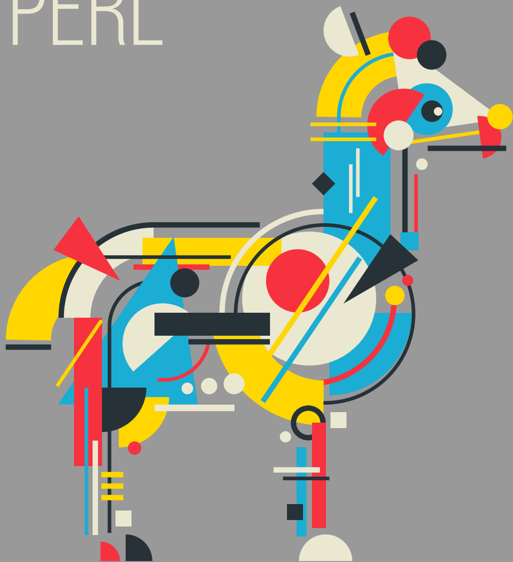
PRAGMATIC PERL

26



04/2015

pragmaticperl.com

Pragmatic Perl 26

pragmaticperl.com

Выпуск 26. Апрель 2015

Другие выпуски и форматы журнала всегда можно загрузить с pragmaticperl.com. С вопросами и предложениями пишите на почту editor@pragmaticperl.com.

Комментарии к каждой статье есть в html-версии. Подписаться на новые выпуски можно по ссылке pragmaticperl.com/subscribe.

Авторы статей: Андрей Шитов, Владимир Леттиев

Обложка: Марко Иванык

Корректор: Андрей Шитов

Выпускающий редактор: Вячеслав Тихановский

Ревизия: 2015-04-07 14:07

© «Pragmatic Perl»

Оглавление

1 От редактора 1

2 Анонс конференции YAPC::Russia 2015 2

3 Работа с WebSocket в Perl . . . 5

4 Промисы в Perl 6 25

5 Грамматики в Perl 6 41

6 Обзор CPAN за март 2015 г. . . 94

7 Интервью с Виктором Турским 111

1 От редактора

Совсем скоро (16 и 17 мая) пройдет конференция YAPC::Russia 2015 в Москве! В ней примут участие приглашенные гости. Приглашаем наших читателей.

Анонс мероприятия читайте в этом выпуске. Задавайте вопросы.

Друзья, журнал ищет новых авторов. Не упускайте такой возможности! Если у вас есть идеи или желание помочь, пожалуйста, свяжитесь с нами.

Приятного чтения.

■ *Вячеслав Тихановский*

2 Анонс конференции YAPC::Russia 2015

YAPC::Russia — ежегодная конференция, посвященная языку программирования Perl и его сообществу, которая проводится в Москве (под названием May Perl), Киеве (під назвою Perl Mova) и Санкт-Петербурге.

16 и 17 мая в московском офисе Mail.Ru Group пройдет конференция May Perl. На нее слетаются Perl-профессионалы из разных уголков земного шара, чтобы обменяться опытом, завести новые знакомства в своей сфере и пообщаться с единомышленниками в неформальной и приятной обстановке.

На предстоящем мероприятии ожидается около 30 докладов от спикеров, среди которых выступят специальные иностранные

гости:

Sawyer X

Программист из Израиля, один из разработчиков Dancer, организатор встреч сообществ TelAviv.pm и AmsterdamX.pm. Спикер давно увлечен Perl и активно следит за новостями в сообществе, а его харизматичные выступления отличаются простой и доступной подачей.

Интервью в журнале)

Peter Rabbitson (ribasushi)

Интервью в журнале

Perl-программист, разработчик DBIx::Class – одной из самых популярных ORM в мире Perl – и постоянный гость конференции.

Он занимает почетное место в сообществе и всегда готов помогать коллегам.

Мы приглашаем на May Perl докладчиков и участников. Чтобы выступить спикером, подайте заявку.

Для участия в качестве слушателя регистрируйтесь здесь. Участие бесплатное!

Официальные языки конференции – русский и английский. Будет организована онлайн-трансляция.

Мероприятие состоится по адресу: Москва, Ленинградский пр-т, 39, стр. 79 (м. «Аэропорт»).

3 Работа с WebSocket в Perl

Рассмотрены несколько подоходов при работе с технологией WebSocket из Perl

Технология WebSocket в современных браузерах уже широко поддерживается и используется во многих интернет-приложениях. В Perl поддержка WebSocket появилась еще на этапе разработки самого протокола во фреймворке Mojolicious. Затем появился модуль общего назначения Protocol::WebSocket и несколько оберток вокруг библиотек на других языках. На данный момент из-за сложности самого протокола в основном все приложения используют Protocol::WebSocket.

Что такое WebSocket?

Технология WebSocket позволяет установить двусторонний постоянный канал между клиентом и сервером используя HTTP-протокол. Вначале клиент посылает заголовок `Upgrade` и некоторый набор специальных заголовков, сервер отвечает своим набором заголовков и соединение устанавливается. Далее данные упаковываются в пакеты и отправляются. Кроме пакетов с данными существуют управляющие пакеты, например, для закрытия соединения.

```
GET /demo HTTP/1.1
Upgrade: WebSocket
Connection: Upgrade
Host: example.com
Cookie: foo=bar; alice=bob
Origin: http://example.com
Sec-WebSocket-Key:
```

```
    dGhlIHNhbXBsZSBub25jZQ==
8 Sec—WebSocket—Version: 13
9
10 HTTP/1.1 101 Switching Protocols
11 Upgrade: websocket
12 Connection: Upgrade
13 Sec—WebSocket—Accept:
    s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

Существует множество нюансов в зависимости от браузера, версии WebSocket, прокси-серверов, стоящих на пути запроса. Поэтому всегда стоит воспользоваться реализацией модуля Protocol::WebSocket, который обрабатывает все специальные случаи, скрывая подробности под простым интерфейсом.

Protocol::WebSocket позволяет написать как клиентскую, так и серверную часть. Рассмотрим низкоуровневую реализацию

серверной части с помощью AnyEvent.

Вначале напишем html-файл. После подключения к серверу посылаем ему сообщение, а получив сообщение, выводим его в консоль.

```
1 <!DOCTYPE html>
2 <meta charset="utf-8" />
3 <title>WebSocket Test</title>
4 <script language="javascript"
     type="text/javascript">
5     websocket = new WebSocket('ws
         ://localhost:3000');
6     websocket.onopen = function(
         evt) {
7          console.log('opened');
8          websocket.send('echo');
9     };
10    websocket.onclose = function(
         evt) { console.log('closed
         ') };
```

```
11     websocket.onmessage =
          function(evt) {
12         console.log('message=' +
              evt.data);
13     };
14     websocket.onerror = function(
          evt) { console.log('error'
          ) };
15 </script>
16 <h2>WebSocket Test</h2>
```

Затем напишем серверную часть:

```
1 #!/usr/bin/env perl
2
3 use strict;
4 use warnings;
5
6 use AnyEvent::Socket;
7 use AnyEvent::Handle;
8
9 use Protocol::WebSocket::
     Handshake::Server;
```

```
10 use Protocol::WebSocket::Frame;
11
12 my $cv = AnyEvent->condvar;
13
14 my $hdl;
15
16 AnyEvent::Socket::tcp_server
       undef, 3000, sub {
17         my ($clsock, $host, $port) =
               @_;
18
19         my $hs    = Protocol::
               WebSocket::Handshake::
               Server->new;
20         my $frame = Protocol::
               WebSocket::Frame->new;
21
22         $hdl = AnyEvent::Handle->new(
               fh => $clsock);
23
24         $hdl->on_read(
25             sub {
26                 my $hdl = shift;
```

```
27
28              my $chunk = $hdl->{
                    rbuf};
29              $hdl->{rbuf} = undef;
30
31              if (!$hs->is_done) {
32                  $hs->parse($chunk
                        );
33
34                  if ($hs->is_done)
                        {
35                      $hdl->
                            push_write
                            ($hs->
                            to_string)
                            ;
36                      return;
37                  }
38              }
39
40              $frame->append($chunk
                    );
41
```

```
42                 while (my $message =
                       $frame->next) {
43                     $hdl->push_write(
                           $frame->new(
                           $message)->
                           to_bytes);
44                 }
45             }
46         );
47 };
48
49 $cv->wait;
```

Все можно сильно упростить, если приложение будет запускаться в PSGI-среде. Protocol::WebSocket позволяет получить все необходимые заговки напрямую из окружения PSGI, например:

```
1 #!/usr/bin/env perl
2
3 use strict;
4 use warnings;
```

```
5
6 use AnyEvent::Handle;
7 use Protocol::WebSocket::
     Handshake::Server;
8 use Protocol::WebSocket::Frame;
9
10 my $psgi_app = sub {
11     my $env = shift;
12
13     my $fh = $env->{'psgix.io'}
         or return [500, [], []];
14
15     my $hs = Protocol::WebSocket
         ::Handshake::Server->
         new_from_psgi($env);
16     $hs->parse($fh) or return
         [400, [], [$hs->error]];
17
18     return sub {
19         my $respond = shift;
20
21         my $h = AnyEvent::Handle
             ->new(fh => $fh);
```

