars=\\\{\}]
  $P0.'buffer_type'('full-buffered')
  $S0 = $P0.'buffer_type'()
```

buffer_size

The `buffer_size` method sets or retrieves the buffer size of the filehandle object.

```
[commandchars=\\\{\}]
  $P0.'buffer_size'(1024)
  $I0 = $P0.'buffer_size'()
```

The buffer size set on the filehandle is only a suggestion. Parrot may allocate a larger buffer, but it will never allocate a smaller buffer.

flush

The `flush` method flushes the buffer if the filehandle object is working in a buffered mode.

```
[commandchars=\\\{\}]
  $P0.'flush'()
```

eof

The `eof` method checks whether a filehandle object has reached the end of the current file. It returns true if the filehandle is at the end of the current file and false otherwise.

```
[commandchars=\\\{\}]
  $I0 = $P0.'eof'()
```

isatty

The `isatty` method returns a boolean value whether the filehandle is a TTY terminal.

```
[commandchars=\\\{\}]
  $P0.'isatty'()
```

get_fd

The `get_fd` method returns the integer file descriptor of the current filehandle object. Not all operating systems use integer file descriptors. Those that don't simply return `-1`.

```
[commandchars=\\\{\}]
  $I0 = $P0.'get_fd'()
```

5.9 Exceptions

Exceptions provide a way of subverting the normal flow of control. Their main use is error reporting and cleanup tasks, but sometimes exceptions are just a funny way to jump from one code location to another one. Parrot uses a robust exception mechanism and makes it available to PIR.

Exceptions are objects that hold essential information about an exceptional situation: the error message, the severity and type of the error, the location of the error, and backtrace information about the chain of calls that led to the error. Exception handlers are ordinary subroutines, but user code never calls them directly from within user code. Instead, Parrot invokes an appropriate exception handler to catch a thrown exception.

Throwing Exceptions

The `throw` opcode throws an exception object. This example creates a new `Exception` object in `$P0` and throws it:

```
[commandchars=\\\{\}]
  $P0 = new 'Exception'
  throw $P0
```

Setting the string value of an exception object sets its error message:

```
[commandchars=\\\{\}]
  $P0 = new 'Exception'
  $P0 = "I really had my heart set on halibut."
  throw $P0
```

Other parts of Parrot throw their own exceptions. The `die` opcode throws a fatal (that is, uncatchable) exception. Many opcodes throw exceptions to indicate error conditions. The `/` operator (the `div` opcode), for example, throws an exception on attempted division by zero.

When no appropriate handlers are available to catch an exception, Parrot treats it as a fatal error and exits, printing the exception message followed by a backtrace showing the location of the thrown exception:

```
[commandchars=\\\{\}]
  I really had my heart set on halibut.
  current instr.: 'main' pc 6 (pet_store.pir:4)
```

Catching Exceptions

Exception handlers catch exceptions, making it possible to recover from errors in a controlled way, instead of terminating the process entirely.

The `push_eh` opcode creates an exception handler object and stores it in the list of currently active exception handlers. The body of the exception handler is a labeled section of code inside the same subroutine as the call to `push_eh`. The opcode takes one argument, the name of the label:

```
[commandchars=\\\{\}]
  push_eh my_handler
    $P0 = new 'Exception'
    throw $P0

    say 'never printed'

  my_handler:
    say 'caught an exception'
```

This example creates an exception handler with a destination address of the `my_handler` label, then creates a new exception and throws it. At this point, Parrot checks to see if there are any appropriate exception handlers in the currently active list. It finds `my_handler` and runs it, printing "caught an exception". The "never printed" line never runs, because the exceptional control flow skips right over it.

Because Parrot scans the list of active handlers from newest to oldest, you don't want to leave exception handlers lying around when you're done with them. The `pop_eh` opcode removes an exception handler from the list of currently active handlers:

```
[commandchars=\\\{\}]
  push_eh my_handler
    $I0 = $I1 / $I2
    pop_eh

    say 'maybe printed'

    goto skip_handler

  my_handler:
    say 'caught an exception'
    pop_eh

  skip_handler:
```

This example creates an exception handler `my_handler` and then runs a division operation that will throw a "division by zero" exception if `$I2` is 0. When `$I2` is 0, `div` throws an exception. The exception handler catches it, prints "caught an exception", and then clears itself with `pop_eh`. When `$I2` is a non-zero value, there is no exception. The code clears the exception handler with `pop_eh`, then prints "maybe printed". The `goto` skips over the code of the exception handler, as it's just a labeled unit of code within the subroutine.

The exception object provides access to various attributes of the exception for additional information about what kind of error it was, and what might have caused it. The directive `.get_results` retrieves the `Exception` object from inside the handler:

```
[commandchars=\\\{\}]
  my_handler:
    .get_results($P0)
```

Not all handlers are able to handle all kinds of exceptions. If a handler determines that it's caught an exception it can't handle, it can `rethrow` the exception to the next handler in the list of active handlers:

```
[commandchars=\\\{\}]
  my_handler:
    .get_results($P0)
    rethrow $P0
```

If none of the active handlers can handle the exception, the exception becomes a fatal error. Parrot will exit, just as if it could find no handlers.

An exception handler creates a return continuation with a snapshot of the current interpreter context. If the handler is successful, it can resume running at the instruction immediately after the one that threw the exception. This resume continuation is available from the `resume` attribute of the exception object. To resume after the exception handler is complete, call the resume handler like an ordinary subroutine:

