YAPC::Europe::2002

Munich, September 18. - 20.

Munich Perl Mongers [http://munich.pm.org/] (Ed.)
Printing of these proceedings is sponsored by PetaMem.

YAPC::Europe::2002: Munich, September 18. - 20.

Chair: Norbert Grüner, Richard Foley
Finance: Richard Foley
Program Committee: Ann Barcomb, Regina Burbach, Richard Foley, Eike Grote, Norbert Grüner, Harald Jörg, Thoren Johne, Kevin Lenzo, Stefan Zehl
Webmaster of YAPC::Europe::2002 [http://www.yapc.org/Europe/2002/]: Thoren Johne
Sponsoring Pumpqueen: Regina Burbach
Spouse/kids event/evening Pumpqueen: Joy Foley
Proposals/Papers Pumpking: Harald Jörg
Proceedings: Harald Jörg, Bernhard Schmalhofer, Regina Aumeier
Venue: Ralph Leonhardt
Accomodation and Catering: Cornelia Rickl
Goodies: Michael Waider
Streaming: Jean-Christophe Zeus
YAPC Registration and Accounts: Kurt DeMaagd
Local IT support (LRZ): Günter May

Published September 2002, revised edition (PDF file only) with some typos fixed February 2004.

Our Sponsors

We are very grateful to the following institutions and companies for their support. They have made this conference possible or at least a lot more comfortable.

O'Reilly & Associates [http://www.oreilly.com/], O'Reilly Verlag [http://www.oreilly.de/]

ActiveState [http://www.activestate.com/]

InterNetX [http://www.internetx.de/]

Galileo Press [http://www.galileo-press.de/]

Deitel & Associates, Inc. [http://www.deitel.com/]

mobile.de AG [http://www.mobile.de/]

Addison-Wesley [http://www.addison-wesley.de/]

Fotango [http://opensource.fotango.com/]

PetaMem [http://www.petamem.com/]

Leon Brocard [http://www.nanoware.org/]

Leolo IT & Media Consulting [http://www.leolo.com/]

Leibniz-Rechenzentrum [http://www.lrz-muenchen.de/]

Manning Publications Co. [http://www.manning.com/]

Max-Planck-Institut für Astrophysik [http://www.mpa-garching.mpg.de/]

Table of Contents

Timetable

Room S0320 - fotango	Room S1128 - Nanoware	Room S0144 - mobile.de AG	Foyer - leolo
FOTANGO :: OPEN SOURCE	nanoware.org	mobile.de	leolo IT & MEDIA Consulting

Tuesday 17 September 2002

Time	
12:00	equipment setup
15:00	reception opens for early arrivals
19:00	welcome early arrivals at gasthof
23:00	begin of a hard days night :)

Wednesday 18 September 2002

reception opens at 7:30 and closes at 18:00			
Time	**Room S0320**	**Room S1128**	**Room S0144**
9:00	Phoenix - Practical Web Programming (240 min)	Davies - Developing GUI Applications with Gtk.pm (180 min)	Overmeer - E-Mail Processing with Perl (180 min)
10:30	morning coffee		
11:00	continue	continue	continue
12:30	lunch		
13:30	continue	Schwern - Advanced Testing (180 min)	Visser - wxPerl -- Another GUI for Perl (180 min)
14:30	Boumans - Introduction to OO Programming (120 min)	continue	continue
15:30	afternoon tea		
16:00	continue	continue	continue
17:00	intentionally left blank :)		
18:00	BOFs (perltrainers, perlmonks, perlbug)		
19:00	speakers dinner		
23:00	begin of a hard days night :)		

Thursday 19 September 2002

reception opens at 8:00 and closes at 18:00			
Time	**Room S0320**	**Room S1128**	**Room S0144**
9:00	Foley / Gruener - Opening speech (15 min)	intentionally left blank :)	
9:15	Bergman - Perl 5.8 At Last (45 min)	Jacobsson - Imaging with Imager (45 min)	West - Creating Dynamic Sites in a Flash with the Template Toolkit (45 min)
10:00	Clark - When Perl is not Quite Fast Enough (30 min)	Dojcsak - OpenMMS - Telco Industry Meets Perl (30 min)	Duncan - Pixie (30 min)
10:30	morning coffee		
11:00	van der Sanden - Perl 5.10 (60 min)	Stowe - How to Integrate Anything with SOAP (60 min)	van Belleghem - Source Filters in Perl (60 min)
12:00	McCarroll - The Simulation and Visualisation of Objects in 3D Space (30 min)	Pauley - Case Study: Psychometric Testing with Perl (30 min)	Avery - ExtUtils::ModuleMaker (30 min)
12:30	lunch		
13:30	Klausner - The Dark Art of Obfuscation (60 min)	Merelo - Evolutionary Computation in Perl (120 min)	Jelinik - ELRIC - An Intelligent Autonomous Agent (60 min)
14:30	Johnson - Testing and Code Coverage (60 min)	continue	Szabo - How to Write Slow Algorithms Quickly in Perl ? (60 min)
15:30	afternoon tea		
16:00	Sanface - Can a Company Use Perl to Develop - and Sell - Commercial Tools? (30 min)	Lightning talks (30 min) • Duncan - We're not Building a F******g House! • Koenig - WAIT@oreilly.de • McWilliam - Sex, Flies and Microarrays • Schwern - Bull In A China Shop, Canary In A Coal Mine	Beckert - How to Build Large Scale Websites/Web Applications with Embperl 2.0 (45 min)
16:30	White - NSRL Distributed Hashing with Sexeger Improvements (30 min)	Boumans - POE for Beginners (90 min)	continue
16:45	continue	continue	Brande - Send and Receive SMS Messages Using a GSM Modem (75 min)
17:00	Conway - Perl 6 Prospective (60 min)	continue	continue
18:00	BOFs (perltrainers, perlmonks, perlbug)		
19:00	p5p dinner		
23:00	begin of a hard days night :)		

Friday 20 September 2002

reception opens at 8:00 and closes at 18:00			
Time	**Room S0320**	**Room S1128**	**Room S0144**
9:00	Sugalski - Inside Parrot (45 min)	Laun - ELEKTRA Peripheral Simulation (60 min)	Bergman - Threads in Perl 5.8 (45 min)
9:45	Boumans - CPANPLUS - Beyond CPAN.pm (45 min)	continue	West - Introduction to Net::DNS (45 min)
10:30	morning coffee		
11:00	Juarez - Security in Perl Scripts (60 min)	Randall - On Topic and Topicalizers in Perl 6 (60 min)	Duncan - OpenFrame Application Server (60 min)
12:00	Larry Wall - the Science of Perl (30 min)	intentionally left blank :)	
12:30	lunch		
13:30	Conway - Quantum::Superpositions (120 min)	Brand - Smoking Bleadperl (60 min)	Payrard - Nemo: TeXmacs the Scientific Editor with Perl (75 min)
14:30	continue	Randall - Tagmemics (60 min)	continue
14:45	continue	continue	Brocard - The Acme::* Modules (45 min)
15:30	afternoon tea		
16:00	Schwern - Tales Of Refactoring: Remaking MakeMaker and Other Horrors (45 min)	Overmeer - Creepy Featurism (45 min)	Fowler - Extending the Template Toolkit (45 min)
17:00	McCarroll - Auction (45 min)	intentionally left blank :)	
17:45	Foley / Gruener - Closing speech (15 min)	intentionally left blank :)	
18:30	clean up		
19:30	organisers dinner		
23:00	begin of a hard days night :)		

Saturday 21 September 2002

Time	
8:00	alpine wanderung
19:00	oktoberfest table
23:00	bearing home... pretty drunk perhaps :)

Part I. Tutorials

Evolutionary Computation in Perl

Juan J. Merelo Guervós `<jmerelo (AT) geneura.ugr.es>`

Introduction: an evolutionary algorithm step by step

Abstract

A gentle introduction to evolutionary algorithms is done in this chapter. After a brief show-and-tell, the chapter describes step by step the architecture and mechanics of an evolutionary algorithm, from the "genetic" operators, on to the selection operations and concepts related to it, and up to a canonical genetic algorithm, a particular example of an evolutionary algorithm. Examples illustrates concepts. All is well.

Warning

The programs in this tutorial have been tested with Perl 5.6.1.633, as downloaded from ActiveState [http://www.activestate.com] in a Windows 98 system. And yes, I accept condolences for that. I didn't have any Linux machine handy during my holidays. Half way through the tutorial, I downloaded Perl 5.8.0 for CygWin; some examples also work with that version, and I guess the rest should have no problem.

Introduction to evolutionary computation

If you have not been in another planet (and without interplanetary internet connection), you probably have a (maybe vague) idea of what evolutionary computation is all about.

The basic idea is surprisingly simple, but incredibly powerful: an algorithm that tries to obtain *good enough* solutions by creating a population of data structures that represent them, and evolve that population by changing those solutions, combine them to create new ones, and, on average, make the better-suited survive and the worst-suited perish. That is the way Evolution of Species, as described by Darwin, has worked for a long time, so, why shouldn't it work for us?

It so happens that it does: evolutionary computation spans a family of algorithms, differing in details such as the way of selecting the best, how to create new solutions from existing ones, and the data structures used to represent those solutions. Those algorithms are called *evolution strategies*, *genetic algorithms*, *genetic programming*, or *evolutionary programming*, although they correspond to the same basic algorithmic skeleton.

Applications of evolutionary algorithms are found everywhere, even in the *real world* [http://everything2.com/index.pl?node_id=1114530], and range from entertainment (playing Mastermind [http://geneura.ugr.es/~jmerelo/GenMM]), to generating works of art like the eArtWeb [http://www.liacs.nl/~jvhemert/eartweb/] does, to more mundane, but interesting nonetheless, things like assigning telecommunication frequencies [http://evonet.dcs.napier.ac.uk/evoweb/resources/books_journals/bjp343.html], designing timetables [http://citeseer.nj.nec.com/fang94genetic.html], or scheduling tasks in an operating system [http://www.iit.edu/~elrad/esep.html#esep].

This tutorial will be divided in two parts: the first will be devoted to explaining the guts (and glory) of an evolutionary algorithm, by programming it from scratch, introducing new elements, until we arrive at the canonical classical genetic algorithm. The second part will use an existing evolutionary computation library, called `Algorithm::Evolutionary`, to design evolutionary computation applications by using XML, Perl, and taking advantage of the facilities Perl modules afford us.

X-Men approach to Evolutionary computation

Warning

The programs shown here have not been optimized for efficiency or elegancy, but rather for clarity and brevity. Even so, they can probably be improved, so I would like to hear [mailto:jmerelo at geneura.ugr.es] any comment or criticism you have on them.