```
22          my $frame = Protocol::
               WebSocket::Frame->new;
23
24          $h->push_write($hs->
               to_string);
25
26          $h->on_eof(sub {});
27          $h->on_read(
28              sub {
29                  $frame->append($_
                       [0]->rbuf);
30
31                  while (my
                       $message =
                       $frame->next)
                       {
32                      $h->
                           push_write
                           (Protocol
                           ::
                           WebSocket
                           ::Frame->
                           new(
```

```
                         $message)
                         ->to_bytes
                         );
33                 }
34             }
35         );
36     };
37 };
38
39 $psgi_app;
```

Запустив это приложение под Twiggy или Feersum или каким-либо другим PSGI-сервером на AnyEvent, получим аналогичное предыдущему примеру поведение.

Написание низкоуровневых WebSocket приложений может быть несколько утомительно, рассмотрим высокоуровный подход к написанию WebSocket-клиента. В модуле Protocol::WebSocket находится утилита `wsconsole`, которая позволяет

очень удобно тестировать или отлаживать WebSocket-серверы. Далее представлен ее упрощенный вид:

```
1  #!/usr/bin/env perl
2
3  use strict;
4  use warnings;
5
6  use AnyEvent;
7  use AnyEvent::Socket;
8  use AnyEvent::Handle;
9  use Protocol::WebSocket::Client;
10
11 my $cv = AnyEvent->condvar;
12
13 my $client = Protocol::WebSocket
       ::Client->new(url => 'ws://
       localhost:3000');
14 my $ws_handle;
15
16 tcp_connect 'localhost', '3000',
       sub {
```

```
17      my ($fh) = @_ or return $cv->send("Connect failed: $!");
18
19      $ws_handle = AnyEvent::Handle->new(
20          fh     => $fh,
21          on_eof => sub {
22              $cv->send;
23          },
24          on_error => sub {
25              $cv->send;
26          },
27          on_read => sub {
28              my ($handle) = @_;
29
30              my $buf = delete $handle->{rbuf};
31
32              $client->read($buf);
33          }
34      );
35
```

```
36      $client->on(
37          write => sub {
38              my $client = shift;
39              my ($buf) = @_;
40
41              $ws_handle->
                   push_write($buf);
42          }
43      );
44      $client->on(
45          read => sub {
46              my $self = shift;
47              my ($buf) = @_;
48
49              print "message=$buf\n
                   ";
50          }
51      );
52      $client->connect;
53  };
54
55  my $stdin = AnyEvent::Handle->new
       (
```

```
56      fh       => \*STDIN,
57      on_read => sub {
58          my $handle = shift;
59
60          my $buf = delete $handle
                ->{rbuf};
61
62          $client->write($buf);
63      },
64      on_eof => sub {
65          $client->disconnect;
66
67          $ws_handle->destroy;
68          $cv->send;
69      }
70  );
71
72  $cv->wait;
```

В данном случае не нужно заботиться об обработке подключения, парсинге пакетов и прочего. Достаточно зарегистрировать

нужные события и обработать их.

Сама утилита `wsconsole` позволяет отправлять сообщения прямо из терминала, видеть полный формат пакетов, сообщать с какой версией подключаться. Так можно проверить, насколько хорошо и полно реализован WebSocket-сервер.

```
1 hello
2 > Writing
3 vvvvvvvvvv
4 [0000]   81 86 9E 17  E2 BC F6 72
       8E D0 F1 1D
     .... ...r ....
5
6 ^^^^^^^^^^
7 < Reading
8 vvvvvvvvvv
9 [0000]   81 06 68 65  6C 6C 6F 0A

     ..he llo.
```

Когда нет WebSocket

Не каждый браузер поддерживает WebSocket, иногда они специально отключаются, но все же приложение должно как-то работать. Есть несколько библиотек, которые имитируя интерфейс WebSocket при отсутствии последних переключаются на другие способы двусторонней передачи данных.

WebSocketJS

WebSocketJS представляет собой реализацию WebSocket с помощью технологии Flash. Исходный код библиотеки доступен на GitHub. Для полноценной работы Flash необходим запуск специального Policy-сервера отдельно или же можно воспользоваться специальным inline-сервером

Fliggy, который на Flash-специфичный запрос формирует нужный ответ, а в остальном работает как Twiggy.

Socket.IO

Socket.IO в зависимости от возможностей браузера использует различные подходы для реализации двусторонней связи: Polling, Streaming, Htmlfile и другие. Socket.IO в Perl реализован PSGI-приложением PocketIO. К сожалению, недавно выпущенная версия 1.0 не поддерживается PocketIO, в связи с сильными изменениями протокола. На сегодняшний день использование Socket.IO в Perl затруднительно.

SockJS

SockJS был написан как альтернатива Socket.IO. Данная библиотека также поддерживает несколько альтернативных WebSocket-технологий. Для Perl есть реализация SockJS-perl. Преимущество SockJS — в наличии удобных утилит для тестирования в независимости от языка реализации самого протокола.

Автор советует использовать именно эту библиотеку на сегодняшний день.

Альтернативы WebSocket

Когда нет необходимости в дуплексной связи, например, для уведомления пользователя или обновления новостной ленты,

можно воспользоваться технологией Server Side Events или EventSource. EventSource работает поверх HTTP и не требует значительной переработки серверной части. Реализация EventSource крайне проста и занимает несколько десятков строк, в этом можно убедиться, посмотрев на исходный код Plack-реализации EventSource.

■ *Вячеслав Тихановский*

4 Промисы в Perl 6

Вторая часть обзора возможностей Perl 6 для параллельных и конкуррентных вычислений

Промисы (promises, обещания) — это объекты, помогающие синхронизировать разные части параллельных процессов. Простейшее использование таких объектов — уведомить или узнать о том, сдержано ли данное ранее обещание, нарушено ли оно или пока неизвестно.

Базовые возможности

Объект создается вызововом `Promise.new`, а статус обещания доступен в методе `status`. Пока никаких действий не вы-

полнено, промис находится в состоянии Planned:

```
my $p = Promise.new;
say $p.status; # Planned
```

Метод keep переводит обещание в статус сдержанного (Kept):

```
$p.keep;
say $p.status; # Kept
```

Аналогично, обещание можно нарушить, вызвав метод break:

```
my $p1 = Promise.new;
say $p1.status; # Planned

$p1.break;
say $p1.status; # Broken
```

Вместо вызова метода status возможно преобразовать объект типа Promise в

булеву величину, вызвав метод `Bool` или воспользовавшись унарным оператором `?`:

```
1 say $p.Bool;
2 say ?$p;
```

Обратите внимание, что в этом случае возвращается только статус обещания (то есть либо еще не известно, сдержано оно или нарушено, либо уже известно), но не его результат.

Результат сообщает метод `result`, но с ним нужно быть осторожным. Если обещание сдержано, `result` возвращает истину:

```
1 my $p = Promise.new;
2 $p.keep;
3 say $p.result; # True
```

Обращение к `result` блокирует программу до тех пор, пока обещание не выйдет из ста-

туса Planned.

Если обещание нарушено:

```
1 my $p = Promise.new;
2 $p.break;
3 say $p.result;
```

то при вызове метода result возникает исключение:

```
1 $ perl6 promise3.pl
2 False
3   in method result at src/gen/m-
       CORE.setting:23096
4   in block <unit> at promise3.pl
       :3
```

Исключения возможно избежать, спрятав вызов внутрь блока try (но say тоже ничего не напечатает):

```
1 my $p = Promise.new;
```

```
2 $p.break;
3 try {
4     say $p.result;
5 }
```

Для нарушенных обещаний детали исключения можно получить, вызвав метод `cause` вместо `result`.

При вызове `keep` или `break` возможно передать параметр с сообщением, которое может быть или текстом, или объектом. В этом случае при вызове `result` вместо `True` и `False` (внутри исключения) будет возвращено переданное сообщение.

Фабричные методы

В классе `Promise` определены несколько интересных методов-фабрик, создающих промисы.

start

Метод `start` создает промис, внутри которого выполняется блок кода. Вместо явного вызова метода на классе `Promise.start` удобнее воспользоваться одноименным предопределенным ключевым словом:

```
my $p = start {
    42
}
```

(Точка с запятой после закрывающей скобки необязательна даже если после нее есть

другие инструкции.)

Результат выполнения кода будет результатом промиса. Если код привел к исключению, промис окажется нарушенным.

```
my $p = start {
    42
}
say $p.result; # 42
say $p.status; # Kept
```

Важно понимать, что создание блока `start` еще не означает, что код из него выполнен. Метод `start` сразу возвращает управление, поэтому если тут же попытаться узнать статус промиса, то результат может оказаться неверным. В предыдущем примере вызов `$p.result` блокировал выполнение программы до того момента, когда код из блока полностью выполнится, соответственно, результат будет содержать

сообщение, которое вернул блок, а статус изменится в состояние `Kept`.

Если поменять местами последние две строки, результат может оказаться другим. Для более воспроизводимого результата можно добавить паузу внутри блока:

```
my $p = start {
    sleep 1;
    42
}
say $p.status; # Planned
say $p.result; # 42
say $p.status; # Kept
```

Первый вызов `$p.status` происходит сразу после создания блока с промисом, и поэтому он еще не выполнен, так что статус оказывается `Planned`. А второй вывзов происходит уже после того, как метод `$p.result` дождался выполнения

блока кода.

Если же блок кода вызвал исключение, то промис окажется нарушенным:

```
my $p = start {
    die;
}
try {
    say $p.result;
}
say $p.status; # Эта строка выполнится и напечатает Broken
```

Вторая ловушка с блоком `start` — важно понимать, что именно вызывает исключение. Например, попытка деления на ноль приведет к исключению только в том случае, когда случится попытка воспользоваться результатом (это называется soft failure), а до тех пор Perl 6 удовлетворится тем, что результат деления на ноль — значение типа

`Rat`.

