

Perl 6: Project: Part 3

Elliot Chandler (Wallace)

10 October 2017

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, and statements are delimited by semicolons (`;`). 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.

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. (Perl 6 Documentation, 2017c) There are seven declarators and two declarator-style prefixes, and there are nine twigils (not counting the absence of a twigil). (Perl 6 Documentation, 2017c)

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**. (Perl 6 Documentation, 2017c) 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” 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 **Reals** representing the number's real and imaginary parts. (Perl 6 Documentation, 2017c)

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. (Perl 6 Documentation, 2017c) 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 (succeeding 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 coercion 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 is unclear about whether the regular rules that apply to operators apply to them.

Appendix: Tables — after Perl 6 Documentation, 2017c

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-carrying
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 \iff leg cmp^ ^.. ^ ..^

C	Chaining infix	\neq $=$ $<$ \leq $>$ \geq eq ne lt le gt ge \sim \equiv eqv !eqv \approx
X	Tight and	$\&\&$
X	Tight or	$\ \ \wedge\wedge\ //$ min max
R	Conditional	?? !! ff fff
R	Item assignment	$= \Rightarrow += -= **=$ 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	\Leftarrow , \Rightarrow , $<\Leftarrow$, $\Rightarrow>$
N	Terminator	; { ... }, unless, extra),], }

Appendix: Program listing

```

1  #!/usr/bin/env perl6
2
3  use v6.c;
4  use Test;
5
6  sub lex(Str $code → List) {
7      my Pair @finishedTokens;
8      my Str $token;
9      my Str $prevChar = "None";
10
11     for $code.split("", :skip-empty) → $char {
12         $_ := $char;
13
14         # Subroutines for when something interesting is found
15         (
16             sub continue( → Nil) {
17                 # Accepted an identifier-part
18                 $token ≈ $char;
19                 $prevChar = $char;
20                 next

```

```

21     }
22
23     sub push(Str $type → Nil) {
24         $_ := $type;
25         if $prevChar ~ /<:L + :N>/ {
26             # Found something non-identifier after an identifier,
27             #   so push the identifier
28             @finishedTokens.push("identifier" ⇒ "$token");
29             $token = ""
30         }
31         # Found a token
32         $token ≈ $char;
33         @finishedTokens.push($type ⇒ "$token");
34         $prevChar = $char;
35         $token = "";
36         next
37     }
38 );
39
40 # Rules for figuring out what the lexer is looking at
41 (
42     if $token [?] <tru fals> {
43         when "e" {
44             push 'bool_literal'
45         }
46     }
47     when /<:L + :N>/ {
48         when /<:N>/ && $prevChar !~ /<:L>/ {
49             fail "Expected an identifier or an operator."
50         }
51         default {
52             continue
53         }
54     }
55     if $_ [?] <( ) > {
56         push 'parenthesis'
57     }
58     when '!' {
59         push 'unary_oper'

```

```

60         }
61         when $_ =~ < & | \< \> > {
62             push 'binary_oper'
63         }
64         when /\s/ {
65             # Skip this space
66             next
67         }
68         default {
69             fail "Input character is not in the language: '" ~ $char ~ "'"
70         }
71     );
72 }
73
74 # If there aren't any more characters to consume
75 # but there is still a token, it's an identifier
76 if $token ne '' {
77     @finishedTokens.push('identifier' => $token)
78 }
79
80 return @finishedTokens
81 }
82
83 sub parse(List $tokens → Nil) {
84     my Pair @state;
85     my Pair @consumed;
86     my Pair @input = $tokens.clone;
87     my Pair $token = "" => "";
88     my Str $lexeme = "";
89
90     # Support subroutines for the parser
91     (
92         sub lexeme( → Pair) {
93             $_ = shift(@input);
94             say $_;
95             $lexeme = ~ $_;
96             unshift(@consumed, $_);
97             when "" => "" {
98                 # do nothing, we don't have any token yet

```

```

99         }
100     default {
101         say "Next token is the " ~ .key ~ " " ~ .value;
102         #say "    State: \n" ~ @state;
103         #say "    Consumed: \n" ~ @consumed;
104         return $_
105     }
106 }
107
108 sub enter(Str $rule → Nil) {
109     say "Enter <$rule>";
110     @state.push("    " x @consumed.elems ~ "<$rule>: " ⇒ "Lexeme: \{ $lexeme \}\n");
111     #say "    State: \n" ~ @state;
112     #say "    Consumed: \n" ~ @consumed;
113 }
114
115 sub give_back( → Nil) {
116     @state.pop();
117     say "Releasing tokens ";
118     for @consumed {
119         unshift(@input, (shift(@consumed)))
120     }
121     @consumed = < >;
122     #say "    State: \n" ~ @state;
123     #say "    Consumed: \n" ~ @consumed;
124 }
125 );
126
127 # Rules for the parser
128 (
129     sub bool_literal( → Nil) {
130         enter "bool_literal";
131         my Str $test where * eq "bool_literal";
132         $test = lexeme().key;
133         CATCH {
134             default {
135                 give_back;
136                 X::AdHoc.new(:payload<Did not match>).throw
137             }