Although Darwin's view of forces at work in the evolutionary process seems quite simple, putting them in black on white in an actual algorithm is something completely different. What kind of modifications should be applied to the data structures? What do we do with modified data structures? Should we put them in the population, just like that? Should we (gulp), like, off some other member of the population? If so, which one?

Historically, the first solutions to this conundrum seemed to be: create a population of possible solutions, take a member of the population, change it until you obtain something better (from the point of view of how close it is to the solution, that is, its *fitness*), and then eliminate one of the least fit members of the population, possible the least fit. That way, every time you introduce a new member of the population you are going one step up the evolutionary ladder (just like the X-Men [http://www.xmenunlimited.com/].)

We will work, from now on, on the following problem: evolve a population of ASCII strings so that they are as close as possible to the string `yetanother`. The *fitness* of each (initially random) string will be the difference between the ASCII values of the letter in each position and the ASCII value of the letter in the target string. Optimal strings will have 0 fitness, so the process will try to minimize fitness. The job is done by the following Perl program (`1stga.pl`, whitespace and POD comments have been suppressed):

```
#Declare variables ## see 1.
my $generations = shift || 500; #Which might be enough
my $popSize = 100; # No need to keep it variable
my $targetString = 'yetanother';
my $strLength = length( $targetString );
my @alphabet = ('a'..'z'); ## see 1.
my @population;
#Declare some subs (not the best place to do it, but we are going to
#need them ## see 2.
sub fitness ($;$) {
  my $string = shift;
  my $distance = 0;
  for ( 0..($strLength -1)) {
    $distance += abs( ord( substr( $string, $_, 1)) - ord( substr( $targetString, $_, 1)));
  }
  return $distance; ## see 2.
}
sub printPopulation {
  for (@population) {
    print "$_->{_str} -> $_->{_fitness} \n";
  }
} ## see 3.
sub mutate {
  my $chromosome = shift;
  my $mutationPoint = rand( length( $chromosome->{_str}));
  substr( $chromosome->{_str}, $mutationPoint, 1 ) = $alphabet[( rand( @alphabet))]; ## see 3.
}
#Init population ## see 4.
for ( 1..$popSize ) {
  my $chromosome = { _str => '',
                     _fitness => 0 };
  for ( 1..$strLength ) {
    $chromosome->{_str} .= $alphabet[( rand( @alphabet))];
  }
  $chromosome->{_fitness} = fitness( $chromosome->{_str} );
  push @population, $chromosome; ## see 4.
}
#Print initial population
printPopulation();
#Sort population
@population = sort { $a->{_fitness} <=> $b->{_fitness} } @population;
#Go ahead with the algorithm ## see 5.
for ( 1..$generations ) {
  my $chromosome = $population[ rand( @population)];
  #Generate offspring that is better
  my $clone ={};
  do {
    $clone = { _str => $chromosome->{_str},
               _fitness => 0 };
    mutate( $clone );
    $clone->{_fitness} = fitness( $clone->{_str} );
  } until ( $clone->{_fitness} > $chromosome->{_fitness});
  #Substitute worst
  $population[$#population]=$clone;
  #Re-sort
  @population = sort { $a->{_fitness} <=> $b->{_fitness} } @population;
  #Print best
  print "Best so far: $population[0]->{_str} => $population[0]->{_fitness} \n";
  #Early exit
  last if $population[0]->{_fitness} == 0;
} ## see 5.
```

1. This is the initial setup for the evolutionary algorithm; it simply declares a group of variables: the number of *generations*, that is, the number of iterations that are going to be done if the solution is not found, the size of the *population*, or the number of different data structures present in the population, and several other constants that will be used through the program.

2. This function computes fitness. And yes, it could be done in a single line.

3. This is the only variation operator we are going to use in this iteration of the evolutionary algorithm: it mutates a single charactere in a random position in the string, substituting it by another random character.

4. The population is initialized with random strings; at the same time, the data structure used for *chromosomes*, which is the conventional name the stuff that evolves is called, is also introduced. From the point of view of evolutionary computation, a *chromosome* is anything that can be changed (mutated) and evaluated to be assigned a fitness, and fitness is anything that can be compared. In most cases is a real number, but in some cases it can be something more complicated: a vector of different values, for instance. The data structure used for evolution is really unimportant, although, traditionally, some data structures, such as binary strings, floating-point vectors, or trees, have been used used; in many cases, also, there is a mapping between the data structure evolved and the data structure that is a solution to the problem: for instance, we might want to use binary strings to represent 2D floating point vectors, which are solutions to a numeric optimization problems. All in all, the representation issue has been the origin of endless wars in the field of evolutionary computation.

   ### Tip

   Use the data structure that best suits your expertise, tickles your fancy, or the one that is closest to the problem you want to solve. Testing different data structures for performance will not hurt you, either.

5. This is the *meat* of the program, the loop that actually does the evolution. Takes a random element of the population, creates a copy of it, mutates this copy until it finds a new string whose fitness is better than the original, which is then inserted in the population eliminating the worst, which probably deserved it.

This program, when run, produces this output:

```
cekyhtvvjh -> 97
mwehwoxscv -> 82
lalmbnbghi -> 81
[More like this...]
Best so far: vowjmwwgft => 41
Best so far: vowjmwwgft => 41
Best so far: vowjmwwgft => 41
[Maaany more like this...]
```

There are several problems with this algorithm. First, the population is not really used, and it is not actually needed. It is actually a hillclimbing algorithm, and very ineffective at that, since it takes an element, improves it a bit, puts it back into the population, takes another one... it would be much better to just take a random string, and start to change it until it hits target, right? In any case, since it is using a random mutation, what we are performing is basically random search over the space of all possible strings. Not an easy task, and this is the reason why the solution is usually not found, even given several hundred thousands iterations.

Tip

Blind mutation usually takes you nowhere, and it takes a long time to do so.

This indicates there is something amiss here; even if nature is a blind watchmaker [http://www.world-of-dawkins.com/Dawkins/Work/Books/blind.htm], it has the help of a very powerful tool: *sex*. And that is what we will use in the next section.

Here comes sex

The power of recombination

As we have seen, having a population and mutating it only takes you so far; there must be something more in Evolution that makes possible to create efficient structures and organisms. And one of these things is probably sex: after fusion of male and female genetic material, recombination takes place, so that the resulting organism takes some traits from each of its parents. In evolutionary computation, the operator that combines "genetic material" from two parents to generate one or more offspring is called *crossover*.

In its simplest form, crossover interchanges a chunk of the two parent's string, spawning two new strings.

Table 1. Two-point crossover on a string

Parent 1	xxxxxxxxxx
Parent 2	yyyyyyyyyy
Offspring 1	xxyyyyxxxx
Sibling 2	yyxxxxyyyy

The exact form of the crossover will obviously depend on the data structure we are using; in some cases it might not even be possible; but the general idea is to combine two or more solutions, so that whatever is good about them mingles, to (maybe) give something even better. Since recombination is blind, the result might be better or not, but it is quite enough that combination yields something better some times for climbing up the evolutionary ladder.

Crossover will be moved, along with the other utility functions, to a small module called `LittleEA.pm` [2ndga.pl], and takes the following form:

```
sub crossover {
  my ($chr1, $chr2) = @_;
  my $crossoverPoint = int (rand( length( $chr1->{_str})));
  my $range = int( rand( length( $chr1->{_str}) - $crossoverPoint + 1));
  my $str = $chr1->{_str};
  substr( $chr1->{_str}, $crossoverPoint, $range,
          substr( $chr2->{_str}, $crossoverPoint, $range));
  substr( $chr2->{_str}, $crossoverPoint, $range,
          substr( $str This is a possible implementation of a simple string
crossover, with two parents and two offspring. Both parameters are
passed by reference, and offspring take the place of parents. , $crossoverPoint, $range ));
}
```

. A crossover point is chosen randomly, and them, a length to swap that cannot be bigger that the total length of both strings. The characters spanned by that range are swapped between the two chromosomes. Since both parents have the same length, it does not matter which parent's length is used to generate the random crossover point; obviously, if variable-length strings are used, the minimal lenght will have to be used; for more complicated data structures, markers, or "hot points", are used sometimes.

Crossover is used in the following program (`2ndga.pl`; some parts have been suppresed for brevity):