```
# $p1 будет Kept
my $p1 = start {
    my $inf = 1 / 0;
}

# $p2 окажется Broken
my $p2 = start {
    my $inf = 1 / 0;
    say $inf;
}
```

in и at

Методы `Promise.in` и `Promise.at` создают промисы, которые будут сдержаны через заданное число секунд или к указанному времени.

Например:

```
my $p = Promise.in(3);

for 1..5 {
    say $p.status;
    sleep 1;
}
```

Эта программа напечатает следующее (то есть промис оказывается выполненным через три секунды):

```
Planned
Planned
Planned
Kept
Kept
```

anyof и allof

Методы `Promise.anyof` и `Promise.allof` создают новые промисы, которые будут сдержаны только тогда, когда будет сдержан хотя бы один из (для `anyof`) или все (для `allof`) промисы, указанные при создании.

Один из полезных вариантов применения, приведенный в документации, — реализация завершения по таймауту для вычислений, котоыре могут занять длительное время:

```
1 my $code = start {
2     sleep 5
3 }
4 my $timeout = Promise.in(3);
5
6 my $done = Promise.anyof($code,
    $timeout);
```

```
say $done.result;
```

Этот код в теории должен завершиться после истечения таймаута в три секунды: в этот момент промис `$timeout` окажется выполненным, поэтому выполненным сразу же станет и промис `$done`.

В Rakudo Start 2015.03 этот код работает не как ожидается (он ждет пять секунд и сообщает о том, что `$code` выполнен), и на IRC-канале предложили обходное решение вместо `Promise.in(3)`:

```
my $timeout = start {
    sleep 3
}
```

then

Метод `then`, вызванный на уже определенном промисе, создает еще один, код которого будет вызван после того, как исходный промис будет сдержан или нарушен.

```
1 my $p = Promise.in(2);
2 my $t = $p.then({say "OK"}); #
     Напечатается через две секунды
3 say "promissed"; # Печатается
     сразу
4 sleep 3;
5 say "done";
```

Другой пример, в котором обещание сдержать не удалось:

```
1 Promise.start({  # Новый промис.
2     say 1 / 0    # Исключение.
3 }).then({        # Код,
     выполняемый после поломки.
4     say "oops"
```

```
5 }).result          # Необходимо,
    чтобы дождаться выполнения.
```

Пример

В заключение приведу пример реализации сотрировки типа sleep sort на промисах.

```
1 my @promises;
2 for @*ARGS -> $a {
3     @promises.push(start {
4         sleep $a;
5         say $a;
6     })
7 }
8
9 await(|@promises);
```

Программа принимает значения с командной строки:

```
1 $ perl6 sleep-sort.pl 3 7 4 9 1 6
      2 5
```

и создает для каждого из них промис, печатающий значение после необходимой задержки. В последней строке вызвана команда `await`, которая (аналогично методу `Promise.allof`) будет дожидаться выполнения всех промисов.

Продолжение следует

В следующих выпусках журнала я хочу рассказать о других возможностях Perl 6 для параллельной обработки. А пока предлагаю посмотреть доклад Патрика Мишо Parallelism in Perl 6.

■ *Андрей Шитов*

5 Грамматики в Perl 6

В этой статье рассказано об одной из наиболее мощных возможностей Perl 6 — грамматиках

Грамматики в Perl 6 — развитие темы регулярных выражений. Они позволяют создавать сколь угодно сложные парсеры текстов, поэтому не выходя за рамки Perl 6 вполне получится создать и свой DSL, и свой транслятор или интерпретатор другого языка. В этой статье мы попытаемся сделать подход к этой сложной задаче и разобраться, что к чему.

Введение

Источники

Подробная документация содержится в документах Synopsis 5: Regexes and Rules и Grammars. Полезно также ознакомиться с методами классов Grammar и Match. Официальные тесты, которые могут служить неплохими примерами кода, находятся в каталоге `t/spec/S05-grammar` и `t/spec/S05-match`. Пример создания грамматики для чтения JSON можно изучить в исходниках модуля JSON::Simple. Наконец, многое можно почерпнуть из доклада «Perl 6: what can you do today?» (доступна и видеозапись).

Терминология

В Perl 6 вместо термина *регулярные выражения* официально закреплено слово *регекс (regex)*. Синтаксис новых регексов заметно отличается от того, что было в Perl 5, однако многие элементы (например, квантификаторы `*` или `+` выглядят знакомо). Регексы создают с помощью ключевого слова `regex`:

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

Квадратные скобки здесь служат для группировки.

Созданный регекс теперь доступен по имени внутри других рекегсов. Чтобы сослаться, достаточно записать его имя в угловых скобках:

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

Эти два сопоставления выведут следующее:

```
｢Thu｣
 weekday => ｢Thu｣
False
```

Резутат сопоставления — объект типа `Match`. Если его напечатать, то совпавшие подстроки будут отмечены скобками вида `｢...｣`.

Кроме простых регексов существуют правила и токены (соответственно, ключевые слова `rule` и `token`).

Отличие токенов от правил прежде всего в том, что внутри правил автоматически обрабатываются пробельные символы

(whitespaces) в сопоставляемой строке, а в токенах они являются частью регекса. Это будет видно на примерах далее.

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

Конструкция `<[...]>` создает символьный класс. В этом примере строка `42` совпадет с токеном `number_token`, но не совпадет с правилом `number_rule`. И наоборот, строка `4 2` успешно совпадет только с правилом, но не с токеном.

Объект $/

Объект типа `Match`, который создается в результате сопоставления с регексом,

будет помещен в переменную `$/`. Из него же можно достать совпавшие подстроки. Синтаксис для захвата совпавших строк — круглые скобки. Нумерация ведется начиная с нуля. Доступ к совпавшим элементам осуществляется либо как к элементам массива: `$/[0]`, либо с помощью эквивалентной сокращенной записи: `$0`.

При этом важно помнить, что в этих элементах окажется элемент типа `Match`. Чтобы получить необходимое текстовое или числовое значение, следует явно указать контекст, например: `~$0` или `+$0`.

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

Грамматики

Следующий уровень организации регексов — грамматики. Грамматика аналогична классу с той лишь разницей, что она объявляется ключевым словом `grammar` и содержит правила и токены вместо методов. Пример — уже в следующем разделе.

Простой парсер

Первый пример реальной грамматики будет для примитивного языка, в котором определены простейшие операции присваивания и печати переменных. Вот такие:

```
1 x = 42;
2 y = x;
3 print x;
```

```
print y;
print 7;
```

Начнем создавать грамматику для этого мини-языка. Прежде всего, программа на таком языке — это последовательность инструкций (statements), разделенных точкой с запятой. Поэтому на верхем уровне грамматика будет выглядеть так:

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

`Lang` — название грамматики, `TOP` — начальное правило, именно с него будет начинаться разбор. Правило начинается

символом ^, обозначающим начало текста, и заканчивается символом $, который должен совпасть с окончанием программы. (В правилах все пробелы обрабатываются интуитивно правильно, то есть в начале программы, например, может быть любое число пробельных символов.)

То, что стоит между ^ и $, целиком вынесено в отдельное правило `<statements>`. В свою очередь, оно представляет собой последовательность как минимум одной (+) инструкции: `<statement>+`, причем они должны разделяться точкой с запятой. Символ-разделитель указан после двух процентов. Если бы процент был один, то любая инструкция была бы должна заканчиваться точкой с запятой. А два процента позволяют не ставить точку с запятой после последней инструкции.

Далее необходимо расписать то, что является инструкцией. В мини-языке на нынешней стадии определены две операции: присваивания и печати. Причем, каждая из них может принимать как число, так и другую переменную.

