Perl 6: Project

Elliot Chandler (Wallace)

11 December 2017

Abstract

This article introduces and summarizes the Perl 6 programming language and some of its various features and capabilities, as well as discussing its applications, effective uses, popularity, prospects, and limitations.

Part 1

My first choice of language is Perl 6.

My second choice of language is COBOL.

Perl 6 is my first choice of language for two main reasons. First, Perl 6 has strong built-in support for implementing other languages, and I would like to learn to use this: I would like to implement a document format that I have been developing, and I think that knowing Perl 6 Grammars could help me with that. While Perl 6 reportedly has issues with performance, and may not be an appropriate long-term choice for implementing the format, it seems like it would be excellent for rapid prototyping and iterative development of the format.

Second, Perl 6 has a number of features that I have heard about, that seem to have a lot of ”buzz”, and that seem to have wide applicability and popular use, but that I don’t have any significant experience with or understanding of the use of: multiple dispatch, promises, asynchronous I/O, polymorphism, functional programming, introspection, generic programming, meta-object programming, and macros. Because that would be a lot of things to learn in only a semester, I would presumably select a few areas to focus on.

My second language choice is COBOL, because it has a long history of being very capable for data processing work and file handling, and a lot of businesses seem to use it successfully. Because of that history of success with it, I think that it would be good to have experience with. (While C and Java have similar histories of successful use, I do not propose studying them as I have used them briefly.) I am also interested in learning about COBOL because, as a file processing and business language, it is quite different from the languages I have experience using. (Perl 6 is also quite different, because of its capacity for functional programming.)

For reference, the languages with which I have the most confidence and experience in are, in roughly descending order:

- Bash (along with PHP, which is a reasonably close second, I have by far the most confidence and experience in Bash, and generally find it enjoyable to use)

- PHP 5 (I used it a lot until I switched to writing most things in Bash)

- C (don’t have a lot of experience with it, only having written small patches, but I’ve appreciated the readability of it in working with others’ code)

- Python (All the apps I wrote in Python that I still use I have since re-written in Bash, and found it much more appropriate for them)
- C# (mostly from doing game programming in Unity using Microsoft Visual Studio for a class)
- Perl (used it for one tiny project about eight years ago, but don't remember anything about it really)
- Java (I don't really get it, and found it frustrating when I tried it)
- C++ (poked around briefly trying to patch a couple of apps in it, no real understanding)

Part 2

Perl 6 is a programming language first released at the end of 2015. It was originally conceived as a revision of the Perl programming language, but because it evolved during the course of its development (begun in 2000), it was split from the older language, which will continue to be maintained separately. It uses a specification and test-suite driven language development process, rather than an implementation-driven process. Consequently, there is not a single official implementation, although the Rakudo implementation is currently the most complete implementation available, and is the only implementation that is currently available.

Perl 6 is a general purpose language that by default for modules is just-in-time compiled to bytecode, and for scripts is interpreted, although this can be changed. The language is implemented using a subset language called NQP, which is run by a virtual machine. There are a few virtual machine implementations available, including a native one, one running on the Java Virtual Machine, and one for JavaScript. The native one, "MoarVM", is the most complete of these, in that it has support for the most language features of the various implementations. [1] Perl 6 has many complex and capable programming constructs included with it. Because of this, it seems quite capable to me. The main detriment compared to other programming languages in general that it seems very module-centric, and that it is fairly a new language, and consequently that it does not have a substantial amount of documentation available for it. Another issue that bothers me specifically, is that it lacks strong facilities out of the box for use as a shell REPL environment and shell-style use for executing external shell commands and manipulating their output, although this is a problem shared by many languages that are not purpose-built as shell languages.

Perl 6 is a module-centric language, including its own package management and package repository. One problem caused by it being a module-centric language, exacerbated by it being a new language, is that packages for it are not available easily as native packages. This is also a common issue with languages that come with their own package managers, such as .NET's NuGet, Node.js's npm, and Python's pip. There are several other issues caused by languages using their own package managers, as well. Another major consequence of this is that there is no significant pressure on package maintainers to keep their packages up to date and interoperable, because any given app only needs to work with one version of each package on which it depends. That system, however, breaks down when an app depends on two packages, each of which depend on a conflicting version of the same package, and one wants to avoid having outdated versions of the packages installed on one's system: the standard model of an elegant one-minor-version-per-package used by system package managers is broken, and rendered unable to ship the app properly.