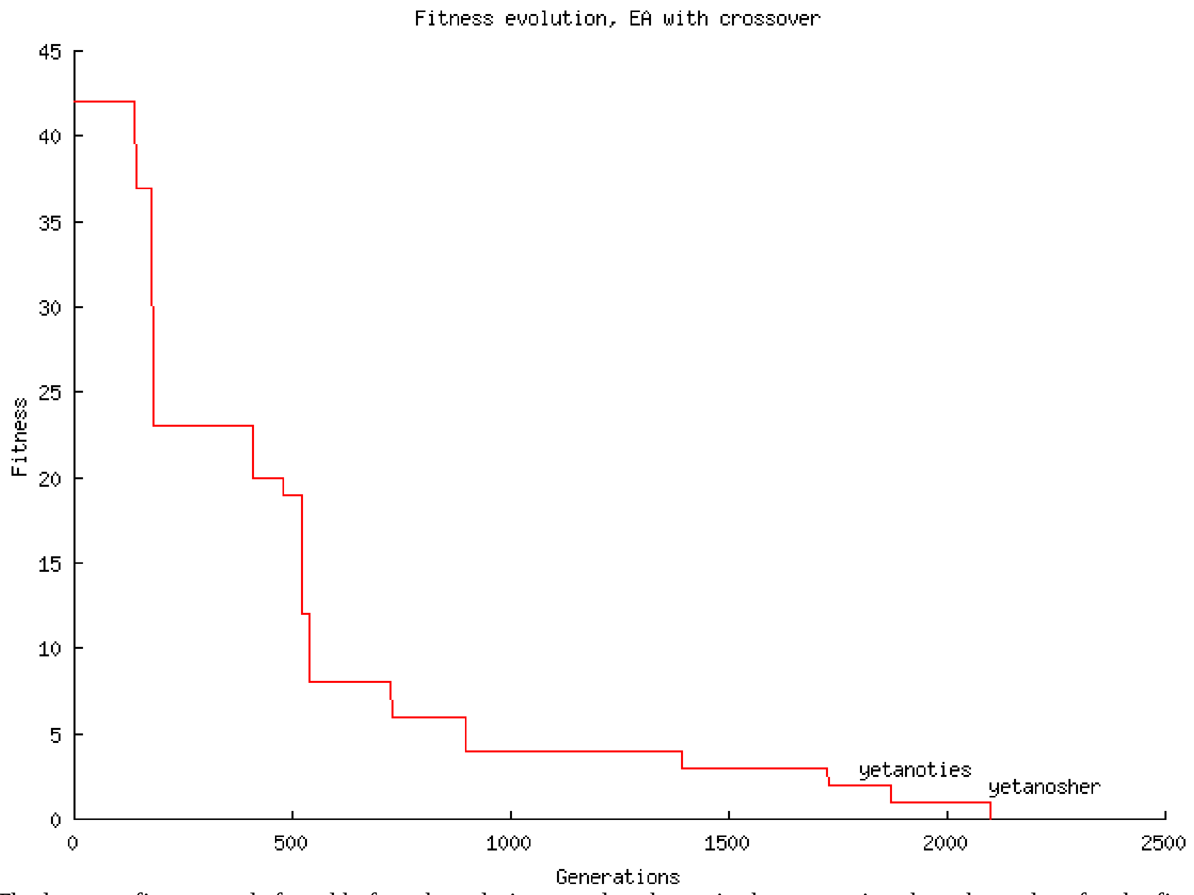
```
require "LittleEA.pm";
my $generations = shift || 500; #Which might be enough
my $popSize = 100; # No need to keep it variable
my $targetString = 'yetanother';
my $strLength = length( $targetString );
my @alphabet = ('a'..'z');
sub fitness ($;$) {
  my $string = shift;
  my $distance = 0;
  for ( 0..($strLength -1)) {
    $distance += abs( ord( substr( $string, $_, 1)) - ord( substr( $targetString, $_, 1)));
  }
  return $distance;
}
my @population = initPopulation( $popSize, $strLength, \@alphabet );
printPopulation( \@population);
@population = sort { $a->{_fitness} <=> $b->{_fitness} } @population; ## see 1.
for ( 1..$generations ) {
  my $chr1 = $population[ rand( @population)];
  my $chr2 = $population[ rand( @population)];
  #Generate offspring that is better
  my $clone1 ={};
  my $clone2 ={};
  do {
    $clone1 = { _str => $chr1->{_str},
         _fitness => 0 };
    $clone2 = { _str => $chr2->{_str},
         _fitness => 0 };
    mutate( $clone1, \@alphabet );
    mutate( $clone2, \@alphabet );
    crossover( $clone1, $clone2 );
    $clone1->{_fitness} = fitness( $clone1->{_str} );
    $clone2->{_fitness} = fitness( $clone2->{_str} );
  } until ( ($clone1->{_fitness} < $population[$#population]->{_fitness}) ||
           ($clone2->{_fitness} < $population[$#population]->{_fitness}));
  if ($clone1->{_fitness} > $population[$#population]->{_fitness}) {
    $population[$#population]=$clone1;
  } else {
    $population[$#population]=$clone1;
  }
  @population = sort { $a->{_fitness} <=> $b->{_fitness} } @population;
  print "Best so far: $population[0]->{_str}\n";
  printPopulation( \@population ); ## see 1.
  last if $population[0]->{_fitness} == 0;
}
```

1. The main loop is very similar to the first example, except that now two parents, instead of only one, are generated randomly, then mutated to generate variation, and then crossed over. In this case, new offspring is generated until at least one is better than the worst in the population, which it eventually substitutes. This requisite is a bit weaker than before: in the previous program, a new chromosome was admitted in the population only if it was better than its (single) parent.

The output of this program, after running it typing **perl 2ndga.pl 100000** will be something like this

```
hnbsqpgknl -> 97
gaheewieww -> 92
[More like this...]
cwceluxeih
kdcseymlot [Strings before crossover]
cwceluxeot
kdcseymlih [And after]
=============
Best so far: zjrcstrhhk
[More stuff...]
Best so far: yetanother
yetanother -> 0
yetanotier -> 1
yetaoother -> 1
yeuanother -> 1
[And the rest of the initial population]
```

In fact, in most cases, a few thousands evaluations are enough to reach the target string. The fitness of the best individual proceeds typically as shown in the figure below:



The last two fittest words found before the solution are also shown in the generation they showed up for the first time. They are at a distance of 2 and 1 from the target string, respectively; in this case, solution was found after around 2100 iterations; with two new individuals generated each generation, that means 4200 evaluations were needed to hit target. Not a bad mark. Not very good either, but not too bad.

Summing up, looks like crossover has made a simple random search something something a bit more complicated, which combines information about search space already present in the population to find better solutions; population allows to keep track of the solutions found so far, and recombination combines them, usually for the better.

Tip

Sex, is after all, important. And, for many, evolutionary computation without crossover cannot really be called that way, but something else: stochastic population-based search, maybe.

Why does two point crossover work better than single-point crossover? For starters, the former is included by the latter (if the second point is the last character in the chromosome). Besides, it allows, in a single pass, the creation of complicated structures such as 101 from two chromosomes "000" and "111", that would need several applications of the operator and several intermediate with single-point crossover.

Fish market

Why only the very best should have offspring

Still, some incremental improvements can be made on this algorithm. So far, just the very last element in the population, the scum of the land, was eliminated in each iteration. But, very close to it, where others that didn't deserved the bits they were codified into. Ratcheting up evolutionary pressure might allow us to reach the solution a bit faster.

Besides, anybody who has seen a National Geographic documentary program or two knows that, very often, only the alpha male, after beating anybody else who dares to move in the pack, gets the chance to pass its genetic material to the next generation; some other less violent animals like peacocks have to boast the best of its feathers to be able to attract peahens (if that term exists, anyways). All in all, while many of the worst die, some others lead a very boring life, because they don't get the chance to mate.

These two sad facts of life lead us to the following improvement on the basic evolutionary algorithm:(`3rdga.pl`; some parts yadda yadda)

```
#Everything else is the same, except this loop
for ( 1..$generations ) {
  for ( my $i = 0; $i < 10; $i ++ ) {
    my $chr1 = $population[ rand( $#population/2)];
    my $chr2 = $population[ rand( $#population/2)];
    #Generate offspring that is better
    my $clone1 ={};
    my $clone2 ={};
    do {
      $clone1 = { _str => $chr1->{_str},
                  _fitness => 0 };
      $clone2 = { _str => $chr2->{_str},
                  _fitness => 0 };
      mutate( $clone1 );
      mutate( $clone2 );
      crossover( $clone1, $clone2 );
      $clone1->{_fitness} = fitness( $clone1->{_str} );
      $clone2->{_fitness} = fitness( $clone2->{_str} );
    } until ( ($clone1->{_fitness} < $population[$#population]->{_fitness}) ||
              ($clone2->{_fitness} < $population[$#population]->{_fitness}));
    if ($clone1->{_fitness} > $population[$#population]->{_fitness}) {
      $population[$#population]=$clone1;
    } else {
      $population[$#population]=$clone1;
    }
    @population = sort { $a->{_fitness} <=> $b->{_fitness} } @population;
  }
```

In this case, first, ten new chromosomes are generated each iteration, one in every iteration of the mutarion/crossover loop. This number is completely arbitrary; it corresponds to 10% of the population, which means we are not really introducing a very strong evolutionary pressure. Each time a new chromosome is introduced, population is resorted (and this could probably be done more efficiently by just inserting the new member of the population in the corresponding place and shifting back the rest of the array, but I just didn't want to add a few more lines to the listing). So, each generation, the ten worst are eliminated.

Besides, the elements that will be combined to take part (if they pass) in the next generation, are selected from the elite first half of the (sorted by fitness) population. That introduces an additional element of efficiency: we already know that what is being selected is, at least, above average (above median, actually).

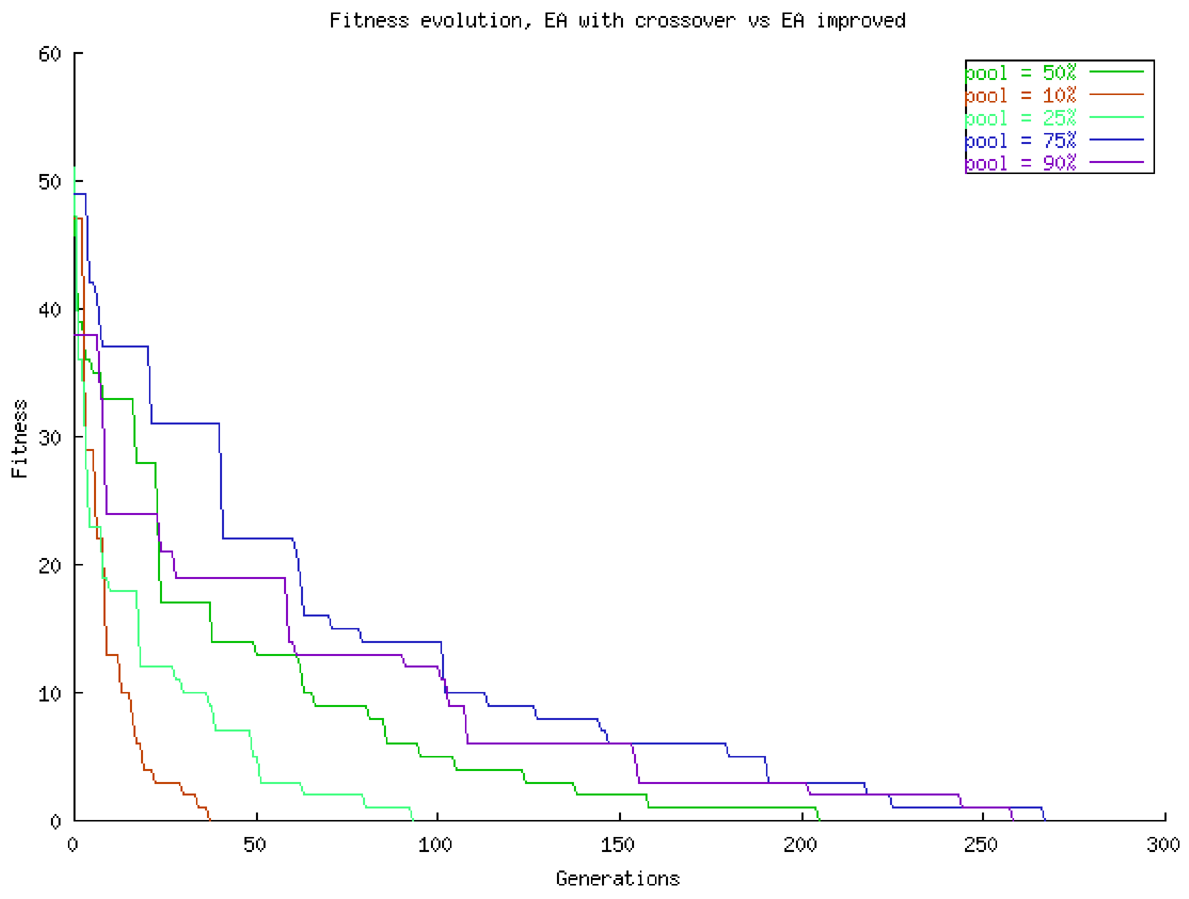
In fact, evolution proceeds faster in this case, but it does not become reflected in the number of iterations taken. Why? Because it decreases the number of iterations needed before offspring "graduates", that is, before they become better than the last element of the population. Thus, on the whole, this algorithm runs a bit faster, but the number of generations needed to reach target is more or less the same.