```
1 rule statement {
2     | <assignment>
3     | <printout>
4 }
```

Вертикальная черта обозначает выбор как и в обычных регулярных выражениях Perl 5, причем для красоты можно ставить еще одну черту перед первым вариантом. Иными словами, следующие два определения правила работают одинаково:

```
1 rule statement {
2       <assignment>
3     | <printout>
4 }
```

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

Далее определены правила для присваивания и печати.

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

Здесь вновь встречаются литеральные выражения — строки `'='` и `'print'`, а наличие или отсутствие пробелов вокруг них в разбираемой программе не имеет значения.

Правило `expression` должно совпадать и с переменной, и с числом, то есть это буквально является либо тем, либо другим:

```
1 rule expression {
2     | <identifier>
3     | <value>
4 }
```

Осталось расшифровать, что является именем переменной (идентификатором) и числом. Для этого вместо правил лучше создать токены — в них пробелы должны быть указаны явно (пробелы около фигурных скобок, обрамляющих описание токена, не учитываются).

Идентификатором сделаем последовательность буквенных символов:

```
1 token identifier {
2     <:alpha>+
3 }
```

Здесь `<:alpha>` — предопределенный символьный класс, совпадающий с буквами.

А значениями могут быть только целые числа, то есть последовательности цифр:

```
1 token value {
2     \d+
3 }
```

Грамматика готова. Теперь с ее помощью можно разобрать тестовый файл:

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

(Для разбора текста, содержащегося в переменной, в классе грамматики существует метод `parse($str)`.)

Если программа в файле записана в соответствии с правилами грамматики `Lang`, в

переменной `$parsed` окажется объект типа `Match`, который можно распечатать (`say $parsed`) и посмотреть, как была разобрана исходная программа:

```
1 ｢x = 42;
2 y = x;
3 print x;
4 print y;
5 print 7;
6 ｣
7  statements => ｢x = 42;
8 y = x;
9 print x;
10 print y;
11 print 7;
12 ｣
13   statement => ｢x = 42｣
14    assignment => ｢x = 42｣
15     identifier => ｢x｣
16     expression => ｢42｣
17      value => ｢42｣
18   statement => ｢y = x｣
```

```
19     assignment => 「y = x」
20      identifier => 「y」
21      expression => 「x」
22       identifier => 「x」
23    statement => 「print x」
24     printout => 「print x」
25      expression => 「x」
26       identifier => 「x」
27    statement => 「print y」
28     printout => 「print y」
29      expression => 「y」
30       identifier => 「y」
31    statement => 「print 7」
32     printout => 「print 7」
33      expression => 「7」
34       value => 「7」
```

Этот вывод содержит структуру разобранной программы, а совпавшие для каждого правила или токена строки видны в скобках 「...」. Сначала показано совпадение, содержащее весь текст программы (а иначе

не могло быть, потому что в главном правиле явно указано `^ ... $`). А затем, начиная со `<statements>`, следует дерево разбора, сначала на отдельные инструкции, а затем вглубь до токенов `identifier` или `value`.

Если программа окажется грамматически неверной, то результатом будет пустое значение: `(Any)`. То же самое произойдет, если с грамматикой совпадет не весь файл, а только его начало.

Полная грамматика на текущий момент выглядит так:

```
1 grammar Lang {
2     rule TOP {
3         ^ <statements> $
4     }
5     rule statements {
6         <statement>+ %% ';'
7     }
```

```
 8      rule statement {
 9          | <assignment>
10          | <printout>
11      }
12      rule assignment {
13          <identifier> '=' <
                expression>
14      }
15      rule printout {
16          'print' <expression>
17      }
18      rule expression {
19          | <identifier>
20          | <value>
21      }
22      token identifier {
23          <:alpha>+
24      }
25      token value {
26          \d+
27      }
28  }
```

Простой интерпретатор

Сейчас программа успешно разбирается, можно увидеть ее структуру, но никаких реальных действий, описанных в программе, не выполняется. В этом разделе мы научим парсер не только разбирать текст, но и выполнять нужные операции.

Нынешний мини-язык работает с переменными и числами. Числа являеются константами и описывают сами себя. А для хранения значений переменных необходимо создать хранилище. В простейшем случае это хеш:

```
my %var;
```

Каждый элемент хеша будет содержать пару имя переменной — ее значение. Областей видимости у нас пока нет.

Действие, которое помещает значение в переменную (и одновременно создает ее при первом упоминании), — `assignment`. В описанной выше грамматике справа от знака равенства ожидается выражение (`expression`), которое может быть либо другой переменной, чье значение придется отыскать, либо числом. Чтобы упростить поиск переменных в этом месте, немного усложним грамматику и распишем правила в виде двух альтернатив, избавившись, таким образом, от правила `expression`:

```
1 rule assignment {
2     | <identifier> '=' <value>
3     | <identifier> '=' <
        identifier>
4 }
5 rule printout {
6     | 'print' <value>
7     | 'print' <identifier>
8 }
```

Грамматика позволяет описывать действия на каждое правило и на каждый вариант, который она способна разобрать. Действия — это блоки кода, внутри которых доступен объект `$/`, а к совпавшим подобъектам можно напрямую обращаться, используя имя. Например, `$<identifier>` будет содержать объект типа `Match`, совпавший с одноименным правилом или токеном внутри другого правила.

```
1 rule assignment {
2     | <identifier> '=' <value>
              {say "$<identifier>=
        $<value>"}
3     | <identifier> '=' <
        identifier>
4 }
```

При интерполяции `"$<identifier>=$<value>"` объекты преобразуются к строке, что в этом примере однозначно дает имя

переменной. Вне строк в двойных кавычках, однако, преобразование следует делать явно:

```
1 rule assignment {
2     | <identifier> '=' <value>
              {%var{~$<identifier
        >} = +$<value>}
3     | <identifier> '=' <
        identifier>
4 }
```

Итак, создано действие, выполняемое для присваивания значения переменной. То есть из исходной тестовой программы будет работать строка `x = 42;`.

Во втором варианте правила `assignment` имя `<identifier>` встречается дважды, поэтому ссылаться на него как `$<identifier>` уже не получится, поскольку там окажется список. Но достаточно ука-

зать индекс элемента:

```
1 rule assignment {
2     | <identifier> '=' <value>
            {%var{~$<identifier
       >} = +$<value>}
3     | <identifier> '=' <
       identifier>  {%var{~$<
       identifier>[0]} = %var{~$<
       identifier>[1]}}
4 }
```

Теперь удалось выполнить действия для строки `y = x`.

Помимо использования имен возможно расставить захватывающие круглые скобки и использовать переменные типа `$0`:

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

```
        identifier>)  {%var{$0} =
        %var{$1}}
4 }
```

Унарный ~ при использовании переменной в качестве ключа хеша тоже можно опустить. (Но если не поставить плюс в +$1, то вместо числа в переменной окажется объект Match.)

Аналогично записываются действия для печати:

```
1 rule printout {
2     | 'print' <value>       {say +
        $<value>}
3     | 'print' <identifier> {say %
        var{$<identifier>}}
4 }
```

Все строки тестовой программы сейчас могут быть распознаны и обработаны. После

присваиваний в хеше `%var` окажется следующее:

```
x => 42, y => 42
```

А в результате вызова метода `Lang.parsefile` напечатается результат работы тестовой программы из файла `test.lang`:

```
42
42
7
```

Еще раз посмотрим на грамматику целиком, а заодно и на всю программу:

```
my %var;

grammar Lang {
    rule TOP {
        ^ <statements> $
    }
    rule statements {
```

```
8        <statement>+ %% ';'
9    }
10   rule statement {
11       | <assignment>
12       | <printout>
13   }
14   rule assignment {
15       | (<identifier>) '=' (<value>)       {%var{$0} = +$1}
16       | (<identifier>) '=' (<identifier>)  {%var{$0} = %var{$1}}
17   }
18   rule printout {
19       | 'print' <value>      {say +$<value>}
20       | 'print' <identifier> {say %var{$<identifier>}}
21   }
22   token identifier {
23       <:alpha>+
```

```
24      }
25      token value {
26          \d+
27      }
28 }
29
30 Lang.parsefile('test.lang');
```

Обратите внимание: после того, как в правиле появились круглые скобки, в объекте с разобранным исходным текстом появились поля с номерами `0` и `1`. Именованные поля (`identifier` и др.) тоже сохраняются. Это хорошо видно на примере разбора конструкции `y = x`:

```
1 statement => ｢y = x｣
2  assignment => ｢y = x｣
3   0 => ｢y｣
4    identifier => ｢y｣
5   1 => ｢x｣
6    identifier => ｢x｣
```

Действия (actions)

В предыдущем примере действия, выполняемые в ответ на совпадение, были оформлены в виде блока прямо внутри правила или токена. Это удобно для простых грамматик с простыми действиями, однако в Perl 6 предусмотрена возможность вынести все действия в отдельный класс.

Действия, соответствующие правилам грамматики, должны быть оформлены в виде одноименных методов класса. При вызове метода ему передается объект `$/` типа `Match`, содержащий текущий разобранный фрагмент.

Методы `parse` и `parsefile` ожидают инстанс класса с действиями в параметре `:actions`:

```
grammar G {
    rule TOP {^ \d+ $}
}
class A {
    method TOP($/) {say ~$/}
}
G.parse("42", :actions(A));
```

В этом примере и в грамматике `G`, и в классе `A` (от слова Actions) определены правило и метод с именем `TOP`. Далее при разборе текста `"42"` происходит удачное поглощение всего текста правилом `^ \d $`, после чего вызывается метод `A::TOP`. Единственный аргумент `$/` выводится на печать после явного (унарным оператором `~`) преобразования в строку. В данном случае тот же результат даст и преобразование в число: `say +$/`.

AST и атрибуты

В предыдущем примере для разбора программы на мини-языке было исключено правило `expression`, из-за чего правила `assignment` и `printout` пришлось усложнить, перечислив в каждом из них альтернативы для всех возможных случаев, в нашем примере эти возможности — принять либо число, либо переменную. На этот шаг пришлось пойти, потому что иначе было непонятно, как получить значение, которое надо либо присвоить, либо вывести на печать, когда само значение вычленяется разными способами: для числа это делает токен `value`, а для переменной следует посмотреть в хеш `%var` по ключу, извлекаемому токеном `identifier`.

Рассмотрим, как вернуть стройность грамматики, одновременно сохранив возмож-

ность альтернативы. Итак, прежде всего возвращаем правило `expression`:

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

Построенная при разборе иерархия — синтаксическое дерево, способно хранить результаты выполнения действий на предыдущих (нижележащих) шагах. Специально для этого у объекта типа `Match` есть поле `ast`, а в каждом узле есть возможность непосредственно получать результат, вычисленный в дочерних узлах.

AST = abstract syntax tree, и абстрактное оно именно потому, что при его использовании можно абстрагироваться от способа получения результата в текущем узле: имеется конкретный результат, а способ его вычисления неважен.

Действие может сохранить результат своей работы (и таким образом передать его выше по дереву), вызвав метод `$/.make`, который сохраняет данные, которые далее могут быть прочитаны из поля `made` (или используя синонимичное обращение `ast`).

Начнем заполнять атрибуты синтаксического дерева (то есть вычисленные значения, связанные с узлом), начиная с токенов `identifier` и `value`. Результат совпадения с первым из них — строка с именем переменной, второго — число:

```
1 method identifier($/) {
```

```
2     $/.make(~$0);
3 }
4 method value($/) {
5     $/.make(+$0);
6 }
```

Теперь поднимемся на уровень выше и построим значение для выражения (`expression`), которое может быть либо значением переменной, либо числом.

Поскольку правило `expression` содержит две альтернативы, нужно прежде всего понять, какая из них совпала, это легко сделать, проверив наличие в объекте `$/` поля `identifier` или `value`. Запись `$<identifier>` — упрощенный вариант конструкции `$/<identifier>` (именно поэтому в качестве аргумента метода удобно использовать переменную `$/`, а не любую другую с обычным именем, например,

$match).

Для каждой ветви результат вычисляется по-своему. Для числа это преобразование из поля value: +$<value>, а для переменной — чтение нужного значения из хеша: %var{$<identifier>}. Полученное значение сохраняется в атрибуте узла синтаксического дерева с помощью вызова $/.make().

```
1 method expression($/) {
2     if $<identifier> {
3         $/.make(%var{$<identifier
              >});
4     }
5     else {
6         $/.make(+$<value>);
7     }
8 }
```

Чтобы разрешить использовать еще необъ-

явленные переменные, можно записать:

```
1 $/.make(%var{$<identifier>} // 0)
      ;
```

На этом этапе правило `expression` будет содержать конкретное значение, а было ли оно получено из константы или переменной — неважно. Поэтому сразу упрощается обработка вышележащих правил `assignment` и `printout`. Начнем со второго:

```
1 method printout($/) {
2     say $<expression>.ast;
3 }
```

Все очень просто: на печать выводится значение, которое уже подготовлено и находится в поле `ast`.

Действие для правила `assignment` чуть по-

сложнее, но тоже умещается в одну строку:

```
1 method assignment($/) {
2     %var{$<identifier>} = $<
          expression>.made;
3 }
```

Этот метод при вызове должен получить объект `$/`, в котором есть поля `identifier` и `expression`. Первое преобразуется в строку и дает имя переменной. Из второго берется атрибут узла (для разнообразия на этот раз через вызов `made`).

Последнее замечание по поводу класса с действиями. Хеш `%var`, в котором хранятся значения переменных, целесообразно сделать данными класса, а не глобальной переменной. Соответственно, в классе появляется строка `has %.var;`, а обращение внутри методов будет выглядеть как `%!var{...}`.

Теперь, поскольку у класса с действиями появляются данные, при вызове методов `parse` или `parsefile` необходимо передавать не имя класса, а создавать его инстанс:

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

Полный пример грамматики с действиями для копипейста:

```
1  grammar Lang {
2      rule TOP {
3          ^ <statements> $
4      }
5      rule statements {
6          <statement>+ %% ';'
7      }
8      rule statement {
9          | <assignment>
10         | <printout>
11     }
```

```
12      rule assignment {
13          <identifier> '=' <
                expression>
14      }
15      rule printout {
16          'print' <expression>
17      }
18      rule expression {
19          | <identifier>
20          | <value>
21      }
22      token identifier {
23          (<:alpha>+)
24      }
25      token value {
26          (\d+)
27      }
28  }
29
30  class LangActions {
31      has %var;
32
33      method assignment($/) {
```

```
34          %!var{$<identifier>} = $<
               expression>.made;
35      }
36      method printout($/) {
37          say $<expression>.ast;
38      }
39      method expression($/) {
40          if $<identifier> {
41              $/.make(%!var{$<
                   identifier>} // 0)
                   ;
42          }
43          else {
44              $/.make(+$<value>);
45          }
46      }
47      method identifier($/) {
48          $/.make(~$0);
49      }
50      method value($/) {
51          $/.make(+$0);
52      }
53  }
```

```
54
55 Lang.parsefile('test.lang', :
    actions(LangActions.new()));
```

Хочу обратить внимание на несколько головоломный момент. Имена переменных выделяет токен `identifier`, который сохраняет имя в атрибуте `ast`. При этом при разборе конструкции `x = y` токен срабатывает дважды, но дальнейшие действия с переменными будут зависеть от того, с какой стороны от знака равенства они стоят. Переменная `x` будет фигурировать внутри метода `assignment`, в то время как из переменной `y` будет сделано готовое значение в методе `expression`. У меня возник вопрос о том, как же так? уже только после того, как программа с написанной грамматикой заработала. Мораль этого лирического отступления: если аккуратно расписать правила языка, то грамматика

будет работать правильно и как ожидалось.

Калькулятор

Рассказ про построение грамматики был бы не полным без классического примера — калькулятора, который умеет вычислять значение арифметического выражения произвольной сложности, состоящего из операций `+`, –, `*`, `/` и скобок. Этот пример хорош тем, что несмотря на простоту выполняемых действий он должен уметь делать вычисления с учетом приоритета операций и группировки, в том числе со вложенными скобками.

Калькулятор будет принимать только одно выражение `expression`. Приоритет операций неявно указан способом построения

грамматики. Выражение может состоять из частей term, разделенных плюсами и минусами:

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

То же самое можно записать более традиционно:

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

Каждая часть, в свою очередь является последовательностью частей factor, разделенных символами умножения или деления:

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

На месте term или factor может стоять либо значение value, либо подвыражение group, заключенное в скобки:

```
1 rule group {
```

```
2     '(' <expression> ')'
3 }
```

Внутри скобок начинается новый виток рекурсии, и все, что находится в них, разбирается точно так же, как и выражение `expression` на верхнем уровне.

Для выделения чисел на этот раз сделаем чуть более сложное описание токена, который позволит принимать и целые числа, и числа с десятичной точкой:

```
1 token value {
2     | \d+['.' \d+]*
3     | '.' \d+
4 }
```

Полная грамматика для калькулятора выглядит так:

```
1 grammar Calc {
2     rule TOP {
```

```
3           ^ <expression> $
4       }
5       rule expression {
6           | <term>+ %% $<op>=(['+'|
               '-'])
7           | <group>
8       }
9       rule term {
10          <factor>+  %% $<op>=(['*'
               |'/'])
11      }
12      rule factor {
13          | <value>
14          | <group>
15      }
16      rule group {
17          '(' <expression> ')'
18      }
19      token value {
20          | \d+['.' \d+]*
21          | '.' \d+
22      }
23  }
```

Обратите внимание на конструкцю `$<op>=(...)` в правилах `expression` и `term`. Это именованный захват. Все, что находится в круглых скобках, окажется в поле `$<op>` переменной `$/`.

Теперь перейдем к классу действий. На верхнем уровне метод `TOP` возвращает вычисленное ниже значение, которое должно находиться в поле `ast`:

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

   . . .
}
```

Так же тривиально выглядят действия для вычисления `group` и `value`:

```
1 method group($/) {
2     $/.make: $<expression>.ast
3 }
4 
5 method value($/) {
6     $/.make: +$/
7 }
```

Правило `factor` содержит две альтернативы, поэтому необходимо выбрать правильную ветвь, после чего все опять просто:

```
1 method factor($/) {
2     if $<value> {
3         $/.make: +$<value>
4     }
5     else {
6         $/.make: $<group>.ast
7     }
8 }
```

Теперь перейдем к правилу `term`. Здесь

обработка чуть более сложная: во-первых, впервые встретилась необязательная последовательность, число повторений которой может быть любым. Во-вторых, необходимо определить, какая производится операция — умножение или деление. Символ арифметической операции сохраняется в переменной `$<op>`.