While being a new language could conceivably be a problem because of a lack of available libraries for a language, this problem is avoided by Perl 6 by including a system for interacting with "C, Python, Perl 5, and Ruby" [2, p. 2]. Perl 6 contains a wide range of capabilities and features. The feature that interested me the most is grammars. That is what made me choose to pursue learning about the language. Perl 6's grammars allow writing code that parses a language and returns an abstract syntax tree without having to manually write the parser: rather, one can write a set of rules that combine to express the syntax of the language. This provides a powerful facility for being able to handle problems of implementing languages and document formats, and consequently is something that interests me, as I have a document format that I would like to implement. The rules in Perl 6 are created using *patterns*, which are somewhat similar to regular expressions in other languages, and generally take their place in Perl 6, although Perl 5–style regular expressions are still available in Perl 6 in case they are needed.

In addition to these abilities, Perl 6 has many other features, as well, that contribute to it being effective and capable at solving a variety of other problems: rather than focusing exclusively on one problem domain at the expense of facility in others, it is a general-purpose language that provides a range of tools that can be used, combined, and adapted for various applications. [3] Its mascot is a butterfly named Camelia [3]. It integrates several techniques different from the way Perl works, such as using lazy evaluation. This creates some changes in the way the language operates compared to Perl, and consequently it is somewhat different, with some language constructs not being available in Perl 6, thus necessitating working around their absence. For example, the `wantsarray` construct from Perl is apparently not available in Perl 6, according to the language FAQ [4]. My lack of familiarity with Perl prevents my assessing thoroughly the extent of a problem that this causes. Perl 6 uses (limited-precision [5]) rational numbers by default, rather than floating point, so that decimals will follow the traditional rules of arithmetic rather than the slightly strange behavior of floating-point numbers (floating-point numbers will produce unexpected results in some cases when doing arithmetic on them [6]), but at the cost of performance. Floating-point numbers can be used if they are desired, though.

Perl 6 *grammars* are an extremely expressive tool for writing parsers for languages and document formats. [3] They are a way to specify the syntax of a language in such a way that given a grammar and a string to parse using it, the Perl 6 language can build an abstract syntax tree that can be used by code generators for compiling a language, or by other tools that can work with abstract syntax trees. A grammar is a set of patterns (which can be named, and reference each other) using which the input string is parsed. A Perl 6 grammar is a class, so it can be used as other classes. To allow better expressivity from grammars, *actions* are available (a type of class that allows the parse process to call its methods). [7] This allows hooking into the parse process with other code, to allow a wide range of uses outside the traditional scope of grammars.

Perl 6 seems to have good prospects ahead of it, being a new language. Now that it has been released, it will presumably see extensive testing, and consequent development, due to its being in more widespread use. In summary, it is a capable and flexible general-purpose language that, while being new and having some rough edges, is quite capable and useful.

Part 3

Basic syntactic structure, including statement terminators or separators, block structure, syntactic peculiarities, etc.

In Perl 6, blocks are typically denoted by braces (`{`, `}`) and can be nested. Statements are delimited by semicolons (`;`), but the semicolons can be omitted in some circumstances, when the end of a statement is unambiguous. A block can be preceded by `->`, with an interceding list of variables, called a *pointy block*, which acts similarly to an anonymous function. Parentheses can be used to denote semantic divisions within a given block. Perl 6 has five types of variables, denoted by one of four indicators called *sigils*, or by the absence of a sigil; these determine the default type of the variable, affect how assignment to the variable takes place, and possibly introduce a type constraint on the variable [3].

The units or levels of scope and the nature and type (runtime or compile-time) of name bindings within the different levels of scope.

By default, the scope of variables in Perl 6 is determined by the *declarator* with which the variable was declared (most commonly `my`). Perl 6 uses a variety of modifiers, called *twigils*, to alter the default scoping [3]. There are seven declarators and two declarator-style prefixes, and there are nine twigils (not counting the absence of a twigil) [3].