Tip

As cattle breeders have known for a long time, breeding using the best material available actually improves performance of evolutionary algorithms. Use it judiciously.

However, magic numbers such as the "10" and "half population" inserted in that program are not good, even in a Perl program. We can alter that program a bit, making the selective pressure variable, so that we can select the proportion of elements that will be selected for reproduction from the command line, giving the fourth example so far [4thga.pl].

With this example we can check what is the effect of reproduction selectivity in the time the algorithm takes to converge to the correct solution. We run it several times, with selectivity ranging from 10% (parents are selected from the best 10% of the population) to 90% (just 10% are considered not suitable for breeding). The effects are plotted in the following figure:



Comparison of the evolutionary algorithm for different reproductive selectivity. In green we can see the original line, in which the reproductive pool was the best half of the population. The most elitist selection strategy seems to be the hands-up winner, with the rest needing increasing number of evaluations to reach target with decreasing selective pressure.

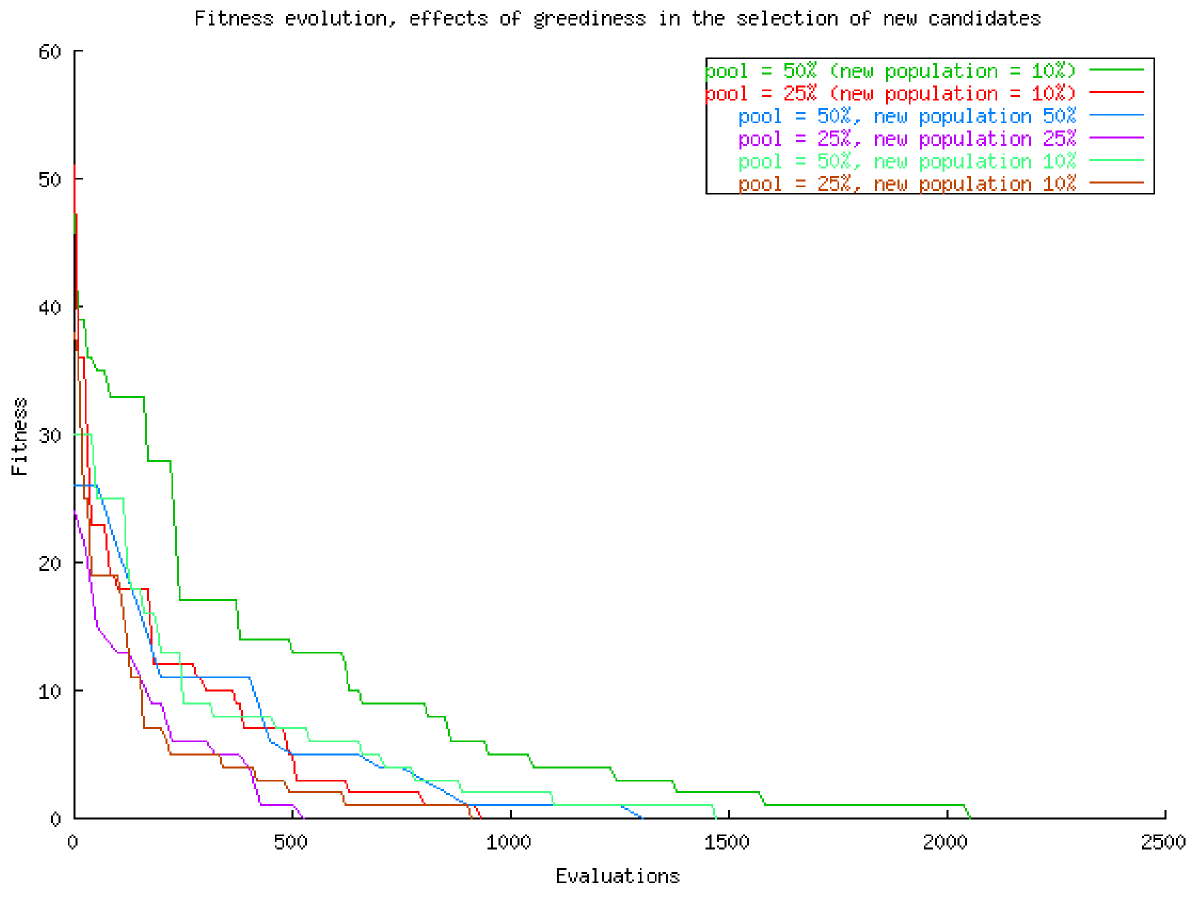
Looks like being selective with the reproductive pool is beneficial, but we should not forget we are solving a very simple problem. If we take that to the limit, choosing just the two best (in each generation) to mutate and recombine, we would be impoverishing the genetic pool, even as we would be *exploiting* what has been achieved so far. On the other hand, using all the population disregarding fitness for mutation and mating *explores* the space more widely, but we are reducing the search to a random one.

Important

Keeping the balance between exploration and exploitatin is one of the most valued skills in evolutionary computation. Too much exploration reduces an evolutionary algorithm to random search, and too much exploitation reduces it to hillclimbing.

The reduction of the *diversity* is something any practitioner is usually afraid, and is produced by too much exploitation, or, in another terms, by the overuse of a small percentage of individuals in the population. This leads to *inbreeding*, and, ultimately, to stagnation of the population in a point from which it cannot escape. These points are usually local minima, and are akin to the problems faced by reduced and segmented wildlife populations: its effects, and increased vulnerability to plagues, has been studied, for instance, in the Ngorongoro's population of lions.

So far, we have used a greedy strategy to select new candidates for inclusion in the population: only when an operation results in something better than the worst in the population, we give it the right to life, and insert in in the new population. However, we already have a mechanism in place for improving the population: use just a part of the population for reproduction, based on its fitness, and substitute always the worst. Even if, in each generation, we do not obtain all individuals better than before, it is enough to find at least a few ones that are better to make the population improve. That is what we do in the 5th example `5thga.pl` [5thga.pl]. The number of individuals generated in each iteration can be passed in the command line, and the reproductive selectivity can be also altered, as before. Results are plotted in the following figure:



(Bad) Comparison among an evolutionary algorithm in which, in each generation, 25 new elements are generated, chosen from the 25 best (pinkish) or 10 best (brown), 50 and 50 (blue) or 10 (light blue), and the previous results (where 10 new elements were renewed every generation). Not using a greedy algorithm to select new individuals, does not make results much worse, even if we take into account that we are not comparing the same thing here, because the actual number of evaluations until one better than before is reached is not measured; the number of evaluations shown. Other than that, the effect of renewing a different proportion of the population depends on how many we have chosen in advance to substitute the eliminated population: if the genetic pool is big, substituting more improves results (green vs dark and light blue lines, 10% and 50% substitution rate); if the genetic pool is small (25%), results look better if more chromosomes are substituted (pink vs brown and red). This might be due to the balance between exploration and exploitation: by generating too many new elements (50% substitution rate) we are moving the balance towards exploration, turning the algorithm into a random search; but if the pool is small (25%), generating too few would shift the balance toward exploitation, going to the verge of inbreeding; in that case, generating more individuals by crossover leads to better results.

Tip

As David Goldberg [http://gal4.ge.uiuc.edu/goldberg/d-goldberg.html] said in *Zen and the Art of Genetic Algorithms* [http://citeseer.nj.nec.com/context/17642/0], *let Nature be your guide*. There is no examination board in Nature that decides what's fit for being given birth or not; even so, species adapt to their environment along time. Evolutionary algorithms follow this advice.

Important

There are a couple of lessons to be learned from this last example: first, plain selection by comparison of each new individual with the current generation is enough for improving results each step; second, the balance between reproductive and eliminative selectivity is a tricky thing, and has a big influence in results. In some case, being very selective for reproduction and renewing a big part of the population might yield good results, but, in most cases, it will make the algorithm decay to random search and lead to stagnation.

The Canonical Genetic Algorithm

The Simple Genetic Algorithm (more or less), as described by David Goldberg.

In the early eighties, David Goldberg published a book, Genetic Algorithms in search, optimization, and machine learning [http://www.amazon.com/exec/obidos/ASIN/0201157675/perltutobyjjmere]. In this book he describes what makes genetic algorithms work, and introduces the simple genetic algorithm: an evolutionary algorithm based on binary strings, with crossover along with mutation as variation operator, and fitness-proportionate selection. This is the algorithm we implement in the next example (`canonical-ga.pl` [canonical ga.pl].

