Chapter 6
Regexes and Grammars

Grammars in Perl 6 are the “next level” of the well-known regular expressions. Grammars let you create much more sophisticated text parsers. A new domain-specific language (DSL), language translator, or interpreter can be created without any external help, using only the facilities that Perl 6 offers with grammars.

Regexes

In fact, Perl 6 just calls regular expressions *regexes*. The basic syntax is a bit different from Perl 5, but most elements (such as quantifiers * or `+`) still look familiar. The `regex` keyword is used to build a regex. Let us create a regex for the short names of weekdays.

```
my regex weekday
    {[Mon | Tue | Wed | Thu | Fri | Sat | Sun]};
```

The square brackets are enclosing the list of alternatives.

You can use the named regex inside other regexes by referring to its name in a pair of angle brackets. To match the string against a regex, use the *smartmatch* operator (~~).

```
say 'Thu' ~~ m/<weekday>/;
say 'Thy' ~~ m/<weekday>/;
```

These two matches will print the following.

```
｢Thu｣
 weekday => ｢Thu｣
False
```

The result of matching is an object of the `Match` type. When you print it, you will see the matched substring inside small square brackets ｢…｣ .

Regexes are the simplest named constructions. Apart from that, rules and tokens exist (and thus, the keywords `rule` and `token`).

Tokens are different from rules first regarding how they handle *whitespaces*. In rules, whitespaces are part of the regexes. In tokens, whitespaces are just visual separators. We will see more about this in the examples below.

```
my token number_token { <[\d]> <[\d]> }
my rule number_rule { <[\d]> <[\d]> }
```

(Note that there is no need in semicolons after the closing brace.)

The `<[...]>` construction creates a character class. In the example above, the two-character string `42` matches with the `number_token` token but not with the `number_rule` rule.

```
say 1 if "42" ~~ /<number_token>/;
say 1 if "4 2" ~~ /<number_rule>/;
```

The `$/` object

As we have just seen, the smartmatch operator comparing a string with a regex returns an object of the `Match` type. This object is stored in the `$/` variable. It also contains all the matching substrings. To keep (catch) the substring a pair of parentheses is used. The first match is indexed as `0`, and you may access it as an array element either using the full syntax `$/[0]` or the shortened one: `$0`.

Remember that even the separate elements like `$0` or `$0` still contain objects of the `Match` type. To cast them to strings or numbers, coercion syntax can be used. For example, `~$0` converts the object to a string, and `+$0` converts it to an integer.

```
'Wed 15' ~~ /(\w+) \s (\d+)/;
say ~$0; # Wed
say +$1; # 15
```

Grammars

Grammars are the development of regular expressions. Syntactically, the grammar is defined similar to a class but using the keyword `grammar`. Inside, it contains `token`s and `rule`s. In the next section, we will be exploring the grammar in the examples.

Simple parser

The first example of the grammar application is on grammar for tiny language that defines an assignment operation and contains the printing instruction. Here is an example of a programme in this language.

```
x = 42;
y = x;
print x;
print y;
print 7;
```

Let's start writing the grammar for the language. First, we have to express the fact that a programme is a sequence of statements separated by a semicolon. Thus, at the top level the grammar looks like this:

```
grammar Lang {
    rule TOP {
        ^ <statements> $
    }
    rule statements {
        <statement>+ %% ';'
    }
}
```

Here, `Lang` is the name of the grammar, and `TOP` is the initial rule from which the parsing will be started. The rule's content is a regex surrounded by with a pair of symbols, `^` and `$`, to tie the rule to the beginning and the end of the text. In other words, the whole programme should match the `TOP` rule. The central part of the rule, `<statements>`, refers to another rule. Rules will ignore all the spaces between their parts. Thus, you may freely add spaces to the grammar definition to make it easily readable.