Primitive data types available, including range limitations or lack thereof.

Of the 260 built-in types in Perl 6, 47 of them are primitive types. All types are subtypes of `Mu` [3]. Things that can be numbers are subtypes of `Numeric`. Common number types are `Int`, `Num`, and `Rat`. `Int`s do not restrict the range of values they can hold, beyond only accepting integers. `Num`s usually hold either "an IEEE 754 64-bit floating point" [3] value, `Inf` (infinity, or a number too large to store in a `Num`), or `NaN` (not a number). `Rat`s are rational numbers, so they do not accumulate errors like floating-point numbers (in Perl 6, `Num`s) do. Their denominators are limited to 64 bits. If one desires a `Rat` that does not have a restriction on the values of its numerator and denominator, a `FatRat` can be used instead. In addition, the `Real` and `Complex` types are available. A `Real` can be created from the non-imaginary numeric types, and a `Complex` can be created from two `Real`s representing the number's real and imaginary parts [3].

Operators for primitive data types and their precedence and associativity.

Perl 6 has twenty-seven levels of precedence. All operators can be written as subroutines [3]. Custom operators are allowed, and can have their precedence specified relative to the built-in operators. Most operators can be overloaded. There are three types of associativity for unary operators: left-associative, right-associative, and non-associative. Binary operators add to these three chain-associative and list-associative. The position of operators relative to their operand(s) has five categories: prefix (preceding the operand), infix (between two operands), postfix (succeding the operand), circumfix (surrounding the operand — 'bracketing' operators), and postcircumfix (bracketing the second of two operands). These five categories, combined with the operator, can be written as subroutines, as in this example from the documentation: `circumfix:«[ ]»(<a b c>)`, being equivalent to `[<a b c>]`. Some operators can be created by combining other operators together, such as type coersion and assignment (`~` coerces to a string, and `=` assigns; combined, `≃` coerces the right operand to a string and assigns it to the left operand). There are also the `s///` and `S///` substitution operators, but the documentation is, as far as I could tell, unclear on which of the above types of operator they fall into, and does not state clearly whether the regular rules that apply to operators apply to them.

Part 4

1. Data types

Perl 6 includes a large amount of non-primitive types in the language itself: 33 compound types, 56 "domain-specific" types, and 113 exception types. This includes common compound types such as arrays, lists, and hashes, as well as various other types for a range of applications, such as for documentation, concurrent programming, and I/O for various operating systems. Perl 6 includes a large amount of capabilities within the language core itself. That allows the base language installation to be quite functional. It means that it is not necessary to install as many additional libraries to have a practical base of tools [3].

2. Standard library

Perl 6 does not itself have a specified standard library *per se*, but the Rakudo Star distribution (the primary way to obtain an implementation of the language) does include a variety of additional *modules* (46 of them) [8]. Rakudo Star includes tools for performance debugging and profiling, such as `debugger-ui-commandline`, `grammar-debugger` and `Grammar-Profiler-Simple`. It provides the `Terminal-ANSIColor` module for creating colored output from command-line programs, and `test-mock` and `Test-META` for creating unit tests. Rakudo Star provides tools for data interchange and serialization, in the form of seven modules for working with JSON, and one for XML, as well as a database interface module (`DBIlish`). For graphics, Rakudo Star comes with the `svg` and `svg-plot` modules. It also includes several such as `perl6-http-easy` and `perl6-http-status` for network interoperation. Perl 6 includes a package manager module, also, called `zef`, which provides a command-line interface for managing the packages installed in a system [8].

3. Semantics of expression evaluation

Perl 6 makes a distinction between "statements" and "expressions". Expressions in Perl 6 are statements that return a value. A program in Perl 6 semantically represents a list of statements (things for the computer to do). To make a statement act as an expression, the `do` keyword can be used, which will cause the statement given to represent its value in the given context. Perl 6 generally evaluates statements in the order in which they are presented in the program (following the natural control flow), although internally it will divide up statements to be run concurrently in multiple threads if the result is the same [3].