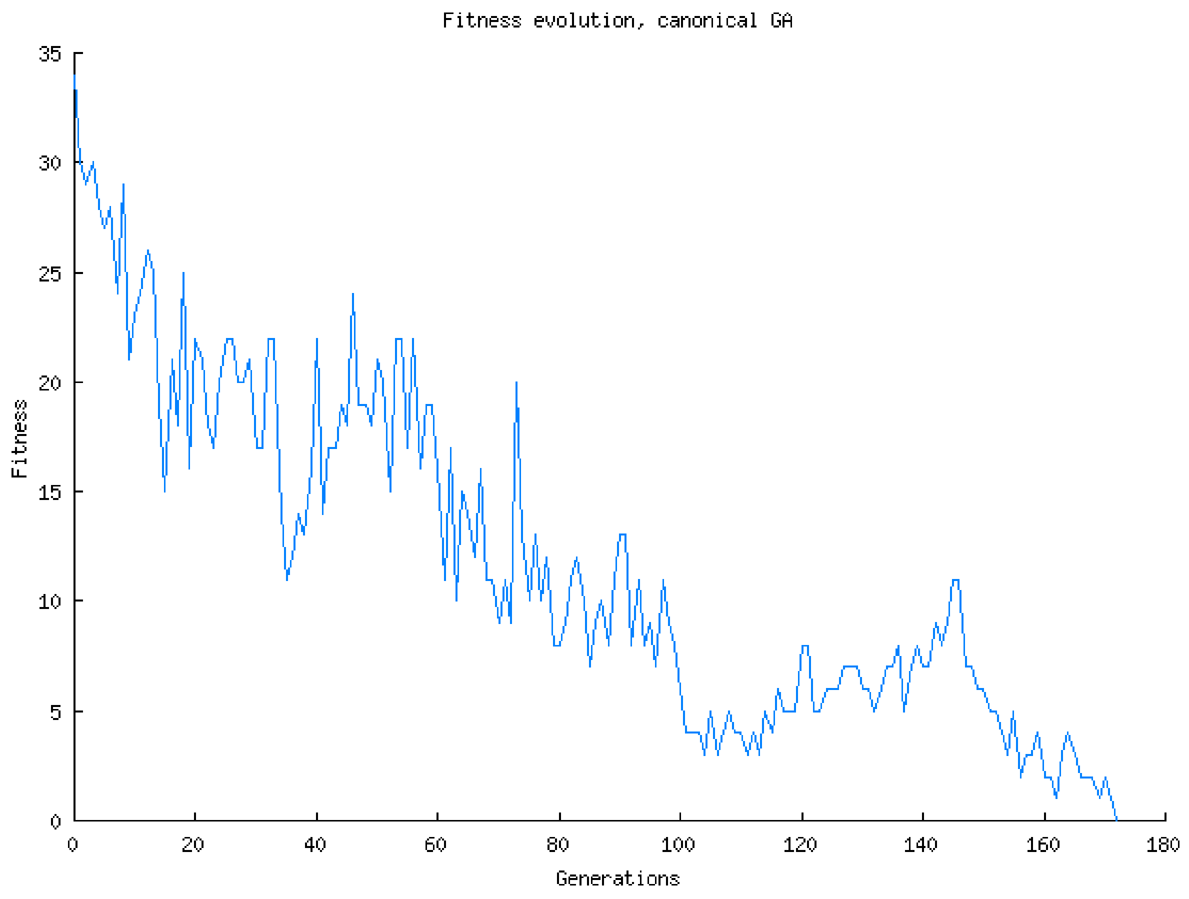
```
use Algorithm::Evolutionary::Wheel;
require "LittleEA.pm";
my $generations = shift || 500; #Which might be enough
my $popSize = 100; # No need to keep it variable
my $targetString = 'yetanother';
my $strLength = length( $targetString );
my @alphabet = ('a'..'z');
sub fitness ($;$) {
  my $string = shift;
  my $distance = 0;
  for ( 0..($strLength -1)) {
    $distance += abs( ord( substr( $string, $_, 1)) - ord( substr( $targetString, $_, 1)));
  }
  return $distance;
}
my @population = initPopulation( $popSize, $strLength, \@alphabet );
printPopulation( \@population);
@population = sort { $a->{_fitness} <=> $b->{_fitness} } @population;
for ( 1..$generations ) {
  my @newPop; ## see 2.
  my @rates;
  for ( @population ) {
    push @rates, 1/$_->{_fitness};
  } ## see 2.
  my $popWheel=new Algorithm::Evolutionary::Wheel @rates; ## see 3.
  for ( my $i = 0; $i < $popSize/2; $i ++ ) {
    my $chr1 = $population[$popWheel->spin()];
    my $chr2 = $population[$popWheel->spin()]; ## see 3.
    my $clone1 = { _str => $chr1->{_str},
                   _fitness => 0 };
    my $clone2 = { _str => $chr2->{_str},
                   _fitness => 0 };
    mutate( $clone1, \@alphabet );
    mutate( $clone2, \@alphabet );
    crossover( $clone1, $clone2 );
    $clone1->{_fitness} = fitness( $clone1->{_str} );
    $clone2->{_fitness} = fitness( $clone2->{_str} );
    push @newPop, $clone1, $clone2;
  }
  @population = sort { $a->{_fitness} <=> $b->{_fitness} } @newPop;
  print "Best so far: $population[0]->{_str}\n";
  printPopulation( \@population );
  last if $population[0]->{_fitness} == 0;
}
```

1. Declaration of a class that belongs to the `Algorithm::Evolutionary` module. This module will be used extensively in the second chapter; so far, if you feel the need, download it from the SF web site [http://sourceforge.net/projects/opeal]. This module is for creating roulette wheels that are biased with respect to probability: the probability that the "ball" stops at one of its slots is proportional to its probability.

2. The probabilities for the wheel are created, taking into account fitness. Since, in this case, lower fitness is better, fitness has to be inverted to create the roulette wheel; that way, individuals with lower fitness (closest to the target string) will have a higher chance of being selected

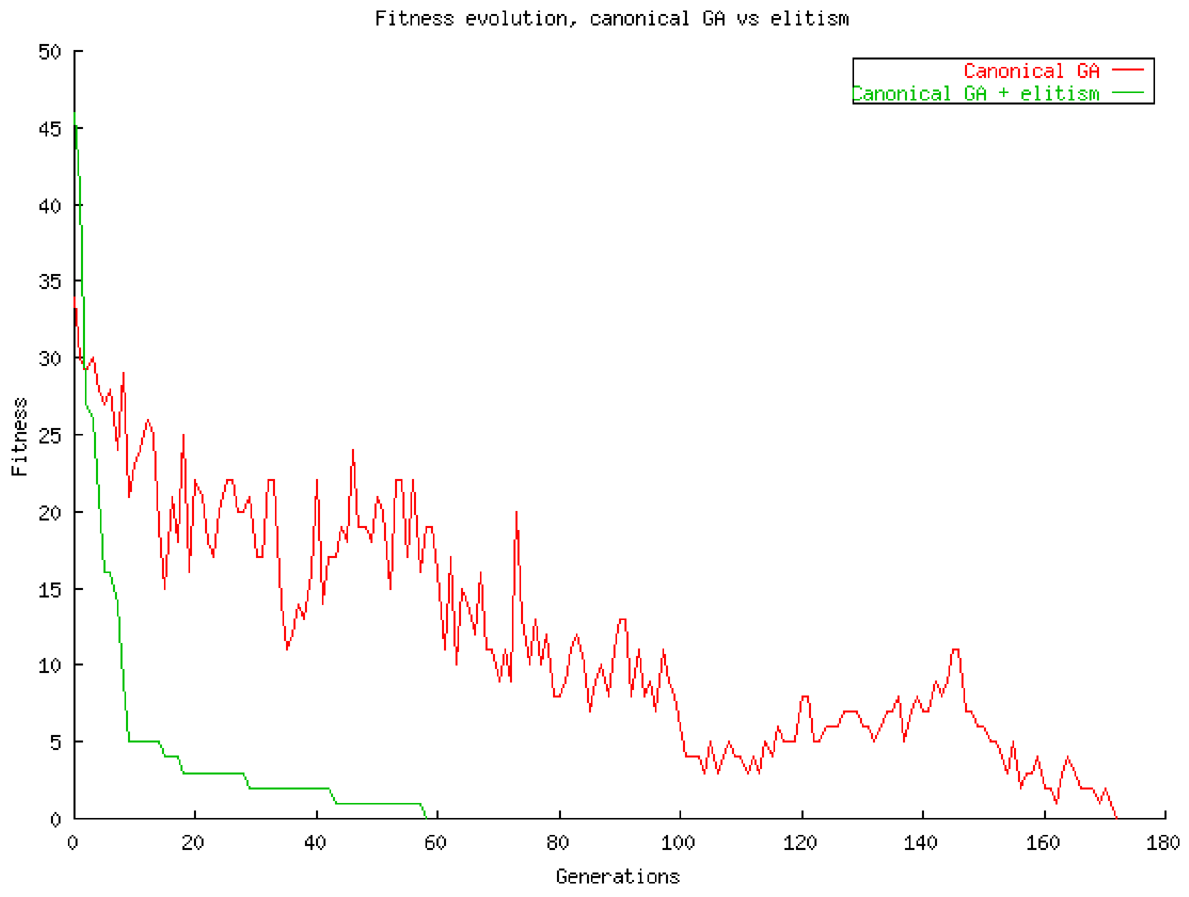
3. A `Wheel` object is created, and used later on to select the individuals that are going to be cloned and reproduced.

The canonical genetic algorithm is the benchmark against any new algorithm will be measured; it performs surprisingly well along a wide range of problems, but, in this case, it is worse than our previous example, as is shown in the next plot:



Evolution of the fitness of the best individual for the canonical GA. It needs 160 generations (in this case) to reach the optimum, which is worse than the best cases before. Actually, in simple problems, strategies that favor exploitation over exploration sometimes are more successful than the canonical GA, however, this is always useful as a first approximation. It should be noted also that, unlike previous examples, since the best of the population are not kept from one generation to the next, fitness can actually decrease from one generation to the next.

The canonical genetic algorithm manages to keep a good balance balanced between exploration and exploitation, which is one of its strong points; this makes it efficient throughout a wide range of problems. However, its efficiency can be improved a bit by just keeping a few *good* members of the population; that way, at least we make sure that the best fitness obtained does not decrease. That members will be called the *elite*, and the mechanism that uses them, *elitism*. We will introduce that mechanism in the next instance of the example, `canonical-ga-elitism.pl`, which is substantially similar to the previous one, except that the first two chromosomes of each generation are preserved for the next. Results obtained are shown in the following plot:



A comparison of the evolution of the canonical GA, with and without elitism. Elitism performs better, obtaining the same result in a third as many generation.

Surprisingly enough, this improvement comes at little cost; there is no significative disminution in diversity during the run, maintaining a good pool of different strings all the time (you can use the `freqs.pl` program to check that, if you feel like it).

Tip

Keeping track of the best-fit chromosomes, and reintroducing them at the next generation, improves performance if done wisely; without the cost of a loss of diversity that can degrade performance in more complex problems.

Even if now, the exact data structure that is being evolved is not that important, original genetic algorithms used, mainly for theoretical reasons (*respect the sieve of schemas*), binary strings. And one of the most widely used benchmarks for binary-string evolutionary algorithms (or simply GAs), is the "count ones" problem, also called "ONEMAX": in this problem, the target string consists of all ones. This problem is solved in the next program (`canonical-classical-ga.pl`)

```
use Algorithm::Evolutionary::Wheel; ## see 1.
require "LittleEA.pm";
my $numberOfBits = shift || 32;
my $popSize = 200; # No need to keep it variable
my @population; ## see 1.
sub fitness ($;$) {
  my $indi=shift;
  my $total = grep( $_ == 1, split(//,$indi ));
  return $total;
}
sub mutateBinary {
  my $chromosome = shift;
  my $mutationPoint = rand( length( $chromosome->{_str}));
  my $bit = substr( $chromosome->{_str}, $mutationPoint, 1 );
  substr( $chromosome->{_str}, $mutationPoint, 1, $bit?1:0 ); ## see 2.
}
for ( 1..$popSize ) {
  my $chromosome = { _str => '',
                     _fitness => 0 };
  for ( 1..$numberOfBits ) { ## see 2.
    $chromosome->{_str} .= rand() > 0.5?1:0;
  }
  $chromosome->{_fitness} = fitness( $chromosome->{_str} );
  push @population, $chromosome;
}
printPopulation( \@population );
@population = sort { $b->{_fitness} <=> $a->{_fitness} } @population;
do {
  my @newPop;
  my @rates;
  for ( @population ) {
    push @rates, $_->{_fitness};
  }
  my $popWheel=new Algorithm::Evolutionary::Wheel @rates;
  for ( my $i = 0; $i < $popSize/2; $i ++ ) {
    my $chr1 = $population[$popWheel->spin()];
    my $chr2 = $population[$popWheel->spin()];
    #Generate offspring that is better
    my $clone1 = { _str => $chr1->{_str},
                   _fitness => 0 };
    my $clone2 = { _str => $chr2->{_str},
                   _fitness => 0 };
    mutateBinary( $clone1 );
    mutateBinary( $clone2 );
    crossover( $clone1, $clone2 );
    $clone1->{_fitness} = fitness( $clone1->{_str} );
    $clone2->{_fitness} = fitness( $clone2->{_str} );
    push @newPop, $clone1, $clone2;
  }
  @population = sort { $b->{_fitness} <=> $a->{_fitness} } @newPop;
  #Print best
  print "Best so far: $population[0]->{_str}\n";
} until ( $population[0]->{_fitness} == $numberOfBits );
print "We're done\n";
printPopulation( \@population );
```

1. The first lines of the program differ a bit: it takes as an argument the number of bits, and the population is bigger. Fitness is also different: the `fitness` subroutine splits the binary strings to count the number of ones, which is returned.