The second rule explains the meaning of `statements`. The `statements` block is a sequence of separate `statement`s. It should contain at least one `statement`, as the `+` quantifier requires, and the delimiter character is a semicolon. The separator is mentioned after the `%%` symbol. In grammar, this means that you must have the separator character between instructions, but you can omit it after the last one. If there's only one percent character instead of two, the rule will also require the separator after the last statement.

The next step is to describe the `statement`. At the moment, our language has only two operations: assignment and printing. Each of them accepts either a value or a variable name.

```
rule statement {
    | <assignment>
    | <printout>
}
```

The vertical bar separates alternative branches like it does in regular expressions in Perl 5. To make the code a bit better-looking and simplify the maintenance, an extra vertical bar may be added before the first subrule. The following two descriptions are identical:

```
rule statement {
      <assignment>
    | <printout>
}
rule statement {
    | <assignment>
    | <printout>
}
```

Then, let us define what do `assignment` and `printout` mean.

```
rule assignment {
    <identifier> '=' <expression>
}
rule printout {
    'print' <expression>
}
```

Here, we see literal strings, namely, `'='` and `'print'`. Again, the spaces around them do not affect the rule.

An `expression` matches with either an identifier (which is a variable name in our case) or with a constant value. Thus, an `expression` is either an `identifier` or a `value` with no additional strings.

```
rule expression {
    | <identifier>
    | <value>
}
```

At this point, we should write the rules for identifiers and values. It is better to use another method, named `token`, for that kind of the grammar bit. In tokens, the spaces matter (except for those that are adjacent to the braces).

An identifier is a sequence of letters:

```
token identifier {
    <:alpha>+
}
```

Here, `<:alpha>` is a predefined character class containing all the alphabetical characters.

A value in our example is a sequence of digits, so we limit ourselves to integers only.

```
token value {
    \d+
}
```

Our first grammar is complete. It is now possible to use it to parse a text file.

```
my $parsed = Lang.parsefile('test.lang');
```

If the file content is already in a variable, you may use the `Lang.parse($str)` method to parse it. (There is more about reading from files in the Appendix.)

If the parsing was successful, that if the file contains a valid programme, the `$parse` variable will contain an object of the `Match` type. It is possible to dump it (`say $parsed`) and see what's there.

```
 ｢x = 42;
y = x;
print x;
print y;
print 7;
｣
 statements =>  ｢x = 42;
y = x;
print x;
print y;
print 7;
｣
  statement =>  ｢x = 42｣
   assignment =>  ｢x = 42｣
    identifier =>  ｢x｣
    expression =>  ｢42｣
     value =>  ｢42｣
  statement =>  ｢y = x｣
   assignment =>  ｢y = x｣
    identifier =>  ｢y｣
    expression =>  ｢x｣
     identifier =>  ｢x｣
  statement =>  ｢print x｣
  printout => 「print x」
   expression => 「x」
    identifier => 「x」
 statement => 「print y」
  printout => 「print y」
   expression => 「y」
    identifier => 「y」
 statement => 「print 7」
  printout => 「print 7」
   expression => 「7」
    value => 「7」
```

This output corresponds to the sample programme from the beginning of this section. It contains the structure of the parsed programme. The captured parts are displayed in the brackets 「...」. First, the whole matched text is printed. Indeed, as the `TOP` rule uses the pair of anchors `^` ... `$`, and so the whole text should match the rule.

Then, the parse tree is printed. It starts with the `<statements>`, and then the other parts of the grammar are presented in full accordance with what the programme in the file contains. On the next level, you can see the content of both the `identifier` and `value` tokens.

If the programme is grammatically incorrect, the parsing methods will return an empty value `(Any)`. The same will happen if only the initial part of the programme matches the rules.

Here is the whole grammar for your convenience:

```
grammar Lang {
    rule TOP {
        ^ <statements> $
    }
    rule statements {
        <statement>+ %% ';'
    }
    rule statement {
        | <assignment>
        | <printout>
    }
    rule assignment {
        <identifier> '=' <expression>
    }
    rule printout {
        'print' <expression>
    }
    rule expression {
        | <identifier>
        | <value>
    }
    token identifier {
        <:alpha>+
    }
    token value {
        \d+
    }
}
```

An interpreter

So far, the grammar sees the structure of the programme and can tell if it is grammatically correct, but it does not execute any instructions contained in the programme. In this section, we will extend the parser so that it can actually execute the programme.