Дерево разбора для выражения `3*4*5` выглядит так:

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

Здесь хорошо видно, что все factor и op находятся на одном уровне. Внутри действия они будут видны как элементы массивов $<factor> и $<op>. Всегда будет доступен как минимум один $<factor>. Сами значения узлов будут к этому моменту уже известны и находятся в ast. Соответственно, нужно пройтись по всем элементам этих двух массивов и сделать для каждого из них умножение или деление.

```
1 method term($/) {
2     my $result = $<factor>[0].ast
          ;
3 
4     if $<op> {
5         my @ops = $<op>.map(~*);
6         my @vals = $<factor
              >[1..*].map(*.ast);
7 
8         for 0..@ops.elems – 1 –>
              $c {
```

```
 9                 if @ops[$c] eq '*' {
10                     $result *= @vals[
                          $c];
11                 }
12                 else {
13                     $result /= @vals[
                          $c];
14                 }
15             }
16         }
17
18         $/.make: $result;
19 }
```

В этом коде звездочка встречается в новом качестве, как плейсхолдер, сообщающий перлу, что надо работать с тем, что есть под рукой.

Массив @ops, где хранится список символов операций, состоит из элементов, полученных из элементов массива $<op>

путем преобразования в строку:

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

Сами значения, окажутся в массиве `@vals`. На этот раз значения взяты из поля `ast`. Чтобы массивы `@vals` и `@ops` точно соответствовали друг другу, из `$<factor>` взят срез, начиная со второго элемента:

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

Действия для `expression` — либо взять вычисленное значение `group`, либо сделать последовательность сложений и вычитаний. Здесь алгоритм почти совпадает с тем, что только что делалось для умножения и деления:

```
1 method expression($/) {
2     if $<group> {
3         $/.make: $<group>.ast
```

```
4       }
5       else {
6           my $result = $<term>[0].ast;
7
8           if $<op> {
9               my @ops = $<op>.map(~*);
10              my @vals = $<term>[1..*].map(*.ast);
11
12              for 0..@ops.elems – 1 –> $c {
13                  if @ops[$c] eq '+' {
14                      $result += @vals[$c];
15                  }
16                  else {
17                      $result –= @vals[$c];
18                  }
```

```
19                 }
20             }
21
22             $/.make: $result;
23         }
24 }
```

Наконец, код, который будет направлять данные из аргумента командной строки на разбор и печатать результат:

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

Проверка работоспособности калькулятора:

```
1 $ perl6 calc.pl '42 + 3.14 * (7 -
      18 / (505 - 502)) - .14'
2 45
```

Читателю предлагается самостоятельно научить калькулятор работать с отрицательными числами, а затем соединить две грамматики таким образом, чтобы интерпретатор мини-языка умел вычислять арифмитические выражения (для этого надо модифицировать правило и действия `expression`), а затем разрешить использовать в выражениях переменные (здесь потребуется расширить правило `term`).

Задание со звездочкой (еще один смысл звездочки): разобраться с метаоператорами и избавиться от циклов внутри методов `term` и `expression`.

Исходные файлы

Исходные тексты всех программ и тестовых примеров, описанных в статье, вместе с их промежуточными версиями доступны на гитхабе, там же есть и реализация мини-языка, скрещенного с калькулятором.

■ *Андрей Шитов*

6 Обзор CPAN за март 2015 г.

Рубрика с обзором интересных новинок CPAN за прошедший месяц.

Статистика

- Новых дистрибутивов — 212
- Новых выпусков — 932

Новые модули

re::engine::GNU

`re::engine::GNU`, как видно из названия, является движком регулярных выражений,

основанный на библиотеке gnulib. Теперь для регулярных выражений становится возможно использовать синтаксис регулярных выражений emacs, egrep/grep, awk, sed и прочих вариаций, поддерживаемых в gnulib. Существует три альтернативных варианта записи регулярных выражений:

```
use re::engine::GNU;

# классический вариант
'test' =~ /\(tes\)t/;

# массив: синтаксис, шаблон
'test' =~ [ 0, '\(tes\)t' ];

# хеш: синтаксис, шаблон
'test' =~ { syntax => 0, pattern
    => '\(tes\)t' };
```

Time::Monotonic

Измерение периодов времени простым вычитанием показаний функции `time()` может иногда быть абсолютно неточным, вплоть до получения отрицательных показаний. Причина в том, что часы компьютера могут быть изменены как вручную, так и утилитами, например, ntpdate. Запущенный ntpd хоть и не изменяет время скачками, но может замедлять или ускорять часы, что также приводит к неаккуратным подсчётам периодов времени.

Модуль `Time::Monotonic` даёт доступ к показаниям источника монотонного времени, существующий на различных платформах, которое позволяет точно вычислять периоды времени.

List::Prefixed

`List::Prefixed` позволяет создавать список строк-префиксов, который можно использовать, например, для создания регулярного выражения, которое совпадает с любой строкой из списка. Так, например, для списка строк

```
"Ba", "Bar", "Baz", "Foo", "Food", "Foot", "For", "Form", "Fu"
```

генерируется регулярное выражение, совпадающее с любой из этих строк

```
/(?:Ba(?:r|z)?|F(?:o(?:o(?:d|t)?|r(?:m)?)|u))/
```

Это можно использовать для создания тестов на вхождение слов в тексте, эффективной реализации автодополнения и даже сжатия.

Plack::Util::Load

Модуль Plack::Util::Load позволяет загружать PSGI-приложение из файла, класса или URL. Экспортируемая функция load_app() возвращает ссылку на код или генерирует исключение в случае ошибки.

```
# Загрузка приложения из файла
$app = load_app('app.psgi');

# Загрузка приложения по имени класса
$app = load_app('MyApp::PSGI');

# Загрузка по URL (приложение работает как прокси на базе HTTP::Tiny)
$app = load_app("http://example.org/");
```

Rapi::Fs

`Rapi::Fs` — это Plack-приложение, созданное на основе фреймворка RapidApp, которое представляет собой файловый браузер. Веб-сервер запускается скриптом `rapi-fs.pl`, которому передаётся в качестве параметра путь к каталогу. Получаем довольно функциональный файловый браузер, которым можно пользоваться в любом браузере.

Deep::Hash::Exists

Полезный модуль `Deep::Hash::Exists` для проверки наличия глубоко вложенного ключа. Как известно, выражение

```
exists $hash_ref->{a}{b}{c}{d};
```

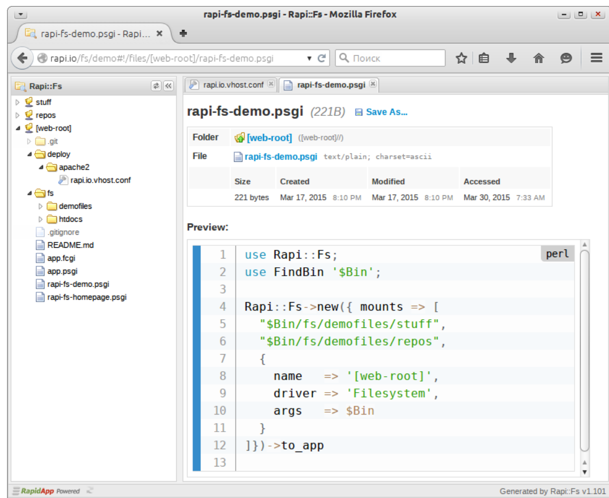


Рис. 0.1: Скриншот Rapi::Fs

благодаря автовификации создаст все промежуточные элементы за исключением последнего ключа `d`. Та же самая операция с помощью функции `key_exists`

```
key_exists( $hash_ref, [ qw(a b c d) ] );
```

не обладает таким побочным эффектом и может быть безопасно использована.

Plack::Middleware::GNUTerryPratchett

Модуль `Plack::Middleware::GNUTerryPrat` добавляет в PSGI-приложение HTTP-заголовок

```
X-Clacks-Overhead: GNU Terry Pratchett
```

Таким образом весь мир отдаёт дань памяти известному английскому писателю Терри Прачетту, который умер 12 марта 2015 года. Происхождение заголовка связано с сюжетом книги «Опочтарение»,

в котором присутствовали щёлкающие башни — семафоры (аналог телеграфа), которые передавали сообщения. Расшифровка кодов: `G` — передача сообщения, `N` — не вести лог сообщения, `U` — отправить сообщение назад, если оно достигло конца линии. Соответственно сообщение, отправленное с кодом `GNU` будет вечно циркулировать в линии. «Человек не умер, пока его имя произносится» говорится в прологе 4-й главы книги. Таким образом пытаются сохранить память о Терри Прачетте.

Crypt::Ed25519

`Crypt::Ed25519` — это реализация алгоритма цифровой подписи EdDSA с использованием эллиптической кривой Twisted Edwards. Особенность данной

эллиптической кривой — в высокой производительности операций и при этом высокой стойкости ко взлому. Как указано в описании модуля, стойкость получаемой подписи (512 бит) эквивалентна 3000 бит RSA или AES-128.

Обновлённые модули

Moo 2.001

Вышел второй мажорный релиз ООП-фреймворка `Moo`. Основное несовместимое изменение — отказ от фатальных предупреждений. В этом отношении Moo стал соответствовать поведению Moose. Кроме того, раньше классы без атрибутов сохраняли все параметры, переданные в `new()`, внутри объекта. Теперь этого

не происходит — поведение приведено в соответствие с классами, которые имеют атрибуты. Попытка переопределить существующий конструктор или изменить тот, который был использован, теперь приводит к ошибке.