2. The `mutateBinary` subroutine is also different: after selecting a position to mutate, it flips the bit in that position. A mutation operator that flips several bits could be thought of, but the same effect is achieved with several applications of the same operator. More complicated mutation operators use "hot spots" to mutate, or evolve the mutation rate, or change the probability of mutation for each locus in the chromosome. Sometimes, these strategies improve performance, some others are not worth the hassle.

As expected, this program finds the solution eventually; it is only shown here for historical reasons. Just by changing the fitness function, many problems that admit a binary codification could also be solved, from the MAXSAT optimization problem, to the well-known travelling salesperson problem.

Doing Evolutionary Algorithms with `Algorithm::Evolutionary`

And other Perl modules

Abstract

This chapter revisits the evolutionary algorithms we have seen so far, and then some, by using an evolutionary algorithm module (which might be in CPAN by the time you read this) called (what else?) `Algorithm::Evolutionary`. This module has been designed by the author to be flexible, integrated with XML, Perlish and easy to extend. We will also show how this library works with other Perl modules.

Introduction to evolutionary algorithms in Perl

So far, there have been many attempts to create evolutionary algorithm modules and programs in Perl; most have concentrated in implementing Genetic Programming, and some have been geared to a particular application, like the GlotBot [http://www.ocf.berkeley.edu/~jkunken/glot-bot/]. The closest thing one can get is the `AI::Gene` [http://search.cpan.org/author/AJGOUGH/AI-Gene-Sequence-0.21/] modules, which were intended for creating the basic infrastructure for an evolutionary algorithm devoted to fighting spam. The canonical genetic algorithm, implemented using `AI::Gene`, would be as follows: `cga-ai-gene.pl`:

```
use AI::Gene::AI::Gene::Simple; ## see 1.
package MyGene;
our @ISA = qw (AI::Gene::Simple);
sub render_gene {
  my $self = shift;
  return (join '', @{$self->[0]});
}
sub mutate_minor {
  my $self = shift;
  my $num = +$_[0] || 1;
  my $rt = 0;
  for (1..$num) {
    my $glen = scalar @{$self->[0]};
    my $pos = defined $_[1] ? $_[1] : int rand $glen;
    next if $pos >= $glen; # pos lies outside of gene
    my $token = $self->generate_token();
    $self->[0][$pos] = $token;
    $rt++;
  }
  return $rt;
} ## see 1.
package main;
use Algorithm::Evolutionary::Wheel;
my $generations = shift || 500; #Which might be enough
my $popSize = 100; # No need to keep it variable
my $targetString = 'yetanother';
my $strLength = length( $targetString ); ## see 2.
sub fitness ($;$) {
  my $ary = shift;
  my $distance = 0;
  for ( 0..($strLength -1)) {
    $distance += abs( ord( $ary->[$_]) - ord( substr( $targetString, $_, 1)));
  }
  return $distance; ## see 2.
}
sub printPopulation {
  my $pop = shift;
  for (@$pop) {
    print $_->render_gene(), " -> $_->[1] \n";
  }
} ## see 3.
sub crossover {
  my ($chr1, $chr2) = @_;
  my $length = scalar( @{$chr1->[0]});
  my $crossoverPoint = int (rand( $length));
  my $range = int( rand( $length - $crossoverPoint ));
  my @tmpAry = @{$chr1->[0]};
  @{$chr1->[0]}[ $crossoverPoint..($crossoverPoint+ $range)] =
      @{$chr2->[0]}[$crossoverPoint..($crossoverPoint+ $range)];
  @{$chr2->[0]}[ $crossoverPoint..($crossoverPoint+ $range)] =
    @tmpAry[ $crossoverPoint..($crossoverPoint+ $range)]; ## see 3.
}
my @population;
for ( 1..$popSize ) { ## see 4.
  my $chromosome = MyGene->new();
  $chromosome->mutate_insert( $strLength );
  $chromosome->[1] = fitness( $chromosome->[0] ); ## see 4.
  push @population, $chromosome;
}
printPopulation( \@population);
@population = sort { $a->[1] <=> $b->[1] } @population;
for ( 1..$generations ) {
  my @newPop;
  my @rates;
  for ( @population ) {
    push @rates, 1/$_->[1];
  }
  my $popWheel=new Algorithm::Evolutionary::Wheel @rates;
  for ( my $i = 0; $i < $popSize/2; $i ++ ) {
    my $chr1 = $population[$popWheel->spin()];
    my $chr2 = $population[$popWheel->spin()];
    my $clone1 = $chr1->clone();
    my $clone2 = $chr2->clone();
    $clone1->mutate_minor(1);
    $clone2->mutate_minor(1);
    crossover( $clone1, $clone2 );
    $clone1->[1] = fitness( $clone1->[0] );
    $clone2->[1] = fitness( $clone2->[0] );
    push @newPop, $clone1, $clone2;
  }
  @population = sort { $a->[1] <=> $b->[1] } @newPop;
  print "Best so far: ", $population[0]->render_gene(), "\n";
  printPopulation( \@population );
  last if $population[0]->[1] == 0;
}
```

1. After a somewhat peculiar declaration of the class (needs to be done this way because it is where it is installed by default, maybe it is a bug), we have to subclass the basic `AI::Gene` class, first to create a rendering of the chromosome in the same way as our previous examples, and then to change the basic definition of mutation, which originally used "character classes"; something we don't need here. It needs to change no further, since it uses as basic alphabet the english lowercase alphabet, as we did in our original programs.

2. The data structure used to represent the chromosome is an array-of-arrays, instead of a hash; the first component of the array contains the chromosome; this fitness function takes that chromosome array, and returns fitness. The second component of the array will be used for the fitness, as will be seen later on.

3. Crossover is also modified according to the new data structure; arrays are used instead of strings. The rest of the program is not highlighted, but has also been modified according to the new data structure.

4. Initializing the chromosome means now creating a new object of the new class `MyGene`, and then initializing it via the provided `AI::Gene::mutate_insert` method, that inserts new characters up to the required number.

5. Mutation is performed via the provided `AI::Gene::mutate_minor`, that changes a single character (given as parameter). The rest of the program is the same as before, except for the particular methods used to print the chromosome.

All in all, some useful code is provided by `AI::Gene`, but, still, you have to write a substantial part of the program. Besides, functionally, mutation operators are functions applied to chromosomes, not part of the chromosome interface, and, as such, should be considered independent classes. In the way `AI::Gene` is designed, any new mutation-like operator can be added by subclassing the base class, but it will not be a part of the class, unless you overload one of the existing functions (like `mutate_minor`). And, finally, it lacks any classes for doing algorithm-level stuff: selection, reproduction, which have to be done by the class client.

Note

`AI::Gene` can be a good CPAN starting point for evolutionary computation, but it has some way to go to become a complete evolutionary algorithm solution.

There are several other published tools you can use to perform genetic algorithms with Perl. Two of them,`AI::GA` [http://www.skamphausen.de/software/AI/ga.html]and `Algorithm::Genetic` [http://www.perlmonks.org/index.pl?node=Algorithm%3A%3AGenetic] are simple and straightforward implementations of a genetic algorithm. An article by Todor Zlatanov, *Cultured Perl: Genetic algorithms applied with Perl Create your own Darwinian breeding grounds* [http://www-106.ibm.com/developerworks/linux/library/l-genperl/], describes a system for doing Genetic Programming with Perl, and includes sample code; this article gets the most mentions in the Perl community. A library, MyBeasties [http://mybeasties.sourceforge.net], stands out among the rest. It is a very complex and general implementation for any kind of genotype, which, besides, has its own lenguage for describing genotypes and its transformations and evaluations. It features many classes for mutation and recombination, but it lacks classes for higher-level operations, and for implementing different kind of algorithms. Its learning curve is somewhat steep, anyhow.

Canonical GA with `Algorithm::Evolutionary`

After having dabbled with other languages for programming evolutionary computation, like C++ (for EO) [http://eodev.sourceforge.net], or JavaScript (for GAJS) [http://geneura.ugr.es/~jmerelo/GAJS.html], the author decided it was about time to program an evolutionary computation library in Perl. It was called initially OPEAL (for Original Perl Evolutionary Algorithm Library), but after some consultations, and previous to uploading it to CPAN (which, er, has not happened yet), it was renamed Algorithm::Evolutionary. It was intended as a complete and extensible evolutionary algorithm implementation, integrated with XML, and easy to learn, use and extend. It has been out there for about a year, and, right now, is available from SourceForge [http://sourceforge.net/projects/opeal] as a GPL module. Last released version is 0.5, which was released in summer 2002.

If you don't know XML, this is the moment to stop for a while and learn it. You can start by having a look at the O'Reilly XML site [http://www.xml.com], or by downloading some Perl modules that will help you through your pilgrimage. You can also subscribe to the Perl-XML [http://listserv.activestate.com/mailman/listinfo/perl-xml] mailing list.