4. Type coercion

By default, Perl 6 when encountering a variable that does not have a specified type will convert it to the necessary type when it is used in a context where its current type is not applicable. To override this behavior, types can be specified for each object, allowing strict type checking. When a specific type is requested for a variable, assignments to that variable will not be coerced, and explicit casting is required [3].

5. Semantics of assignment statements

Perl 6 has complex assignment semantics, similar to Java, because it uses containers for some values, but values can also exist without containers. Containers are object-oriented wrappers around the native (native to the Perl 6 virtual machine) types. The documentation writes that "The assignment operator asks the container on the left to store the value on its right" [3]. Because of this, the actual effect of assigning a value to a name depends on the type of container that has been created by variable declaration. In practice, this is largely transparent to the user, and the context of a reference to a variable will affect whether the container is used or the value within the container is used. If the user wants control over this, the language provides some constructs for controlling these issues [3], but in my experience the defaults have generally worked well.

A list can contain containers or values. Because of this, it is not always possible to assign to list elements, because lists are immutable and values cannot be changed. However, if the items in a list are containers, they can be changed because the container itself will stay the same. That is because the containers themselves do not change; rather, what they point to changes. Unlike lists, arrays only hold containers. Consequently, their contents can be changed [3].

Part 5: Control constructs and subprograms

Control flow constructs in Perl 6

Perl 6 control flow takes the form of statements. Statements can be grouped into blocks using braces. A block can be treated as a statement, used alone on a logical line, in which case it will be executed, or it can be used as a literal anonymous subprogram. Keywords can be used to guide a program's control flow. In contexts where it would usually be treated as a literal, it can be prefaced with the `do` keyword to execute it instead. For conditional branching, `if` can be used instead, which uses a condition to determine whether to execute a given block. Along with `if`, `else` and `elsif` can be used to select other blocks to execute. There is also `unless`, which acts similar to `if` with the condition inverted. For testing "definedness rather than truth" [3], the `with`, `orwith`, and `without` statements can be used, corresponding to the `if`, `elsif`, and `unless` statements, respectively. `when` is similar to `if`, but exits the enclosing block, and does not resume with it after the selected block finishes. With the `when` keyword, the `default` keyword can be used to construct multi-way branches similar to a C switch statement. This is often done using the `given` statement, which can be used to select a topic variable for `when` and `default` statements. Using `proceed` and `succeed` statements, the behavior of `when` and `default` blocks can be controlled. For iteration, `loop` provides simple incremental or infinite looping, and `while` and `until` keywords can be used for other types of looping; the `for` keyword is also available for iterating over lists. Within a "loop or recursive routine" [3], the `once` keyword can be used to specify that a subblock of the loop body should only be executed once. The `next`, `last`, and `redo` keywords can be used to continue the loop without finishing the current iteration, to leave the loop without finishing the current iteration, and to start the current iteration of the loop body again. To return from a block, the `return` or `return-rw` keywords can be used (`return-rw` gives a mutable result). Within a block, the `fail` keyword can be used to throw an exception. Additionally, "phasers" can be used to restrict execution of subsets of a program to certain contexts, such as the `CATCH` phaser for when an exception is thrown, or various phasers for important points within a program such as entering and leaving blocks. [3]

Subprograms in Perl 6

In Perl 6, subprograms are declared using the `sub` keyword. For anonymous functions, a literal inline block is more concise in some contexts. To indicate that multiple dispatch should be allowed for a named subprogram, it must be declared with the `multi` keyword. The `proto` keyword can be used to specify a template for type signatures that all the `multi` subprograms sharing that name must match. Subprograms can be used as operators, or as *traits*, which "modify the behavior of a ... language object" [3]. Perl 6 also provides tools for re-dispatching, which is giving another subprogram the same arguments the current one was called with.

References

[1] Perl 6 project, "NQP — Not Quite Perl (6): README.pod: `https://github.com/perl6/nqp`," 31 May 2017. This is where I got information about how Perl 6 is implemented, and its portability.