DBIx::Class 0.082820

Появилось обновление ORM `DBIx::Class` с исправлениями ошибок и улучшениями в документации. `DBIx::Class` теперь требует `Moo` не ниже версии 2, чтобы обеспечить более предсказуемую и безопасную работу модуля.

HTTP::BrowserDetect 2.00

Вышел второй мажорный релиз модуля `HTTP::BrowserDetect` для определения веб-браузера, версии и платформы по HTTP-заголовку `User-Agent`. В новой версии стандартизирован интерфейс, добавлена информация о новых браузерах. Основное несовместимое изменение — методу `user_agent` больше нельзя передавать параметр для задания строки веб-агента, для этих целей нужно использовать конструктор `HTTP::BrowserDetect->new()`.

Raisin 0.58

В новом релизе микро-фреймворка `Raisin` для создания REST API появилась поддерж-

ка спецификации Swagger 2.0.

Devel::PPPort 3.31

Вышло обновление модуля `Devel::PPPort` для поддержки новых функции Perl API в старых версиях Perl. Релиз в частности содержит исправление функции `SvPV_renew`, в которой параметр длины строки имел тип `int` вместо `size_t`, что потенциально могло быть источником проблем. Проблема была обнаружена сканером Coverity.

Rex 1.1.0

В марте вышел первый мажорный релиз `Rex` — фреймворка автоматизации задач по

администрированию. Ветка 1.0 объявлена как LTS-релиз и будет поддерживаться до марта 2017 года. В новом релизе произошёл переход от использования libssh2 (`Net::SSH2`) к openssh (`Net::OpenSSH`) по умолчанию, восстановлена совместимость с Perl 5.8.9, появилась поддержка команд PkgConf для запроса изменения конфигурации пакета (особенно полезно на Debian/Ubuntu). Также большое число багфиксов и других улучшений.

Net::OpenSSH 0.64

Новый стабильный релиз модуля `Net::OpenSSH` содержит достаточно много внутренних изменений, и не исключены регрессии или новые баги. Будьте внимательны при обновлении.

Server::Starter 0.23

Обновлён модуль супердемона `Server::Starter`. В новом релизе удалены все зависимости, которых нет в базовой поставке Perl. Появилась поддержка ipv6.

perl 5.21.10

Вышел новый релиз Perl ветки 5.21 для разработчиков. Основные изменения:

- Операторы в выражениях для экспериментальных расширенных классов символов `(?[...])` теперь следуют стандартным правилам приоритетов операций в Perl. Операция пересечения классов символов `&` теперь

имеет более высокий приоритет, чем другие бинарные операции.

- Функции `utf8::native_to_unicode()` и `utf8::unicode_to_native()` теперь оптимизированы для платформ ASCII. Написание кода, переносимого между ASCII и EBCDIC, теперь не будет испытывать пенальти в производительности.
- Perl теперь корректно компилируется и работает на z/OS с кодовой страницей 1047 EBCDIC.
- Исправлена проблема, когда повторяемый глобальный поиск шаблона в скалярном контексте на больших tainted-строках приводил к экспоненциальному замедлению.
- Очередная пачка исправлений крахов Perl, найденных с помощью фаззера AFL.
- Исправлена регрессия в операторе

readline, появившаяся после добавления нового оператора двойного бриллианта <<>>.

DBD::Firebird 1.19

В новом релизе DBI-драйвера СУБД Firebird помимо других исправлений ошибок исправлено переполнение буфера в dbdimp.c. Сообщения об ошибках помещаются с помощью sprintf в буфер фиксированной длины, который в некоторых ситуациях может оказаться слишком мал. Проблема получила идентификатор CVE-2015-2788.

■ *Владимир Леттиев*

7 Интервью с Виктором Турским

Виктор Турский (koorchik) — украинский Perl-программист, сооснователь компании WebbyLab

Как и когда научился программировать?

Это длинная история. В целом, я никогда не собирался быть программистом, даже когда уже работал им, я все еще не собирался им быть. Так вот, в детстве мне родители купили книгу «А я был в компьютерном городе» (поищи в google, там классные картинки). Это было то ли ли в 3-м, то ли в 4-м классе, и думаю, что после этого я загорелся компьютерами. Правда первый компьютер у меня появился только через 4 года (это был старый «commodore 64»), но к тому моменту я прочитал о них все,

что мог найти. В 9-м классе мне достался другой очень «крутой» компьютер: 1MB Ram, CPU 8086(4 Mhz), 5.25 floppy drive. Вот тогда я и начал играться с Basic, разобрался с DOS, с драйверами, файловой системой и так далее. Помню, что первое, что я написал, была программа, которая решала квадратные уравнения. А делать что-то более полезное я стал значительно позже — в универе, писал разного рода приложения на Bash, AWK, Perl, Visual Basic.

Какой редактор используешь?

В свое время, много лет сидел на Eclipse + Epic, но года три назад полностью перешел на Sublime Text. Vim тоже пробовал, через 2-3 месяца понял, что я менее эффективен в нем. Сейчас у нас в офисе не услышишь холиваров по поводу редактора — Sublime Text всех подружил :)

Как и когда познакомился с Perl?

Как я уже сказал, я никогда не планировал быть программистом. Меня больше интересовали информационная безопасность, сети, взломы, архитектура операционной системы. Учась в университете, я часто игрался с разными эксплоитами, бекдорами, сетевыми утилитами и т.д. Чаще всего они были написаны на C++ и реже на Perl. Я понимал, что умение программировать мне сильно поможет в моем увлечении. Эксплоиты бывало содержали какую-то мелкую ошибку, и нужно было ее исправить (такой себе механизм защиты от script kiddies). Исправить ошибку мне хватало скилов, но написать какое-то сетевое приложение я не мог. Тогда я купил две книги «Сетевое прораммирование на Perl» и книгу с верблюдом. Кроме того, что Perl позволял легко писать сетевые прило-

жения, он отлично справлялся с задачами вычистки лишних данных из логов и т.д. Если говорить о чем-то полезном, то одним из первых моих скриптов был скрипт, который читал excel-файл, смотрел, кто не заплатил абонплату за сеть в общаге, а потом ходил по телнету на управляемый свитч и блокировал неплательщиков по мак-адресу :).

С какими другими языками интересно работать?

Сейчас много работаю с JavaScript. Поначалу, после Perl, меня от него коробило, но когда появились ES5 и ES6, то отпустило. Сейчас же стартую все проекты именно на JavaScript.

Также считаю интересным Golang. Правда реального продакшена нет, так поигрался.

Сейчас очень немного практических задач, где я вижу от него пользу.

Слежу за Perl6. Не думаю, что когда-то у меня будет продакшен проект на нем, но сам язык просто кладезь интересных подходов и решений. Чего стоит только банальный блок CATCH, давно мечтал о таком. Или «reduction operators». Высокоуровневые примитивы для работы с многопоточностью, как то: каналы, промисы и т.д.

Считаю, что со многими языками интересно работать (если это не VB.Net). По факту же интерес вызывает не просто сам язык, а задача, которую ты на нем решаешь.

Что, по-твоему, является самым большим преимуществом Perl?

Мне нравится гибкость Perl и его стабиль-

ность. Стабильность интерпретатора и всей экосистемы. Также у Perl есть своя философия, если ты ее понимаешь, то Perl никогда не стоит у тебя на пути, он ведет себя ожидаемо.

Есть еще масса позитивных мелочей. Например, в Perl действительно круто реализована нестрогая типизация. Благодаря мономорфным операторам (и use warnings) я никогда не задумываюсь, число или строка в переменной. В том же JavaScript это сделано очень криво. Вроде бы там нестрогая типизация, но ввиду полиморфности операторов я всегда должен помнить типы, иначе два плюс два даст двадцать два. Нигде мне не было так удобно использовать регулярные выражения, как в Perl. Когда я выкладываю модуль на CPAN, он будет протестирован просто в безумной массе окружений. Не встречал нигде тако-

го крутого CI. Качество модулей на CPAN значительно выше, чем на том же npmjs, и меньше ломаешь голову, что же из этого хлама заработает.

В общем, вывод следующий: делать работу на Perl — это «Fun». Думаю это и есть самое большое преимущество.

Что, по-твоему, является самой важной особенностью языков будущего?

Никогда не задумывался об этом. Естественно, можно сказать, что это многопоточность и без нее правда никуда, но давай поразмыслим. Язык программирования — это лишь инструмент для решения задач. То есть, язык будущего — это тот язык, который сможет решать задачи и проблемы будущего.

Полагаю, что в будущем будут усиливаться две проблемы:

1. Дефицит программистов на рынке и их высокая стоимость.
2. Рост количества данных.

Более детально разберем проблему №1. Нужно откуда-то брать программистов. Единственный способ — это обучение. Но крутыми программисты становятся после многих лет работы. Для того, чтобы удовлетворить спрос на программистов, нам нужно уменьшить срок их подготовки, чтобы они могли решать задачу сложности X не через 2 года, а через 5 месяцев, например. Мы помним, что сложность решения задачи состоит из «essential complexity» и «accidental complexity». Брукс говорил, что проблемы «accidental complexity» в

значительной мере решены и потенциал для уменьшения «accidental complexity» минимален. В целом это верно, особенно, если мы разрабатываем, например, бухгалтерскую систему, где очень высокий уровень «essential complexity». Но сегодня масса приложений значительно проще и уменьшение «accidental complexity» сильно бы сократило время разработки и повысило стабильность продукта.