Enough hype, and let's see what the boy is able to do. Download it, do the three-phrase spell **`perl Make-file.PL; make; make install`** (and, for the wary, **`make test`**), and you'll have it installed in your preferred modules directory. You can then run this program (`ea-ex1.pl`):

```
use Algorithm::Evolutionary::Experiment;
use Algorithm::Evolutionary::Op::Easy;
my $fitness = sub {
  my $indi = shift;
  my $total = grep( $_ == 1, split(//,$indi->Chrom() ));
  return $total;
};
my $ez = new Algorithm::Evolutionary::Op::Easy $fitness;
my $popSize = 100;
my $indiType = 'BitString';
my $indiSize = 32;
my $e = new Algorithm::Evolutionary::Experiment $popSize, $indiType, $indiSize, $ez;
print $e->asXML();
my $populationRef;
do {
  $populationRef = $e->go();
  print "Best so far: ", $populationRef->[0]->asString(), "\n";
} until ($populationRef->[0]->Fitness() == $indiSize);
print "Final\n", $e->asXML();
```

. Easy as breading butter, and twice as tasty, ain't it? Well, first of all, we are not doing the canonical GA, but a *steady-state* GA that, each generation, substitutes 40% of the population (a default value). The program goes like this: after loading needed classes, we declare the fitness function. In `Algorithm::Evolutionary`, the chromosome can be any data structure, but the actual data structure evolved is always accesible via the `Chrom` method. In this case, that method returns a string, that is dealt with as before, to return the total number of ones. After the fitness declaration, an `Algorithm::Evolutionary::Op::Easy` algorithm object is created. That object is passed to another `Algorithm::Evolutionary::Experiment` object, which contains all stuff needed to solve the problem: algorithm, plus population. You can print that experiment object as an XML document, which would look more or less like this:

```
<ea xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation='ea-alpha.xsd'
    version='0.3'>
<!-- Serialization of an Experiment object. Generated automatically by
     Experiment $Revision: 1.2 $ -->
    <initial>
<op name='Easy' ><op name='Bitflip' rate='1' >
 <param name='howMany' value='1' />
</op>
<op name='Crossover' rate='1' >
 <param name='numPoints' value='2' />
</op>
 <param name='selrate' value='0.4' /><code type='eval' language='perl'>
<src><![CDATA[{
    my $indi = shift @_;
    my $total = grep(($_ == 1), split(//, $indi->Chrom, 0));
    return $total;
}]]&gt;
 </src>
</code>
</op>
 </initial>
<!-- Population --><pop>
<indi   type='BitString'   >   <atom>0</atom>   <atom>1</atom>   <atom>1</atom>   <atom>1</atom>
<atom>1</atom>  <atom>1</atom>  <atom>0</atom>  <atom>0</atom>  <atom>0</atom>  <atom>0</atom>
<atom>0</atom>  <atom>1</atom>  <atom>1</atom>  <atom>0</atom>  <atom>0</atom>  <atom>1</atom>
<atom>0</atom>  <atom>1</atom>  <atom>0</atom>  <atom>0</atom>  <atom>1</atom>  <atom>0</atom>
<atom>1</atom>  <atom>1</atom>  <atom>0</atom>  <atom>1</atom>  <atom>1</atom>  <atom>1</atom>
<atom>0</atom> <atom>1</atom> <atom>1</atom> <atom>0</atom>
</indi>
<!-- more indis like this one -->
 </pop>
</ea>
```

. By default, the "Easy" operator includes Bitflip mutation and crossover, each with a rate of 1 (that means that they are applied with the same probability). Each one takes a parameter, with are passed via the tag `param`. The Easy operator takes also a parameter, and another code section, which is converted to a subroutine of the same name as included in the attribute 'eval'; the *language* attribute is included for future extensions. The source code within that tag does not look exactly the same as the one above, because it has been deparsed (using `B::Deparse`) from the original pointer-to-sub.

This is the kind of stuff that makes Perl unique; having a compiler/decompiler embedded in the same interpreter makes easy to serialize even complicated stuff, as data structures with pointers-to-function. I can't imagine how this could be done in C++, and it's probably impossible in Java too (Is it possible in Ruby or Python, I wonder?)

After the `initial` section, comes the `pop` section, that includes the components of the initially generated population. Each individual is enclosed by the tag `indi`, with a *type* attribute that indicates the class the individual belongs to, and them, one `atom` for every "atomic" component of the data structure.

That XML document can be retrieved back into a program by loading the file into a variable `$xml` and using this:**my $experiment= Algorithm::Evolutionary::Experiment->fromXML($xml);** .

However, as we said, this is not the canonical genetic algorithm. The program that implements it would be use the `CanonicalGA` class, like the example in `ea-ex2.pl`, which is exactly the same, except that, instead of declaring an `Easy` object, we declare a `CanonicalGA`. This object, besides implementing a canonical GA without elitism, uses `QuadXOver`, the crossover used before that takes two arguments by reference and returns offspring in the same arguments. The `Crossover` object takes arguments by value, not modifying them, and returns a single offspring.

A different problem might require a different fitness function, and probably different type of individuals. The default `Algorithm::Evolutionary` distribution includes four classes: `Vector`, for anything vectorial, from strings represented as vectors, through vector of floating point numbers, up to vectors of frobnicated foobars; `String`, with the `BinaryString` subclass, and `Tree`, for doing Genetic Programming or anything else that requires a tree data structure for representation. Using any of these data structures for solving a problem is left as an exercise to the student.

`Algorithm::Evolutionary` problems can be specified using Perl, but you can use an "universal reader" (`ea-ex3.pl`) [ea-ex3.pl] to read the description of the algorithm in XML.

```
#!perl
use strict;
use warnings;
use Algorithm::Evolutionary::Experiment;
my $xmlDoc = join("",<>);
my $e = Algorithm::Evolutionary::Experiment->fromXML($xmlDoc);
my $populationRef = $e->go();
print "Final\n", $e->asXML();
```

This reader has been listed here in its entirety, but, however, since the `CanonicalGA` which has been used in the previous example performs a single generation, it is quite limited, and only takes you so far. That is why we need to implement a whole genetic algorithm using `Algorithm::Evolutionary` classes (and see how they get reflected in the XML document).

Growing up: a whole evolutionary algorithm with `Algorithm::Evolutionary`

How to program and serialize all parts of an evolutionary algorithm

It is about time we go into more complex stuff, and, since many subalgorithms are already programmed into `Algorithm::Evolutionary`, we will use it. The not so-simple genetic algorithm is composed of a main loop that does, in sequence,

1. selection of the group of individuals that will be the parents of the next generation
2. application of genetic operators to those elements
3. insertion of those parents in the population
4. elimination of a part of the population, and
5. checking for termination conditions

, many of those parts can be delegated to subalgorithms. `Algorithm::Evolutionary` includes skeleton classes, `GeneralGeneration` and `FullAlgorithm`, with pluggable submodules, so that, on the basic schema, you can mix and match any combination of sub-algorithms. This is how it is done in the next example :`ea-ex4.pl`:

```
#!perl
use strict;
use warnings;
use Algorithm::Evolutionary::Experiment;
use Algorithm::Evolutionary::Op::Creator;
use Algorithm::Evolutionary::Op::Bitflip;
use Algorithm::Evolutionary::Op::Crossover;
use Algorithm::Evolutionary::Op::RouletteWheel;
use Algorithm::Evolutionary::Op::GeneralGeneration;
use Algorithm::Evolutionary::Op::DeltaTerm;
use Algorithm::Evolutionary::Op::FullAlgorithm;
my $numberOfBits = shift || 32;
my $popSize = 100;
my $fitness = sub {
  my $indi = shift;
  my $total = grep( $_ == 1, split(//,$indi->Chrom() ));
  return $total;
};
my $creator = ## see 2.
  new Algorithm::Evolutionary::Op::Creator( $popSize, 'BitString',
                                                   { length => $numberOfBits });
my $selector = new Algorithm::Evolutionary::Op::RouletteWheel $popSize;
my $mutation = new Algorithm::Evolutionary::Op::Bitflip;
my $crossover = new Algorithm::Evolutionary::Op::Crossover;
my $replacementRate = 0.4; #Replacement rate
my $generation =
  new Algorithm::Evolutionary::Op::GeneralGeneration( $fitness, $selector, ## see 2.
                                                     [$mutation,          $crossover],
                                                     $replacementRate ); ## see 3.
my $terminator = new Algorithm::Evolutionary::Op::DeltaTerm $numberOfBits, 0;
my  $algorithm  =  new  Algorithm::Evolutionary::Op::FullAlgorithm  $generation,  $terminator;  ##
see 3.
my $experiment = new Algorithm::Evolutionary::Experiment $creator, $algorithm; ## see 4.
print<<EOC;
<?xml version="1.0"?>
<experiment>
EOC
print $experiment->asXML(); ## see 4.
$experiment->go();
print $experiment->asXML(), "</experiment>";
```

1. Declaration of a `Creator`, which is factory class for algorithm individuals, `Bitstring`s in this case. This class takes as arguments the number of elements it will produce, the name of the class, and a hash that contains a list of named arguments, that will be passed to the class constructor. In this case, just the length of the chromosome is needed

2. A `Generation` includes a selector, which decides what elements of the population will be used for reproduction. In this case, `RouletteWheel` is chosen, the same reproductive method we have seen before: the elements of the population are selected with a probability proportional to its fitness. Another option would have been `TournamentSelect`, which takes a set of several individuals, and select only the best of them. It is a greedier way of selection, and its greediness depends on the number of elements in the tournament: the higher the number, the greedier. Crossover and mutation operators are declared in the next line; in `Algorithm::Evolutionary`, operators are independent classes, so that you are free to declare and use as many as you want. The operators are passed in an array ref to the `Generation`, and are selected according to *rate*: by default, each operator gets a rate of 1, and thus have the same probability. Rate can be changed during runtime, since it is an instance variable. Finally, the `$replacementRate` rules how many elements of the population will be substituted each generation. Finally, the object is declared, passing all the previous stuff as arguments.

3. Finally, the full algorithm itself, in all its majesty, is declared. It needs the previously-declared `$generation` object, plus a termination condition. And then, the two operators that are going to be applied sequentially to the population, the creator and then the algorithm, are used to create an `Experiment` object. This object takes as an argument operators that will be applied sequentially to the population; any operator whose `apply` method takes an array as an argument could be passed here. It could be possible, for instance, to use a creator, then an evolutionary algorithm, then a bridge, and then another kind of algorithm, population-based incremental learning, for instance, or simulated annealing, to improve the final result. This, of course, could be done in a single operator.

4. The algorithm is run, and its initial and final state is included in a well-formed XML document that is sent to standard output.

The good thing about having this XML output is that you can process it very easily, for instance, to pretty-print final population, using this XSLT stylesheet [final.xsl] to obtain this web page [ea-ex4.html]. The XML document can be used for post-algorithmical analysism, for interchange with other evolutionary algorithms, possibly written in other languages, or even for external data representation for parallel and distributed algorithms. For instance, the output of the algorithm can be converted to a combined HTML/SVG (Scalable Vector Graphics) [http://www.w3.org/TR/SVG/] document, which can be used for presentation straight away. It could also be imagined a "literate evolutionary computation" application that would mix the output of an evolutionary algorithm, with the description of the classes obtained via **pod2xml**, to create an XSLT stylesheet that would process output and create a document with output along explanation. This is left as an exercise to the reader.

