# Banking Left Into the Parser Zone

*The examples used here are from the weekly challenge problem statement and demonstrate the working solution.*

## Part 1: Banking Day Offset

*You are given a start date and offset counter. Optionally you also get bank holiday date list. Given a number (of days) and a start date, return the number (of days) adjusted to take into account non-banking days. In other words: convert a banking day offset to a calendar day offset.*

*Non-banking days are:*

*(a) Weekends*

*(b) Bank holidays*

Using **Time::Piece** the work can be contained in a single function. Really the main piece of logic required of **sub count_days()** is for us to check if a day is a weekend or bank holiday.

⟨ *count days* 1 ⟩ ≡

```
sub count_days{
    my($start, $offset, $holidays) = @_;
    $start = Time::Piece->strptime($start, q/%Y-%m-%d/);
    my $t = $start;
    my $end = $start;
    {
      $t += ONE_DAY;
      unless(⟨ The day is a weekend. 2 ⟩ || ⟨ The day is a bank holiday. 3 ⟩){
          $end = $t;
          $offset--;
      }
      redo if $offset > 0;
    }
    return $end->strftime(q/%Y-%m-%d/);
}
```

◇

Fragment referenced in 4.

⟨ *The day is a weekend.* 2 ⟩ ≡

```
$t->wday >= 6
```

◇

Fragment referenced in 1.

⟨ *The day is a bank holiday.* 3 ⟩ ≡

```
1 == grep {$t->strftime(q/%Y-%m-%d/) eq $_} @{$holidays}
```

◇

Fragment referenced in 1.

The rest of the code just tests this function.

`"perl/ch-1.pl"` 4≡

```
⟨ preamble 5 ⟩
⟨ count days 1 ⟩
⟨ main 6 ⟩
```

◇

⟨ *preamble* 5 ⟩ ≡

```
use v5.38;
use Time::Piece;
use Time::Seconds;
```

◇

Fragment referenced in 4.

⟨ *main* 6 ⟩ ≡

```
MAIN:{
  say count_days q/2018-06-28/, 3, [q/2018-07-03/];
  say count_days q/2018-06-28/, 3;
}
```

◇

Fragment referenced in 4.

### Sample Run

```
$ perl perl/ch-1.pl
2018-07-04
2018-07-03
```

## Part 2: Line Parser

*You are given a line like below:*

*{% id field1="value1"field2="value2"field3=42 %}*

*Where*

(a) *"id"can be \w+.*

(b) *There can be 0 or more field-value pairs.*

(c) *The name of the fields are \w+.*

(d) *The values are either number in which case we don't need double quotes or string in which case we need double quotes around them.*

*The line parser should return a structure like:*

```
{
        name => id,
        fields => {
            field1 => value1,
            field2 => value2,
            field3 => value3,
        }
}
```

*It should be able to parse the following edge cases too:*

```
{%  youtube  title="Title␣\"quoted\"␣done" %}
```

*and*

```
{%  youtube  title="Title␣with␣escaped␣backslash␣\\" %}
```

Most of the work is done in a parser constructed using **Parse::Yapp**.

### ch-2.pl

First off, before we get into the parser, here is a small bit of code for driving the tests.

⟨ *print the parser results* 7 ⟩ ≡

```
sub print_record{
  my($record) = @_;
  say q/{/;
  say qq/\tname => / . $record->{name};
  say qq/\tfields => {/;
  for my $field (sort {$a cmp $b} keys %{$record->{fields}}){
      say qq/\t\t$field => / . q/ / . $record->{fields}->{$field};
  }
  say qq/\t}/;
  say q/}/;
}
```

◇

Fragment referenced in 8.

The rest of the code drives some tests.

`"perl/ch-2.pl"` 8≡

```
⟨ preamble 9 ⟩
⟨ print the parser results 7 ⟩
⟨ main 10 ⟩
```

◇

⟨ *preamble* 9 ⟩ ≡

```
use v5.38;

use Ch2;
use constant TEST0 => q/{% id field1="value1" field2="value2" field3=42 %}/;
use constant TEST1 => q/{% youtube title="Title␣\"quoted\"␣done" %}/;
use constant TEST2 => q/{% youtube title="Title␣with␣escaped␣backslash␣\\\\" %}/;
```

◇

Fragment referenced in 8.

⟨ *main* 10 ⟩ ≡

```
MAIN:{
  my $parser = Ch2->new();
  say TEST0;
  print_record($parser->parse(TEST0));
  say TEST1;
  print_record($parser->parse(TEST1));
  say TEST2;
  print_record($parser->parse(TEST2));
}
```

◇

Fragment referenced in 8.

### The Parser

Here is where the work is really done. **Parse::Yapp** is given the following grammar. A parser is generated, contained in it's own module.