[2] M. Lenz, *Perl 6 Fundamentals: A Primer with Examples, Projects, and Case Studies*. 2017. This book provides a general introduction to Perl 6. It seems quite accessible to me.

[3] Perl 6 Documentation, "Perl 6 documentation: `https://docs.perl6.org/`," 19 Sept. 2017. The official Perl 6 documentation is very helpful for getting an understanding of the language.

[4] Perl 6 Documentation, "FAQ: Frequently asked questions about Perl 6: `https://docs.perl6.org/language/faq`," 19 Sept. 2017. I used this article to learn some basic information about Perl 6.

[5] Hacker News, "Why hasn't Perl 6 taken off yet?: `https://news.ycombinator.com/item?id=12888784`," Nov. 2016. This is where I learned that Perl 6 Rationals aren't arbitrary-precision.

[6] Stack Overflow, "Floating point inaccuracy examples: `https://stackoverflow.com/questions/2100490/floating-point-inaccuracy-examples`," 2010–2013. A citation for the claim regarding floating points being a little odd.

[7] Perl 6 Documentation, "Grammars: `https://docs.perl6.org/language/grammars`," 19 Sept. 2017. This documentation page was

one of my primary sources for getting an understanding of the scope and capabilities of grammars in Perl 6.

[8] Rakudo organization on GitHub, "Rakudo Star: star/modules at master: `https://github.com/rakudo/star/tree/master/modules`," 2017.

[9] R. Holloway, "The must-have features for Perl 6: `https://opensource.com/life/16/9/perl-6-features`," 23 Sept. 2016. This article provided a resource for learning about some of the things that distinguish and differentiate Perl 6 from other languages.

[10] E. Miller, "A review of Perl 6: `https://www.evanmiller.org/a-review-of-perl-6.html`," 13 Aug. 2017. This article discusses Perl 6's merits and weaknesses.

[11] jnthnwrthngtn, "Grammar::Tracer and Grammar::Debugger: `https://perl6advent.wordpress.com/2011/12/02/grammartracer-and-grammardebugger/`," *Perl 6 Advent Calendar*, 2 Dec. 2011. Debugging grammars is hard out of the box.

[12] M. Lenz, "How to debug a Perl 6 grammar: `https://perlgeek.de/en/article/debug-a-perl-6-grammar`," 1 Sept. 2010. This is more information on debugging Perl 6 grammars.

[13] The Perl 6 Programming Language, "The Perl 6 programming language: `https://perl6.org/`," n.d. This home page for the Perl 6 language introduces it and its culture. Retrieved 20 Sept. 2017.

[14] L. Rosenfeld and A. B. Downey, “Think Perl 6,” 2017. I used this book as a supplementary resource for getting an overview of Perl 6.

[15] Wikipedia, “Perl 6: `https://en.wikipedia.org/wiki/Perl_6`,” I used Wikipedia’s article on Perl 6 as a way to find other articles and books to look at (as well as Fogler Library and Web search).

Part 3 Appendix: Tables — after [3]

Sigils

Sigil	**Type constraint**	**Default type**	**Assignment**
`$`	`Mu` (no constraint)	Any	item
`@`	`Positional`	Array	list
`%`	`Associative`	Hash	list
`&`	`Callable`	Callable	item
none (declared with `\`)		(does not create containers or enforce context)	

Declarators

Declarator	**Effect**
`my`	Introduces lexically scoped names
`our`	Introduces package-scoped names
`has`	Introduces attribute names
`anon`	Introduces names that are private to the construct
`state`	Introduces lexically scoped but persistent names
`augment`	Adds definitions to an existing name
`supersede`	Replaces definitions of an existing name
`temp` (prefix: not a declarator)	Restores a variable's value at the end of scope
`let` (prefix: not a declarator)	Restores a variable's value at the end of scope if the block exits unsuccessfully

Twigils

Twigil	Scope
(none)	Based only on declarator
`*`	Dynamic
`!`	Attribute (class member)
`?`	Compile-time variable
`.`	Method (not really a variable)
`<`	Index into match object (not really a variable)
`^`	Self-declared formal positional parameter
`:`	Self-declared formal named parameter
`=`	Pod variables
`~`	The sublanguage seen by the parser at this lexical spot