```
[commandchars=\\\{\}]
  my_handler:
    .get_results($P0)
    $P1 = $P0['resume']
    $P1()
```

Exception PMC

`Exception` objects contain several useful pieces of information about the exception. To set and retrieve the exception message, use the `message` key on the exception object:

```
[commandchars=\\\{\}]
  $P0            = new 'Exception'
  $P0['message'] = "this is an error message for the exception"
```

. . . or set and retrieve the string value of the exception object directly:

```
[commandchars=\\\{\}]
  $S0 = $P0
```

The severity and type of the exception are both integer values:

```
[commandchars=\\\{\}]
  $P0['severity'] = 1
  $P0['type']     = 2
```

The payload holds any user-defined data attached to the exception object:

```
[commandchars=\\\{\}]
  $P0['payload'] = $P2
```

The attributes of the exception are useful in the handler for making decisions about how and whether to handle an exception and report its results:

```
[commandchars=\\\{\}]
  my_handler:
    .get_results($P2)
    $S0 = $P2['message']
    print 'caught exception: "'
    print $S0
    $I0 = $P2['type']
    print '", of type '
    say $I0
```

ExceptionHandler PMC

Exception handlers are subroutine-like PMC objects, derived from Parrot's `Continuation` type. When you use `push_eh` with a label to create an exception handler, Parrot creates the handler PMC for you. You can also create it directly by creating a new `ExceptionHandler` object, and setting its destination address to the label of the handler using the `set_addr` opcode:

```
[commandchars=\\\{\}]
    $P0 = new 'ExceptionHandler'
    set_addr $P0, my_handler
    push_eh $P0
    # ...

  my_handler:
    # ...
```

`ExceptionHandler` PMCs have several methods for setting or checking handler attributes. The `can_handle` method reports whether the handler is willing or able to handle a particular exception. It takes one argument, the exception object to test:

```
[commandchars=\\\{\}]
  $I0 = $P0.'can_handle'($P1)
```

The `min_severity` and `max_severity` methods set and retrieve the severity attributes of the handler, allowing it to refuse to handle any exceptions whose severity is too high or too low. Both take a single optional integer argument to set the severity; both return the current value of the attribute as a result:

```
[commandchars=\\\{\}]
  $P0.'min_severity'(5)
  $I0 = $P0.'max_severity'()
```

The `handle_types` and `handle_types_except` methods tell the exception handler what types of exceptions it should or shouldn't handle. Both take a list of integer types, which correspond to the `type` attribute set on an exception object:

```
[commandchars=\\\{\}]
  $P0.'handle_types'(5, 78, 42)
```

The following example creates an exception handler that only handles exception types 1 and 2. Instead of having `push_eh` create the exception handler object, it creates a new `ExceptionHandler` object manually. It then calls `handle_types` to identify the exception types it will handle:

```
[commandchars=\\\{\}]
    $P0 = new 'ExceptionHandler'
    set_addr $P0, my_handler
    $P0.'handle_types'(1, 2)
    push_eh $P0
    # ...
  my_handler:
    # ...
```

This handler can only handle exception objects with a type of 1 or 2. Parrot will skip over this handler for all other exception types.

```
[commandchars=\\\{\}]
  $P1         = new 'Exception'
  $P1['type'] = 2
  throw $P1                     # caught

  $P1         = new 'Exception'
  $P1['type'] = 3
  throw $P1                     # uncaught
```

Annotations

Annotations are pieces of metadata code stored in a bytecode file. This is especially important when dealing with high-level languages, where annotations contain information about the HLL's source code such as the current line number and file name.

Create an annotation with the `.annotate` directive. Annotations consist of a key/value pair, where the key is a string and the value is an integer, or a string. Bytecode stores annotations as constants in the compiled bytecode. Consequently, you may not store PMCs.

```
[commandchars=\\\{\}]
  .annotate 'file', 'mysource.lang'
  .annotate 'line', 42
  .annotate 'compiletime', '0.3456'
```

Annotations exist, or are "in force" throughout the entire subroutine or until their redefinition. Creating a new annotation with the same name as an old one overwrites it with the new value. The `annotations` opcode retrieves the current hash of annotations:

```
[commandchars=\\\{\}]
  .annotate 'line', 1
  $P0 = annotations # \{'line' => 1\}

  .annotate 'line', 2
  $P0 = annotations # \{'line' => 2\}
```

To retrieve a single annotation by name, use the name with `annotations`:

```
[commandchars=\\\{\}]
  $P0 = annotations 'line'
```

Exception objects contain information about the annotations that were in force when the exception was thrown. Retrieve them with the `annotations` method on the exception PMC object:

```
[commandchars=\\\{\}]
  $I0 = $P0.'annotations'('line')  # only the 'line' annotation
  $P1 = $P0.'annotations'()        # hash of all annotations
```

Exceptions can also include a backtrace to display the program flow to the point of the throw:

```
[commandchars=\\\{\}]
  $P1 = $P0.'backtrace'()
```

The backtrace PMC is an array of hashes. Each element in the array corresponds to a function in the current call chain. Each hash has two elements: `annotation` (the hash of annotations in effect at that point) and `sub` (the Sub PMC of that function).