First off is the grammar's header. Here we define the symbols used in the rules which follow. We also add a small code block which contains a hash for holding the structure obtained from the parsed text.

⟨ *header* 11 ⟩ ≡

```
%token NUMBER
%token START
%token END
%token WORD
%token QUOTE
%token ESCAPED_QUOTE
%{
  my %record = (fields => {});
%}
```

◇

Fragment referenced in 17.

Here is the most important section, the rules for processing the input! For some rules we have also added action code blocks. We want to construct a data structure from the given input and in these action code blocks that final result is accumulated. Remember, the first rule is going to be called last, when the input is complete, so there we give a reference to a hash containing the result. This is the return value for the parse function found in the grammar's footer.

⟨ *rules* 12 ⟩ ≡

```
file : START id fields END          {$record{name} = $_[2]; \%record;}
;

id: WORD
;

words: WORD
 | words WORD
 | words ESCAPED_QUOTE WORD ESCAPED_QUOTE
;

field : WORD '=' NUMBER              {$record{fields}->{$_[1]} = $_[3]}
 | WORD '=' QUOTE words QUOTE        {$record{fields}->{$_[1]} = $_[4]}
;

fields : field
 | fields field
;
```

◇

Fragment referenced in 17.

The footer contains additional Perl code for the lexer, error handing, and a parse function which provides the main point of execution from code that wants to call the parser that has been generated from the grammar.

The lexer function is called repeatedly for the entire input. Regular expressions are used to identify symbols (the ones declared in the header) and pass them along for the rules processing.

⟨ *lexer* 13 ⟩ ≡

```
sub lexer{
  my($parser) = @_;
  $parser->YYData->{INPUT} or return('', undef);
  $parser->YYData->{INPUT} =~ s/^[ \t]//g;
  ##
  # send tokens to parser
  ##
  for($parser->YYData->{INPUT}){
      s/^([0-9]+)// and return ("NUMBER", $1);
      s/^({%)// and return ("START", $1);
      s/^(%})// and return ("END", $1);
      s/^(\w+)// and return ("WORD", $1);
      s/^(=)// and return ("=", $1);
      s/^(")//␣and␣return␣("QUOTE",␣$1);
      s/^(\\")//␣and␣return␣("ESCAPED_QUOTE",␣$1);
      s/^(\\\\)// and return ("WORD", $1);
  }
}
```

◇

Fragment referenced in 16.

The parse function is for the convenience of calling the generated parser from other code. **yapp** will generate a module and this will be the module's method used by other code to execute the parser against a given input.

Notice here that we are *squashing* white space, both tabs and spaces, using **tr**. This reduces all repeated tabs and spaces to a single one. The eases further processing since extra whitespace is just ignored, according to the rules we've been given.

Also notice the return value from parsing. In the rules section we provide a return value, a hash reference, in the final action code block executed.

⟨ *parse function* 14 ⟩ ≡

```
sub parse{
  my($self, $input) = @_;
  $input =~ tr/\t/ /s;
  $input =~ tr/ //s;
   $self->YYData->{INPUT} = $input;
  my $result = $self->YYParse(yylex => \\&lexer, yyerror => \\&error);
  return $result;
}
```

◇

Fragment referenced in 16.

This is really just about the most minimal error handling function there can be! All this does is print “syntax error”when the parser encounters a problem.

⟨ *error handler* 15 ⟩ ≡

```
sub error{
    exists $_[0]->YYData->{ERRMSG}
    and do{
        print $_[0]->YYData->{ERRMSG};
            return;
    };
    print "syntax␣error\n";
}
```

◇

Fragment referenced in 16.

⟨ *footer* 16 ⟩ ≡

```
⟨ lexer 13 ⟩
⟨ error handler 15 ⟩
⟨ parse function 14 ⟩
```

◇

Fragment referenced in 17.

`"perl/ch-2.yp"` 17≡

```
⟨ header 11 ⟩
%%
⟨ rules 12 ⟩
%%
⟨ footer 16 ⟩
```

◇

### Sample Run

```
$ yapp -m Ch2 perl/ch-2.yp; mv Ch2.pm perl; perl -I. ch-2.pl
{% id    field1="value1"    field2="value2"    field3=42 %}
{
          name => id
          fields => {
                    field1 =>  value1
                    field2 =>  value2
                    field3 =>  42
```

```
        }
}
{%  youtube title="Title \"quoted\" done" %}
{
        name => youtube
        fields => {
                field1 =>  value1
                field2 =>  value2
                field3 =>  42
                title =>  Title
        }
}
{%  youtube title="Title with escaped backslash \\" %}
{
        name => youtube
        fields => {
                field1 =>  value1
                field2 =>  value2
                field3 =>  42
                title =>  Title
        }
}
```

---

## References