Built-in types

Category	Type	Description
class	`AST`	Abstract representation of a piece of source code
class	`Any`	Thing/object
class	`Block`	Code object with its own lexical scope
enum	`Bool`	Logical boolean
class	`CallFrame`	Capturing current frame state
role	`Callable`	Invocable code object
class	`Code`	Code object
class	`Complex`	Complex number
class	`ComplexStr`	Dual Value Complex number and String

class	`Cool`	Object that can be treated as both a string and number
class	`Date`	Calendar date
class	`DateTime`	Calendar date with time
role	`Dateish`	Object that can be treated as a date
class	`Duration`	Length of time
class	`FatRat`	Rational number (arbitrary-precision)
class	`Instant`	Specific moment in time
class	`Int`	Integer (arbitrary-precision)
class	`IntStr`	Dual Value Integer and String
class	`Junction`	Logical superposition of values
class	`Label`	Tagged location in the source code
class	`Macro`	Compile-time routine
class	`Method`	Member function
class	`Mu`	The root of the Perl 6 type hierarchy.
class	`Nil`	Absence of a value or a benign failure
class	`Num`	Floating-point number
class	`NumStr`	Dual Value Floating-point number and String
role	`Numeric`	Number or object that can act as a number
class	`ObjAt`	Unique identification for an object
class	`Parameter`	Element of a signature
class	`Proxy`	Item container with custom storage and retrieval
class	`Rat`	Rational number (limited-precision)
class	`RatStr`	Dual Value Rational number and String
role	`Rational`	Number stored as numerator and denominator

role	`Real`	Non-complex number
class	`Routine`	Code object with its own lexical scope and ‘return‘ handling
class	`Scalar`	A mostly transparent container used for indirections
class	`Signature`	Parameter list pattern
class	`Str`	String of characters
role	`Stringy`	String or object that can act as a string
class	`Sub`	Subroutine
class	`Submethod`	Member function that is not inherited by subclasses
class	`Variable`	Object representation of a variable for use in traits
class	`Version`	Module version descriptor
class	`Whatever`	Placeholder for an unspecified value/argument
class	`WhateverCode`	Code object constructed by Whatever-currying
class	`atomicint`	Integer (native storage at the platform’s atomic operation size)
class	`int`	Integer (native storage; machine precision)

Levels of precedence

Associativity	**Level of precedence**	**Examples**
N	Terms	`42 3.14 "eek" qq["foo"] $x :!verbose @$array`
L	Method postfix	`.meth .+ .? .* .() .[] .{} .<> .«» .:: .= .^ .:`
N	Autoincrement	`++ --`

R	Exponentiation	**
L	Symbolic unary	! + - ~ ? | || +^ ~^ ?^ ^
L	Dotty infix	.= .
L	Multiplicative	* / % %% +& +< +> ~& ~< ~> ?& div mod gcd lcm
L	Additive	+ - +| +^ ~| ~^ ?| ?^
L	Replication	x xx
X	Concatenation	~
X	Junctive and	&
X	Junctive or	| ^
L	Named unary	temp let
N	Structural infix	but does <=> leg cmp .. ..^ ^.. ^..^
C	Chaining infix	!= == < <= > >= eq ne lt le gt ge ~~ === eqv !eqv =~=
X	Tight and	&&
X	Tight or	|| ^^ // min max
R	Conditional	?? !! ff fff
R	Item assignment	= => += -= **= xx=
L	Loose unary	so not

X	Comma operator	`, :`
X	List infix	`Z minmax X X~ X* Xeqv ...`
R	List prefix	`print push say die map substr ... [+] [*] any Z=`
X	Loose and	`and andthen notandthen`
X	Loose or	`or xor orelse`
X	Sequencer	`⟸, ⟹, <⟸, ⟹>`
N	Terminator	`; {...}, unless, extra ), ], }`

Part 4 Appendix

The appendices are not included here due to length. Tables of the composite types, domain-specific types, and exception types are available in the Perl 6 documentation [3]. A list of modules included with Rakudo Star is available on request from this author, based on the information from [8].