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. 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 (Perl 6 Documentation,

2017c).

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" (Perl 6 Documentation, 2017c) 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 (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 (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.

References

Hacker News (2016). Why hasn't Perl 6 taken off yet?: https://news.ycombinator.com/item?id=12888784. This is where I learned that Perl 6 Rationals aren't arbitrary-precision.

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

jnthnwrthngtn (2011). Grammar::Tracer and Grammar::Debugger: https://perl6advent.wordpress.com/2011/12/02/grammartracer-and-grammardebugger/. *Perl 6 Advent Calendar*. Debugging grammars is hard out of the box.

Lenz, M. (2010). How to debug a Perl 6 grammar: https://perlgeek.de/en/article/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: https://www.evanmiller.org/a-review-of-perl-6.html. This article discusses Perl 6's merits and weaknesses.

Perl 6 Documentation (2017a). FAQ: Frequently asked questions about Perl 6: https://docs.perl6.org/language/faq. I used this article to learn some basic information about Perl 6.

Perl 6 Documentation (2017b). Grammars: https://docs.perl6.org/language/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: https://docs.perl6.org/. The official Perl 6 documentation is very helpful for getting an understanding of the language.

Perl 6 project (2017). NQP — Not Quite Perl (6): README.pod: https://github.com/perl6/nqp. 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: https://stackoverflow.com/questions/2100490/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: https://perl6.org/. This home page for the Perl 6 language introduces it and its culture. Retrieved 20 Sept. 2017.

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).

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-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 ), ], }`