```



```

138     }
139 }
140
141 sub relop( → Nil) {
142     enter "relop";
143     my Str $test where * [?] < \< \> >;
144     $test = lexeme().value;
145     CATCH {
146         default {
147             give_back;
148             X::AdHoc.new(:payload<Did not match>).throw
149         }
150     }
151 }
152
153 sub id( → Nil) {
154     enter "id";
155     lexeme().key eq "identifier";
156     CATCH {
157         default {
158             give_back;
159             X::AdHoc.new(:payload<Did not match>).throw
160         }
161     }
162 }
163
164 sub relation_expr( → Nil) {
165     enter "relation_expr";
166     id;
167     {
168         while relop() {
169             id
170         }
171         CATCH {
172             default {
173                 say "(Matched short relop)"
174             }
175         }
176     }

```

```

177         CATCH {
178             default {
179                 give_back;
180                 X::AdHoc.new(:payload<Did not match>).throw
181             }
182         }
183     }
184
185     sub bool_factor( → Nil) {
186         enter "bool_factor";
187         {
188             bool_literal();
189             CATCH {
190                 default {
191                     my Str $lexeme = lexeme().value;
192                     my $extest where * eq '!';
193                     $extest = $lexeme;
194                     bool_factor;
195                     CATCH {
196                         default {
197                             my $lptest where * eq '(';
198                             $lptest = $lexeme;
199                             $lexeme eq "(";
200                             bool_expr;
201                             my $rptest where * eq ')';
202                             $rptest = lexeme;
203                             CATCH {
204                                 default {
205                                     relation_expr
206                                 }
207                             }
208                         }
209                     }
210                 }
211             }
212         }
213     }
214     CATCH {
215         default {
216             give_back;

```

```

216             X::AdHoc.new(:payload<Did not match>).throw
217         }
218     }
219 }
220
221 sub and_term( → Nil) {
222     enter "and_term";
223     bool_factor;
224     while lexeme().value eq "&" {
225         bool_factor
226     }
227     CATCH {
228         default {
229             give_back;
230             X::AdHoc.new(:payload<Did not match>).throw
231         }
232     }
233 }
234
235 sub bool_expr( → Nil) {
236     enter "bool_expr";
237     and_term;
238     while lexeme().value eq "|" {
239         and_term
240     }
241     CATCH {
242         default {
243             give_back;
244             X::AdHoc.new(:payload<Did not match>).throw
245         }
246     }
247 }
248 );
249
250 # Enter the parser from the top of the tree
251 bool_expr;
252 if @input.elems > 0 {
253     fail "The input string does not match the grammar. Unused input: " ~ @input
254 }

```

```

255     say @state;
256 }
257
258 # Test suite
259 (
260     # Test lexer
261     nok lex('String qux?');
262     isa-ok lex('Stringqux'), List;
263     isa-ok lex('foo & !( a2 > bar & w < foo | x < y)'), List;
264
265     # Test parser
266     #lex('foo & !( a2 > bar & w < foo | x < y)')
267     #    ==> parse;
268     say parse(lex('foo & !( a2 > bar & w < foo | x < y)'));
269
270     say "Done running tests. Report:";
271     done-testing;
272 );

```

References

Hacker News (2016). Why hasn't perl 6 taken off yet? This is where I learned that Perl 6 Rationals aren't arbitrary-precision.

Holloway, R. (2016). The must-have features for Perl 6. This article provided a resource for learning about some of the things that distinguish and differentiate Perl 6 from other languages.

jnthnrwrthngtn (2011). Grammar::Tracer and Grammar::Debugger. *Perl 6 Advent Calendar*. Debugging grammars is hard out of the box.

Lenz, M. (2010). How to debug a Perl 6 grammar. This is more information on debugging Perl 6 grammars.

Lenz, M. (2017). *Perl 6 Fundamentals: A Primer with Examples, Projects, and Case Studies*. doi:10.1007/978-1-4842-2899-9. This book provides a general introduction to Perl 6. It seems quite accessible to me.

Miller, E. (2017). A review of Perl 6. This article discusses Perl 6's merits and weaknesses.

Perl 6 Documentation (2017a). FAQ: Frequently asked questions about Perl 6. I used this article to learn some basic information about Perl 6.

Perl 6 Documentation (2017b). Grammars. This documentation page was one of my primary sources for getting an understanding of the scope and capabilities of grammars in Perl 6.

Perl 6 Documentation (2017c). Perl 6 documentation. The official Perl 6 documentation is very helpful for getting an understanding of the language.

Perl 6 project (2017). This is where I got information about how Perl 6 is implemented, and its portability.

Rosenfeld, L. and Downey, A. B. (2017). Think Perl 6. I used this book as a supplementary resource for getting an overview of Perl 6.

Stack Overflow (2010–2013). Floating point inaccuracy examples. A citation for the claim regarding floating points being a little odd.

The Perl 6 Programming Language (n.d.). The Perl 6 programming language. This home page for the Perl 6 language introduces it and its culture. Retrieved 20 Sept. 2017.

Wikipedia. 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).