Теперь представим, что новичок-программист должен с легкостью начинать писать вменяемый код и иметь возможность обрабатывать достаточно большие объемы данных. Где тут сложности:

1. Язык не должен заставлять пользователя изучать много концепций и создавать сложности. То есть Java это не язык будущего :)

2. Сообщение об ошибках. Язык будущего должен обладать мощным статическим анализатором и предупреждать об ошибках заранее. И самое главное, все ошибки должны быть понятными!
3. Упрощение отладки кода. Это не только про инструменты, но и про сам синтаксис языка. Синтаксис должен быть лаконичным и однозначным.
4. Разделяемое изменяемое состояние часто бывает источником проблем (как говорится, «Shared mutable state is the root of all evil»). Должна быть возможность легко сделать структуры неизменяемыми. Например, я хочу заморозить иерархическую структуру, я говорю freeze(struct). В JavaScript есть похожее Object.freeze, но там имеется два недостатка: нет возможности создать новую струк-

туру на базе существующей, при попытке изменения не возникает ошибка.

5. Паралельная обработка данных (многопоточность, асинхронность и т.д). Все это, само по себе, является сплошным accidental complexity. Одно из тех мест, где еще остался высокий уровень сложности. Язык будущего должен иметь встроенные (как регулярки в Perl) высокоуровневые абстракции (Promises, Channels, Generators…), которые инкапсулируют эти сложности.
6. Организация связей между сущностями. Это одна из ключевых проблем. ООП решает эту проблему лишь частично, а часто даже создает лишние проблемы. Раньше мы говорили про наследование, сейчас говорим про композицию. Стали модными

декораторы, роли (traits) и так далее. Кто-то скажет, что функциональное программирование эти проблемы решает. Не буду спорить, но мое видение такое — универсального решения еще нет и возможно не будет. Мы не знаем, что наиболее эффективно в конкретном случае. Тут лучше отталкиваться от обратного — язык будущего не должен включать (по умолчанию) ничего, что сильно повышает риск создания спагетти-кода.

7. Развертывание приложения и управление зависимостями.
8. И самое главное. Должен быть «Fun». «Fun» мотивирует и ускоряет процесс обучения.

Ух, жесть. Если не знаешь ответа, то коротко ответить не получится :).

Что думаешь о будущем Perl?

Если говорить про Web-разработку, то в ближайшие 5 лет JavaScript будет набирать все большую популярность. Perl, Ruby, Python будут терять позиции. Perl, как и раньше, будет отлично справляться со своими задачами, но он будет сидеть в своей нише. Например, даже Ruby — язык, который сейчас ассоциируется с веб-разработкой, вытесняется nodejs. Не думаю, что и Perl сможет увеличить свою долю на этом рынке. Если говорить про использование в общем, то в ближайшие 10 лет ничего не изменится.

Хотелось бы, чтобы выстрелил Perl6, но для этого нужен большой продакшен на нем и продвижение.

Что такое LIVR?

Каждый программист неоднократно сталкивался с необходимостью проверки пользовательского ввода. Занимаясь веб-разработкой уже более 10 лет, я перепробовал массу библиотек, но так и не нашел той единственной, которая решала бы поставленные мною задачи. Три года назад было решено написать собственный валидатор, который был бы идеальным. Так появился LIVR (Language Independent Validation Rules, http://livr-spec.org). Есть реализации на Perl, PHP, JavaScript. Валидатор используется в продакшене уже несколько лет практически в каждом проекте компании. Валидатор работает как на бекенде, так и на фронтенде. Также есть сторонняя реализация, написанная на python, мы на python не пишем — фидбек дать не могу. Поиграться с валидатаром можно тут: webbylab.github.io/livr-playground.

В Pragmatic Perl уже была статья про LIVR, можно с нее начать. Сейчас пишу более детальную статью, будет доступна на хабре.

В твоей компании часто используется Mojolicious. С чем это связано?

Mojolicious мне очень нравится концепцией. Нет, я не про то, что у него нет внешних зависимостей. Мне нравится, что это Web-фреймворк, который делает акцент на Web. Он не пытается решить проблемы хранения данных и прочее, а концентрируется на Web-составляющей: websockets, HTTP/2, HTML Parsing. То есть, делает то, что должен делать Web-фреймворк, но не скатывается до уровня микро-фреймворков. Такой подход — редкость, обычно либо микрофреймворк, либо убийственно толстый fullstack-фреймворк. Mojolicious — это идеальный компромисс.

Требуется ли бизнесу Perl? Как нанимаете Perl-программистов?

Perl отличный язык, но я считаю, что в веб-разработке будущее за JavaScript. Пересекающийся стек технологий и на фронтенде, и на бекенде — это значительно эффективнее.

У нас наиболее крупные проекты на Perl, но в новых проектах Perl используется только тогда, когда JavaScript не справляется (есть такие моменты :)). Каких-то особенностей в найме Perl-программистов нет, возможно только то, что мы выросли из Perl-комьюнити и там больше знакомых.

Используете ли какую-то методологию разработки, как контролируете и улучшаете качество своих продуктов?

Подходы варьируются от проекта к проекту и от заказчика к заказчику, но это всегда Agile (часто SCRUM, но не во всех проектах). Обычно мы просим заказчика выделить одного человека, который будет ответственен за постановку требований и за прием результата. Мы же, со своей стороны, тоже предоставляем человека, который будет точкой коммуникации для заказчика. Практически всегда работаем без предоплаты, чтобы минимизировать риски заказчика. Если не сделаем то, что хочет заказчик, то оплату не получим — все очень просто. Это очень мотивирует строить эффективный процесс в компании. В целом ничего нового, просто стараемся использовать лучшие практики и наиболее эффективные инструменты.

С технической стороны ничего особенного:

- Используем Git, каждая задача делается в отдельной ветке.
- Test coverage для бекенда должен быть не менее 80%.
- Все задачи проходят code review (используем gitlab для этого).
- Перед тем, как тикет попадет в master, должны успешно пройти тесты. Для continuous integration используем Gitlab-CI.
- Каждая задача проходит проверку тестировщиком. Он ответственен за то, чтобы до заказчика доходил только качественный продукт.
- Всегда пишется документация к REST API.
- Разворачиваем все через Ansible.
- Проекты ведем в Redmine.

Если интересен стек технологий, то вот он

http://stackshare.io/webbylab/webbylab.

Можно ли совмещать управление с разработкой?

Сейчас я практически перестал писать код по проектам клиентов (только изредка берусь за какие-то уж очень критические куски). Но код писать важно, иначе перестанешь чувствовать разработку и ее проблемы. В связи с этим, я пишу код для проектов, где нет жесткого дедлайна. Это обычно внутренние инструменты, библиотеки, какие-то proof of concepts и т.д.

Я всегда стараюсь пропустить через себя технологию перед тем, как начинать ее использовать на уровне компании. В связи с этим я могу взять уже готовый проект и попробовать его переписать на

другой технологии и сравнить, будет ли это эффективней.

Сколько времени проводишь за написанием Perl-кода?

Сейчас пишу мало кода в принципе. На Perl пишу еще меньше — обновляю внутренние инструменты, модули на CPAN.

Вы спонсировали несколько Perl-мероприят Чем руководствовались при принятии этого решения? Будете ли продолжать?

Основатели компании вышли из Perl-комьюнити. Спонсорство, предоставление офиса компании на технические встречи, участие в Perl-хакатонах — это все возможность держать связь с комьюнити. Мы время от времени спонсируем интересные для нас мероприятия. Например, второй

год подряд мы являемся спонсорами iForum. Это конференция связана с решением основать компанию, мы ее любим сильно и всех приглашаем посетить.

Стоит ли советовать молодым программистам учить сейчас Perl?

Молодым программистам советую идти работать. Если работа подразумевает использование Perl — отлично, и это будет замечательный опыт, если нет — ничего страшного. Только код, доведенный до продакшена, делает из тебя программиста, другого пути нет :).

Вопросы от читателей

Когда будут еще статьи для журнала?

Чтобы написать статью, должно совпасть

много факторов: свободное время, желание писать, хорошая идея, свежая голова. Сейчас в планах написать статью про LIVR на хабр, но за последние несколько месяцев я особо не продвинулся. Если что-то будет написано для журнала, то это точно не в ближайшее время.

Какую следующую конференцию планируешь посетить?

Планирую быть на iForum-2015 вместе со всей командой. Как и уже упоминал, мы опять спонсоры этого мероприятия. У нас есть традиция: каждый год компания оплачивает всем сотрудникам поход на iForum, и мы там делаем коллективное фото в футболках WebbyLab. Эти фото потом висят по офису, всегда приятно смотреть, как всего за чуть больше трех лет мы выросли с четырех до 20 человек.

Что значит koorchik?

Сколько раз мне уже этот вопрос задавали :). koorchik я уже около 20-ти лет, еще со школы повелось. То есть я стал koorchik-ом еще до того, как появились соцсети и у меня интернет. Просто среди друзей кто-то меня так назвал и прицепилось.

■ *Вячеслав Тихановский*