Our sample language uses variables and integer values. The values are constants and describe themselves. For the variables, we need to create a storage. In the simplest case, all the variables are global, and a single hash is required: `my %var;`.

The first action that we will implement now, is an assignment. It will take the value and save it in the variable storage. In the `assignment` rule in the grammar, an `expression` is expected on the right side of the equals sign. An expression can be either a variable or a number. To simplify the variable name lookup, let's make the grammar a bit more complicated and split the rules for assignments and printing out into two alternatives.

```
rule assignment {
    | <identifier> '=' <value>
    | <identifier> '=' <identifier>
}
rule printout {
    | 'print' <value>
    | 'print' <identifier>
}
```

Actions

The grammars in Perl 6 allow actions in response to the rule or token matching. Actions are code blocks that are executed when the corresponding rule or token is found in the parsed text. Actions receive an object `$/`, where you can see the details of the match. For example, the value of `$<identifier>` will contain an object of the `Match` type with the information about the substring that actually was consumed by the grammar.

```
rule assignment {
    | <identifier> '=' <value>
         {say "$<identifier>=$<value>"}
    | <identifier> '=' <identifier>
}
```

If you update the grammar with the action above and run the programme against the same sample file, then you will see the substring *`x=42`* in the output.

The `Match` objects are converted to strings when they are interpolated in double quotes as in the given example: `"$<identifier>=$<value>"`. To use the text value from outside the quoted string, you should make an explicit typecast:

```
rule assignment {
    | <identifier> '=' <value>
         {%var{~$<identifier>} = +$<value>}
    | <identifier> '=' <identifier>
}
```

So far, we've got an action for assigning a value to a variable and can process the first line of the file. The variable storage will contain the pair `{x => 42}`.

In the second alternative of the `assignment` rule, the `<identifier>` name is mentioned twice; that is why you can reference it as to an array element of `$<identifier>`.

```
rule assignment {
    | <identifier> '=' <value>
      {
          %var{~$<identifier>} = +$<value>
      }
    | <identifier> '=' <identifier>
      {
          %var{~$<identifier>[0]} =
          %var{~$<identifier>[1]}
      }
}
```

This addition to the code makes it possible to parse an assignment with two variables: `y = x`. The `%var` hash will contain both values: `{x => 42, y => 42}`.

Alternatively, capturing parentheses may be used. In this case, to access the captured substring, use special variables, such as `$0`:

```
rule assignment {
    | (<identifier>) '=' (<value>)
      {
           %var{$0} = +$1
      }
    | (<identifier>) '=' (<identifier>)
      {
           %var{$0} = %var{$1}
      }
}
```

Here, the unary ~ is no longer required when the variable is used as a hash key, but the unary `+` before `$1` is still needed to convert the `Match` object to a number.

Similarly, create the actions for printing.

```
rule printout {
    | 'print' <value>
      {
          say +$<value>
      }
    | 'print' <identifier>
      {
          say %var{$<identifier>}
      }
}
```

Now, the grammar is able to do all the actions required by the language design, and it will print the requested values:

```
42
42
7
```

As soon as we used capturing parentheses in the rules, the parse tree will contain entries named as `0` and `1`, together with the named strings, such as `identifier`. You can clearly see it when parsing the `y = x` string:

```
statement => ｢y = x｣
 assignment => ｢y = x｣
  0 => ｢y｣
   identifier => ｢y｣
  1 => ｢x｣
   identifier => ｢x｣
```