Appendix: Program listing

This program tries to parse a simple grammar, although in the end I was not able to get it to work.

```
#!/usr/bin/env perl6

use v6.c;
use Test;

sub lex(Str $code ⟶ List) {
    my Pair @finishedTokens;
    my Str $token;
    my Str $prevChar = "None";

    for $code.split("", :skip-empty) → $char {
        $_ := $char;

        # Subroutines for when something interesting is found
        (
            sub continue( ⟶ Nil) {
```

```
            # Accepted an identifier-part
            $token ≃ $char;
            $prevChar = $char;
            next
        }

        sub push(Str $type ⟶ Nil) {
            $_ := $type;
            if $prevChar ~~ /<:L + :N>/ {
                # Found something non-identifier after an identifier,
                #   so push the identifier
                @finishedTokens.push("identifier" ⇒ "$token");
                $token = ""
            }
            # Found a token
            $token ≃ $char;
            @finishedTokens.push($type ⇒ "$token");
            $prevChar = $char;
            $token = "";
            next
        }
    );

    # Rules for figuring out what the lexer is looking at
    (
        if $token ∈ <tru fals> {
            when "e" {
                push 'bool_literal'
            }
        }
        when /<:L + :N>/ {
            when /<:N>/ && $prevChar !~~ /<:L>/ {
                fail "Expected an identifier or an operator."
            }
            default {
                continue
            }
        }
        if $_ ∈ < ( ) > {
```

```
                push 'parenthesis'
            }
            when '!' {
                push 'unary_oper'
            }
            when $_ ? < & | \< \> > {
                push 'binary_oper'
            }
            when /\s/ {
                # Skip this space
                next
            }
            default {
                fail 'Input character is not in the language: "' ~ $char ~ '"'
            }
        );
    }

    # If there aren't any more characters to consume
    # but there is still a token, it's an identifier
    if $token ne '' {
        @finishedTokens.push('identifier' ⇒ $token)
    }

    @finishedTokens.push('EOF' ⇒ "");

    return @finishedTokens
}

sub parse(List $tokens ⟶ Nil) {
    my Pair @state;
    my Pair @consumed;
    my Pair @input = $tokens.clone;
    my Pair $token = "" ⇒ "";
    my Str $lexeme = "";
    my Str @currentRules = "";
    my Int $levelsCount = 0;

    # Support subroutines for the parser
```

```
95      (
96          sub lexeme( --> Pair) {
97              $_ = shift(@input);
98              #say $_;
99              $lexeme = ~ $_;
100             unshift(@consumed, $_);
101             when "" => "" {
102                 # do nothing, we don't have any token yet
103             }
104             default {
105                 say "Next token is the " ~ .key ~ " " ~ .value;
106                 return $_
107             }
108         }
109
110         sub enter(Str $rule --> Nil) {
111             say "Enter <$rule>";
112             @currentRules.push($rule);
113             $levelsCount = $levelsCount + 1;
114             @state.push("    " x $levelsCount ~ "<$rule>: " => "Lexeme: \{ $lexeme \}\n");
115         }
116
117         sub accept( --> Nil) {
118             say "Exit <" ~ @currentRules.pop() ~ ">";
119             $levelsCount = $levelsCount - 1;
120         }
121
122         sub give_back( --> Nil) {
123             @state.pop();
124             for @consumed {
125                 unshift(@input, (shift(@consumed)))
126             }
127             @consumed = < >;
128         }
129         sub failMatch( --> Nil) {
130             say "DEBUG: Did not match <" ~ @currentRules.pop() ~ "> (depth: $levelsCount)"
131             $levelsCount = $levelsCount - 1;
132             give_back;
133         }
```

```
134     );
135
136     # Rules for the parser
137     (
138         sub bool_literal( --> Nil) {
139             enter "bool_literal";
140             my Str $test where * eq "bool_literal";
141             $test = lexeme().key;
142             CATCH {
143                 default {
144                     failMatch;
145                     X::AdHoc.new(:payload<Did not match>).throw
146                 }
147             }
148             accept
149         }
150
151         sub relop( --> Nil) {
152             enter "relop";
153             my Str $test where * ∈ < \< \> >;
154             $test = lexeme().value;
155             CATCH {
156                 default {
157                     failMatch;
158                     X::AdHoc.new(:payload<Did not match>).throw
159                 }
160             }
161             accept
162         }
163
164         sub id( --> Nil) {
165             enter "id";
166             lexeme().key eq "identifier" or X::AdHoc.new(:payload<Did not match>).throw;