An updated parser looks like this:

```
my %var;

grammar Lang {
    rule TOP {
        ^ <statements> $
    }
    rule statements {
        <statement>+ %% ';'
    }
    rule statement {
        | <assignment>
        | <printout>
    }
    rule assignment {
        | (<identifier>) '=' (<value>)
          {
              %var{$0} = +$1
          }
        | (<identifier>) '=' (<identifier>)
          {
              %var{$0} = %var{$1}
          }
    }
    rule printout {
        | 'print' <value>
          {
              say +$<value>
          }
        | 'print' <identifier>
          {
              say %var{$<identifier>}
          }
    }
    token identifier {
        <:alpha>+
    }
    token value {
        \d+
    }
}

Lang.parsefile('test.lang');
```

For convenience, it is possible to put the code of actions in a separate class. This helps a lot when the actions are more complex and contain more than one or two lines of code.

To create an external action, create a class, which will later be referenced via the `:actions` parameter upon the call of the `parse` or `parsefile` methods of the grammar. As with built-in actions, the actions in an external class receive the `$/` object of the `Match` type.

First, we will train on a small isolated example and then return to our custom language parser.

```
grammar G {
    rule TOP {^ \d+ $}
}

class A {
    method TOP($/) {say ~$/}
}

G.parse("42", :actions(A));
```

Both the grammar `G` and the action class `A` have a method called `TOP`. The common name connects the action with the corresponding rule. When the grammar parses the provided test string and consumes the value of `42` by the `^ \d $` rule, the `A::TOP` action is triggered, and the `$/` argument is passed to it, which is immediately printed.

AST and attributes

Now, we are ready to simplify the grammar again after we split the `assignment` and `printout` rules into two alternatives each. The difficulty was that without the split, it was not possible to understand which branch had been triggered. You either needed to read the value from the `value` token or get the name of the variable from the `identifier` token and look it up in the variable storage.

Perl 6's grammars offer a great mechanism that is common in language parsing theory, the abstract syntax tree, shortened as AST.

First of all, update the rules and remove the alternatives from some of them. The only rule containing two branches is the `expression` rule.

```
rule assignment {
    <identifier> '=' <expression>
}
rule printout {
    'print' <expression>
}
rule expression {
    | <identifier>
    | <value>
}
```

The syntax tree that is built during the parse phase can contain the results of the calculations in the previous steps. The `Match` object has a field `ast`, dedicated especially to keep the calculated values on each node. It is possible to simply read the value to get the result of the previously completed actions. The tree is called abstract because how the value is calculated is not very important. What is important is that when the action is triggered, you have a single place with the result you need to complete an action.

The action can save its own result (and thus pass it further on the tree) by calling the `$/.make` method. The data you save there are accessible via the `made` field, which has the synonym `ast`.

Let’s fill the attribute of the syntax tree for the `identifier` and `value` tokens. The match with an identifier produces the variable name; when the value is found, the action generates a number. Here are the methods of the actions’ class.

```
method identifier($/) {
    $/.make(~$0);
}
method value($/) {
    $/.make(+$0);
}
```

Move one step higher, where we build the value of the expression. It can be either a variable value or an integer.

As the `expression` rule has two alternatives, the first task will be to understand which one matches. For that, check the presence of the corresponding fields in the `$/` object.

(If you use the recommended variable name `$/` in the signature of the action method, you may access its fields differently. The full syntax is `$/<identifier>`, but there is an alternative version `$<identifier>`.)

The two branches of the expression method behave differently. For a number, it extracts the value directly from the captured substring. For a variable, it gets the value from the `%var` hash. In both cases, the result is stored in the AST using the `make` method.

```
method expression($/) {
    if $<identifier> {
        $/.make(%var{$<identifier>});
    }
    else {
        $/.make(+$<value>);
    }
}
```

To use the variables that are not yet defined, we can add the defined-or operator to initialise the variable with the zero value.

```
$/.make(%var{$<identifier>} // 0);
```

Now, the expression will have a value attributed to it, but the source of the value is not known anymore. It can be a variable value or a constant from the file. This makes the `assignment` and `printout` actions simpler:

```
method printout($/) {
    say $<expression>.ast;
}
```

All you need for printing the value is to get it from the `ast` field.