167             CATCH {
168                 default {
169                     failMatch;
170                     X::AdHoc.new(:payload<Did not match>).throw
171                 }
172             }
```

```
173             accept
174         }
175
176         sub relation_expr( --> Nil) {
177             enter "relation_expr";
178             id;
179
180             {
181                 if relop() {
182                     id
183                 }
184                 CATCH {
185                     default {
186                         say "DEBUG: Matched ID-only relation_expr";
187                         accept
188                     }
189                 }
190             }
191             CATCH {
192                 default {
193                     failMatch;
194                     X::AdHoc.new(:payload<Did not match>).throw
195                 }
196             }
197             accept
198         }
199
200         sub eof( --> Nil) {
201             lexeme().key eq "EOF" or X::AdHoc.new(:payload<Did not match>).throw;
202             CATCH {
203                 default {
204                     X::AdHoc.new(:payload<Did not match>).throw
205                 }
206             }
207         }
208
209         sub bool_factor( --> Nil) {
210             enter "bool_factor";
211             {
```

```
            bool_literal;
            CATCH {
                default {
                    my Str $lexeme = lexeme().value;
                    my $extest where * eq '!';
                    $extest = $lexeme;
                    bool_factor;
                    CATCH {
                        default {
                            my $lptest where * eq '(';
                            # [sic] — $lexeme here but lexeme in a few lines
                            $lptest = $lexeme;
                            bool_expr;
                            my $rptest where * eq ')';
                            $rptest = lexeme;
                            CATCH {
                                default {
                                    give_back;
                                    relation_expr;
                                    CATCH {
                                        default {
                                            failMatch;
                                            X::AdHoc.new(:payload<Did not match>).throw
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
            CATCH {
                default {
                    failMatch;
                    X::AdHoc.new(:payload<Did not match>).throw
                }
            }
            accept
```

```
251         }
252
253         sub and_term( --> Nil) {
254             enter "and_term";
255             bool_factor;
256             while lexeme().value eq "&" {
257                 bool_factor
258             }
259             CATCH {
260                 default {
261                     failMatch;
262                     X::AdHoc.new(:payload<Did not match>).throw
263                 }
264             }
265             accept
266         }
267
268         sub bool_expr( --> Nil) {
269             enter "bool_expr";
270             and_term;
271             while lexeme().value eq "|" {
272                 and_term
273             }
274             eof;
275             CATCH {
276                 default {
277                     failMatch;
278                     X::AdHoc.new(:payload<Did not match>).throw
279                 }
280             }
281             accept
282         }
283     );
284
285     # Enter the parser from the top of the tree
286     bool_expr;
287     CATCH {
288         default {
289             say "The input string does not match the grammar. Current parse state: ";
```

```
290             say @state;
291             fail "The input string does not match the grammar. Unused input: " ~ @input
292         }
293     }
294     if @input.elems > 0 {
295         say "The input string does not match the grammar. Current parse state: ";
296         say @state;
297         fail "The input string does not match the grammar. Unused input: " ~ @input
298     }
299     say @state;
300 }
301 
302 # Test suite
303 (
304     # Test lexer
305     say "Running lexer tests";
306     nok lex('String qux?');
307     isa-ok lex('Stringqux'), List;
308     say lex('foo & !( a2 > bar & w < foo | x < y)');
309     say lex('A1 & B1 | A2 & B1 | (! C | A <> B )');
310 
311     say "";
312     say "Starting parser test";
313     say "";
314 
315     # Test parser
316     lex('foo & !( a2 > bar & w < foo | x < y)')
317         ⟹ parse;
318         say "";
319         say "Starting second test";
320         say "";
321     lex('A1 & B1 | A2 & B1 | (! C | A <> B )')
322         ⟹ parse;
323 
324         say "";
325     say "Done running tests. Report:";
326     done-testing;
327 );
```