For the `assignment`, it is a bit more complex but can still be written in a single line.

```
method assignment($/) {
    %var{$<identifier>} = $<expression>.made;
}
```

The method gets the `$/` object and uses the values of its `identifier` and `expression` elements. The first one is converted to the string and becomes the key of the `%var` hash. From the second one, we get the value by fetching the `made` attribute.

Finally, let us stop using the global variable storage and move the hash into the action class (we don't need it in the grammar itself). It thus will be declared as `has %!var;` and used as a private key variable in the body of the actions: `%!var{...}`.

After this change, it is important to create an instance of the actions class before paring it with a grammar:

```
Lang.parsefile(
    'test.lang',
    :actions(LangActions.new())
);
```

Here is the complete code of the parser with actions.

```
grammar Lang {
    rule TOP {
        ^ <statements> $
    }
    rule statements {
        <statement>+ %% ';'
    }
    rule statement {
        | <assignment>
        | <printout>
    }
    rule assignment {
        <identifier> '=' <expression>
    }
    rule printout {
        'print' <expression>
    }
    rule expression {
        | <identifier>
        | <value>
    }
    token identifier {
        (<:alpha>+)
    }
    token value {
        (\d+)
    }
}
```

```
class LangActions {
    has %var;

    method assignment($/) {
        %!var{$<identifier>} = $<expression>.made;
    }
    method printout($/) {
        say $<expression>.ast;
    }
    method expression($/) {
        if $<identifier> {
            $/.make(%!var{$<identifier>} // 0);
        }
        else {
            $/.make(+$<value>);
        }
    }
    method identifier($/) {
        $/.make(~$0);
    }
    method value($/) {
        $/.make(+$0);
    }
}

Lang.parsefile(
    'test.lang',
    :actions(LangActions.new())
);
```

Calculator

When considering language parsers, implementing a calculator is like writing a “Hello, World!” programme. In this section, we will create a grammar for the calculator that can handle the four arithmetical operations and parentheses. The hidden advantage of the calculator example is that you have to teach it to follow the operations priority and nested expressions.

Our calculator grammar will expect the single expression at a top level. The priority of operations will be automatically achieved by the traditional approach to grammar construction, in which the expression consists of terms and factors.

The terms are parts separated by pluses and minuses:

```
<term>+ %% ['+'|'-']
```

Here, Perl 6's `%%` symbol is used. You may also rewrite the rule using more traditional quantifiers:

```
<term> [['+'|'-'] <term>]*
```

Each term is, in turn, a list of factors separated by the symbols for multiplication or division:

```
<factor>+  %% ['*'|'/']
```

Both terms and factors can contain either a value or a group in parentheses. The group is basically another expression.

```
rule group {
    '(' <expression> ')'
}
```

This rule refers to the expression rule and thus can start another recursion loop.

It's time to introduce the enhancement of the `value` token so that it accepts the floating point values. This task is easy; it only requires creating a regex that matches the number in as many formats as possible. I will skip the negative numbers and the numbers in scientific notation.

```
token value {
    | \d+['.' \d+]*
    | '.' \d+
}
```

Here is the complete grammar of the calculator:

```
grammar Calc {
    rule TOP {
        ^ <expression> $
    }
    rule expression {
        | <term>+ %% $<op>=(['+'|'-'])
        | <group>
    }
    rule term {
        <factor>+  %% $<op>=(['*'|'/'])
    }
    rule factor {
        | <value>
        | <group>
    }
    rule group {
        '(' <expression> ')'
    }
    token value {
        | \d+['.' \d+]*
        | '.' \d+
    }
}
```

Note the `$<op>=(...)` construction in some of the rules. This is the named capture. The name simplifies the access to a value via the `$/` variable. In this case, you can reach the value as `$<op>`, and you don't have to worry about the possible change of the variable name after you update a rule as it happens with the numbered variables `$0`, `$1`, etc.

Now, create the actions for the compiler. At the `TOP` level, the rule returns the calculated value, which it takes from the `ast` field of the `expression`.

```
class CalcActions {
    method TOP($/) {
        $/.make: $<expression>.ast
    }
    . . .
}
```

The actions of the underlying rules `groups` and `value` are as simple as we've just seen.

```
method group($/) {
    $/.make: $<expression>.ast
}

method value($/) {
    $/.make: +$/
}
```

The rest of the actions are a bit more complicated. The `factor` action contains two alternative branches, just as the `factor` rule does.

```
method factor($/) {
    if $<value> {
        $/.make: +$<value>
    }
    else {
        $/.make: $<group>.ast
    }
}
```

Move on to the `term` action. Here, we have to take care of the list with its variable length. The rule's regex has the `+` quantifier, which means that it can capture one or more elements. Also, as the rule handles both the multiplication and the division operators, the action must distinguish between the two cases. The `$<op>` variable contains either the `*` or the `/` character.

This is how the syntax tree looks like for the string with three terms, `3*4*5`:

```
expression => 「3*4*5」
 term => 「3*4*5」
  factor => 「3」
   value => 「3」
  op => 「*」
  factor => 「4」
   value => 「4」
  op => 「*」
 factor => 「5」
  value => 「5」
```

As you can see, there are `factor` and `op` entries on the top levels. You will see the values as `$<factor>` and `$<op>` inside the actions. At least one `$<factor>` will always be available. The values of the nodes will already be known and available in the `ast` property. Thus, all you need to do is to traverse over the elements of those two arrays and perform either multiplication or division.

```
method term($/) {
    my $result = $<factor>[0].ast;

    if $<op> {
        my @ops = $<op>.map(~*);
        my @vals = $<factor>[1..*].map(*.ast);

        for 0..@ops.elems - 1 -> $c {
            if @ops[$c] eq '*' {
                $result *= @vals[$c];
            }
            else {
                $result /= @vals[$c];
            }
        }
    }

    $/.make: $result;
}
```

In this code fragment, the star character appears in the new role of a placeholder that tells Perl that it should process the data that it can get at this moment. It sounds weird, but it works perfectly and intuitively.

The @ops array with a list of the operation symbols consists of the elements that we got after stringifying the $<op>'s value:

```
my @ops = $<op>.map(~*);
```

The values themselves will land in the @vals array. To ensure that the values of the two arrays, @vals and @ops, correspond to each other, the slice of $<factor> is taken, which starts at the second element:

```
my @vals = $<factor>[1..*].map(*.ast);
```

Finally, the expression action is either to take the calculated value of group or to perform the sequence of additions and subtractions. The algorithm is close to the one of the term's action.

```
method expression($/) {
    if $<group> {
        $/.make: $<group>.ast
    }
    else {
        my $result = $<term>[0].ast;

        if $<op> {
            my @ops = $<op>.map(~*);
            my @vals = $<term>[1..*].map(*.ast);
            for 0..@ops.elems - 1 -> $c {
                if @ops[$c] eq '+' {
                    $result += @vals[$c];
                }
                else {
                    $result -= @vals[$c];
                }
            }
        }
         $/.make: $result;
    }
}
```

The majority of the code for the calculator is ready. Now, we need to read the string from the user, pass it to the parser, and print the result.

```
my $calc = Calc.parse(
               @*ARGS[0],
               :actions(CalcActions)
           );
say $calc.ast;
```

Let's see if it works.

```
$ perl6 calc.pl '39 + 3.14 * (7 - 18 / (505 - 502)) - .14'
42
```

It does.

On github.com/ash/lang, you can find the continuation of the code demonstrated in this chapter, which combines both the language translator and the calculator to allow the user write the arithmetical expressions in the variable assignments and the print instructions. Here is an example of what that interpreter can process:

```
x = 40 + 2;
print x;

y = x - (5/2);
print y;

z = 1 + y * x;
print z;

print 14 - 16/3 + x;
```