Appendix: Program listing

```
Running lexer tests
ok 1 -
ok 2 - The object is-a 'List'
[identifier ⇒ foo binary_oper ⇒ & unary_oper ⇒ ! parenthesis ⇒ (
identifier ⇒ a2 binary_oper ⇒ > identifier ⇒ bar binary_oper ⇒ &
identifier ⇒ w binary_oper ⇒ < identifier ⇒ foo binary_oper ⇒ |
identifier ⇒ x binary_oper ⇒ < identifier ⇒ y parenthesis ⇒ ) EOF
⇒ ]
[identifier ⇒ A1 binary_oper ⇒ & identifier ⇒ B1 binary_oper ⇒ |
identifier ⇒ A2 binary_oper ⇒ & identifier ⇒ B1 binary_oper ⇒ |
parenthesis ⇒ ( unary_oper ⇒ ! identifier ⇒ C binary_oper ⇒ |
identifier ⇒ A binary_oper ⇒ < binary_oper ⇒ > identifier ⇒ B
parenthesis ⇒ ) EOF ⇒ ]

Starting parser test

Enter <bool_expr>
Enter <and_term>
Enter <bool_factor>
Enter <bool_literal>
Next token is the identifier foo
DEBUG: Did not match <bool_literal> (depth: 4)
Next token is the identifier foo
Enter <relation_expr>
Enter <id>
Next token is the identifier foo
Exit <id>
Enter <relop>
Next token is the binary_oper &
DEBUG: Did not match <relop> (depth: 5)
DEBUG: Matched ID-only relation_expr
Exit <relation_expr>
Exit <bool_factor>
Exit <and_term>
```

```
Next token is the binary_oper &
Enter <bool_factor>
Enter <bool_literal>
Next token is the unary_oper !
DEBUG: Did not match <bool_literal> (depth: 3)
Next token is the unary_oper !
Enter <bool_factor>
Enter <bool_literal>
Next token is the parenthesis (
DEBUG: Did not match <bool_literal> (depth: 4)
Next token is the parenthesis (
Enter <bool_expr>
Enter <and_term>
Enter <bool_factor>
Enter <bool_literal>
Next token is the identifier a2
DEBUG: Did not match <bool_literal> (depth: 7)
Next token is the identifier a2
Enter <relation_expr>
Enter <id>
Next token is the identifier a2
Exit <id>
Enter <relop>
Next token is the binary_oper >
Exit <relop>
Exit <relation_expr>
Exit <bool_factor>
Next token is the identifier bar
Exit <and_term>
Next token is the binary_oper &
Next token is the identifier w
DEBUG: Did not match <bool_expr> (depth: 4)
Enter <relation_expr>
Enter <id>
Next token is the identifier bar
Exit <id>
Enter <relop>
Next token is the binary_oper &
DEBUG: Did not match <relop> (depth: 5)
```

```
74  DEBUG: Matched ID-only relation_expr
75  Exit <relation_expr>
76  Exit <bool_factor>
77  Exit <bool_factor>
78  Exit <bool_expr>
79  Next token is the binary_oper &
80  Enter <bool_factor>
81  Enter <bool_literal>
82  Next token is the identifier w
83  DEBUG: Did not match <bool_literal> (depth: 2)
84  Next token is the identifier w
85  Enter <relation_expr>
86  Enter <id>
87  Next token is the identifier w
88  Exit <id>
89  Enter <relop>
90  Next token is the binary_oper <
91  Exit <relop>
92  Exit <relation_expr>
93  Exit <bool_factor>
94  Next token is the identifier foo
95  Exit <>
96  Next token is the binary_oper |
97  Enter <and_term>
98  Enter <bool_factor>
99  Enter <bool_literal>
100 Next token is the identifier x
101 DEBUG: Did not match <bool_literal> (depth: 2)
102 Next token is the identifier foo
103 Enter <relation_expr>
104 Enter <id>
105 Next token is the identifier foo
106 Exit <id>
107 Enter <relop>
108 Next token is the binary_oper |
109 DEBUG: Did not match <relop> (depth: 3)
110 DEBUG: Matched ID-only relation_expr
111 Exit <relation_expr>
112 Exit <bool_factor>
```

```
Exit <and_term>
Next token is the binary_oper |
The input string does not match the grammar. Current parse state:
[    <bool_expr>:  ⇒ Lexeme: {  }
         <and_term>:  ⇒ Lexeme: {  }
                 <relation_expr>:  ⇒ Lexeme: { identifier      foo }
                     <id>:  ⇒ Lexeme: { identifier     foo }
         <bool_factor>:  ⇒ Lexeme: { binary_oper       & }
             <bool_factor>:  ⇒ Lexeme: { unary_oper    ! }
                 <bool_expr>:  ⇒ Lexeme: { parenthesis ( }
                     <and_term>:  ⇒ Lexeme: { parenthesis      ( }
                             <relation_expr>:  ⇒ Lexeme: { identifier
a2 }
                 <relation_expr>:  ⇒ Lexeme: { identifier      w }
                     <id>:  ⇒ Lexeme: { identifier     w }
         <relation_expr>:  ⇒ Lexeme: { identifier      w }
             <id>:  ⇒ Lexeme: { identifier     w }
             <relop>:  ⇒ Lexeme: { identifier  w }
 <and_term>:  ⇒ Lexeme: { binary_oper  | }
         <relation_expr>:  ⇒ Lexeme: { identifier      foo }
             <id>:  ⇒ Lexeme: { identifier     foo }
]

Starting second test

Enter <bool_expr>
Enter <and_term>
Enter <bool_factor>
Enter <bool_literal>
Next token is the identifier A1
DEBUG: Did not match <bool_literal> (depth: 4)
Next token is the identifier A1
Enter <relation_expr>
Enter <id>
Next token is the identifier A1
Exit <id>
Enter <relop>
Next token is the binary_oper &
DEBUG: Did not match <relop> (depth: 5)
```

```
DEBUG: Matched ID-only relation_expr
Exit <relation_expr>
Exit <bool_factor>
Exit <and_term>
Next token is the binary_oper &
Enter <bool_factor>
Enter <bool_literal>
Next token is the identifier B1
DEBUG: Did not match <bool_literal> (depth: 3)
Next token is the identifier B1
Enter <relation_expr>
Enter <id>
Next token is the identifier B1
Exit <id>
Enter <relop>
Next token is the binary_oper |
DEBUG: Did not match <relop> (depth: 4)
DEBUG: Matched ID-only relation_expr
Exit <relation_expr>
Exit <bool_factor>
Exit <bool_expr>
Next token is the binary_oper |
Exit <>
Next token is the identifier A2
Next token is the binary_oper &
The input string does not match the grammar. Current parse state:
[    <bool_expr>:  ⇒ Lexeme: {  }
         <and_term>:  ⇒ Lexeme: {  }
                 <relation_expr>:  ⇒ Lexeme: { identifier      A1 }
                     <id>:  ⇒ Lexeme: { identifier     A1 }
             <relation_expr>:  ⇒ Lexeme: { identifier  B1 }
                 <id>:  ⇒ Lexeme: { identifier B1 }
]

Done running tests. Report:
1..2
```