Tip

XML is cool. 'Nuff said.

Extending `Algorithm::Evolutionary`

With a little help from our friend Perl

First, we are going to take advantage of a Perl module, to extend our library with new classes. There's a wealth of Perl modules out there, and many of them are devoted to working with data structures such as lists or strings. For instance, the `Algorithm::Permute` class comes in handy to create a permutation operator that acts on strings (included binary strings), as is done next

```
package Algorithm::Evolutionary::Op::Permutation; ## see 1.
use Carp;
use Algorithm::Evolutionary::Op::Base;
use Algorithm::Permute;
our @ISA = qw (Algorithm::Evolutionary::Op::Base);
our $APPLIESTO = 'Algorithm::Evolutionary::Individual::String';
our $ARITY = 1; ## see 1.
sub new { ## see 2.
  my $class = shift;
  my $rate = shift || 1;
  my        $self         =         Algorithm::Evolutionary::Op::Base::new(         'Algo-
  rithm::Evolutionary::Op::Permutation', $rate );
  return $self;
} ## see 2.
sub create { ## see 3.
  my $class = shift;
  my $rate = shift || 1;
  my $self = { rate => $rate };
  bless $self, $class;
  return $self; ## see 3.
} ## see 4.
sub apply ($;$) {
  my $self = shift;
  my $arg = shift || croak "No victim here!";
  my $victim = $arg->clone();
  croak "Incorrect type ".(ref $victim) if ! $self->check( $victim );
  my @arr = split("",$victim->{_str});
  my $p = new Algorithm::Permute( \@arr );
  $victim->{_str} = join( "",$p->next );
  return $victim; ## see 4.
}
```

1. This is the usual introduction to modules, which should be preceded with some POD documentation: description, synopsis, and so on. After declaration of the package name, we declare needed modules:`Algorithm::Permute` [http://search.cpan.org/author/EDPRATOMO/Algorithm-Permute-0.04/Permute.pm], a class for fast permutations, and base class for all operators, `Algorithm::Evolutionary::Op::Base`. Two constants should be defined also for the module: one of them is optional, the `$APPLIESTO` variable, which states to which individual class it might apply to; this will be used in the `apply` method, but if it applies to a whole hierarchy, for instance, all subclasses of `String`, it's better to find out a more sophisticated check; the second one, `$ARITY`, is used by other objects to find the number of arguments the `apply` method needs.

Tip

Do not reinvent the wheel: always look up CPAN when writing operators or individuals; you might find the right class for the job.

2. The `new` method does not do much this time, other than forward object creation to the base class (as all objects should do). An operator just has a *rate* as default variable; and this one has not got any other.

3. This is equivalent to new, and it's a fossil. Do not worry about it, it will probably be eliminated in other versions of the library

4. This is the most important method: is the one that actually does the job, and the one it is called to modify chromosomes. Chromosomes are passed by value, that is why it is cloned, and the result of the modification is returned; this is the way the higher-level classes, such as `Algorithm::Evolutionary::Op::Full`, expect them to behave that way, but they might do something different for particular algorithms (for instance, `Algorithm::Evolutionary::Op::QuadXOver` takes both arguments by reference, and is used within the canonical GA). This method creates a permutation object from the cromosome, and permutes it, assigns it back to the created chromosome, and returns it.

Other methods, such as `set` or `asXML`, can also be overriden from the base class, but just if you want to do something very specific; the base class versions will do just fine in most cases. Subclassing an Individual, or creating new kinds of data structures to evolve, is just as simple, and it is left as an exercise to the reader.

We will use this class to evolve DNA strings; just as we did before, but, first, our target string will only be composed of A, C, G and T, and, second, we will have no "distance to char", but overall distance among strings. The exercise is purely academic, but a similar problem is solved when sequence alignments want to be done, only in that cases there are several targets. We will use anothero module, `String::Approx` [http://search.cpan.org/author/JHI/String-Approx-3.19/] to compute approximate distances among strings `ea-(ex5.pl)`.

```
use Algorithm::Evolutionary::Op::Creator;
use Algorithm::Evolutionary::Op::Permutation;
use Algorithm::Evolutionary::Op::IncMutation;
use Algorithm::Evolutionary::Op::Crossover;
use Algorithm::Evolutionary::Op::CanonicalGA;
use String::Approx qw( adistr );
my $target =shift || 'CGATACGTTGCA';
my $maxGenerations = shift || 100;
my $popSize = shift || 100;
my $fitness = sub {
  my $indi = shift;
  return 1 - abs ( adistr( $indi->Chrom, $target ) );
};
my $incmutation = new Algorithm::Evolutionary::Op::IncMutation;
my $mutation = new Algorithm::Evolutionary::Op::Permutation;
my $crossover = new Algorithm::Evolutionary::Op::Crossover;
my  $ez  =  new  Algorithm::Evolutionary::Op::Easy  $fitness,  0.4,  [$mutation,  $crossover,
$incmutation ];
my $indiType = 'String';
my $hash = { length => length( $target ),
             chars => ['A','C','G','T']} ;
my $creator = new Algorithm::Evolutionary::Op::Creator( $popSize, 'String', $hash);
my @population = ();
$creator->apply( \@population );
my $gen;
do {
  $ez->apply (\@population );
  print "Best so far: ", $population[0]->asString(), "\n";
} until ( $population[0]->Chrom eq $target ) || ($gen++ > $maxGenerations) ;
print "Final\n", $population[0]->asString();
```

This program is very similar to previous examples. The only differences are that we use a different kind of chromosome, `Individual::String`, which uses any alphabet, and that we use several variation operators: `Op::IncMutation`, which increments a single element in the chromosome by one, taking into account the alphabet (that is, it would cycle A -> C -> G -> T); `Op::Permutation`, which we just declared. The fitness returns the distance between the string and the target string, taking into account the length difference, and the insertions and deletions needed to turn a string into the other. This is a problem, since AA and TA will have the same distance to GA, and there are many mutations which are *neutral*, leading to no change in fitness. Furthermore, strings such as GAAAAA are at a distance of 1 (or 1/divided by total length) from AAAAAG, but a very lucky permutation is needed to turn one into the other. This leads to the fact that, in this case, the evolutionary algorithm does not always find the solution.

Warning

Combinatorial optimization problems like this one are usually hard for evolutionary algorithms (and for any other search method, for that matter). It always help to have a good knowledge of the problem, and use any other methods available to us to improve search and make the *fitness landcape* less rugged.

Frequently asked questions

1. Evolutionary computation usually needs lots of CPU cycles. Wouldn't Perl make evolutionary algorithm programs slower?

   This question has two answers: the first one is no, and the second one is yes, so what?. Let is go to the first answer: it is always complicated to compare two different languages, even on a single algorithm, because it is virtually impossible to translate, sentence by sentence, from one lenguage to the next one; besides, even if you do, you have to take into account the quirks of the language, and what kind of things it does better: what kind of data structures, for instance. So, if we take all that into account, and we look at a particular language, well, we would have to run some tests to see which one runs faster. It is quite likely that a C program is faster than the equivalent Perl program (if translation time is significative with respect to total time), but I would say that, second for second, Perl is no slower than Java or C++ or Ruby. But, of course, I would be happy to hear the results of any benchmarks. The second answer is that performance is not, after all, so important: if your preferred tool is Perl, and you can code stuff blindfolded and single-handedly, you'd better do evolutionary algorithms with Perl than with, say, Fortran 9X, even if this language is able to extract the last drop of performance from your old processor. If you have to learn a new language, plus write an evolutionary algorithm in it, performance does not matter so much.

2. What other kind of cool stuff can you do with evolutionary computation and Perl?

   Besides the aforementioned GlotBot, there's something very similar, written using `Algorithm::Evolutionary` and `HTML::Mason`, available from http://geneura.ugr.es/Mason/EvolveWordsPPSN.html. The evolutionary algorithm was also combined with `SOAP::Lite` to carry out evolutionary algorithms with distributed population (code is available along with the first versions of OPEAL). As a degree project, some students of mine used OPEAL to optimize fantasy soccer teams, by optimizing step-by-step the team, or by optimizing the rules used to substitute players from one set of matches to the next. And, finally, using the same library, we optimized the assignment of papers to reviewers for the PPSN 2002 conference [http://ppsn2002.ugr.es]

References

- There are several books that deal with evolutionary computation thoroughly. The one that is more similar in approach to this tutorial is Genetic Algorithms+Data Structures = Evolution Programs [http://www.amazon.com/exec/obidos/ASIN/3540606769/perltutobyjjmere], by Zbigniew Michalewicz; already in its third edition, it has a practical approach from the beginning in its first part, and has a second part devoted to applications. Another interesting book is Introduction to Genetic Algorithms [http://www.amazon.com/exec/obidos/ASIN/0262631857/perltutobyjjmere], by Melanie Mitchell; although it devotes too much space to evolving cellular automata, it has a good balance between theory, practice and applications.

- The Hitchhiker's guide to evolutionary computation (HHGTEC) [http://www.cs.bham.ac.uk/Mirrors/ftp.de.uu.net/EC/clife/www/] is the field FAQ. Besides the FAQ, it includes lots of resources, links to other web pages, mailing lists, and home pages related to the subject.

- The main conferences in the field are GECCO (Genetic and Evolutionary Computation Conference) [http://www.isgec.org] and CEC (Congress on Evolutionary Computation) [http://www.wcci2002.org/cec/call.html], which take place anually, and are big events with lots of people and humongous proceedings, and PPSN, Parallel Problem Solving From Nature [http://ppsn2002.ugr.es], which takes place bianually (in even years) in Europe, usually in september; it is an smaller event, with around 150-200 attendees.

- EvoNet [http://http://evonet.dcs.napier.ac.uk/] is the european network for evolutionary computation, a consortium of european university departments and enterprises devoted to the promotion and application of evolutionary computation, in all its forms. Its web site contains all kind of things, from tutorials, to case studies, to lists of places where you can get degrees on evolutionary computation.

Developing GUI Applications with Gtk.pm

Red <red@criticalintegration.com>

The slides and details are still linked off the 2001 YAPC website or directly at http://www.criticalintegration.com/gtk-tutorial/

Gtk is a graphical toolkit for building X11 based graphical applications. This tutorial assumes Perl knowledge and is not for novices.

The first 20-30 mins or so of the tutorial is a background of Gtk, why, where and how. It emphasises the change in style from the more familier "straight line" programming to the seeming chaotic event-driven programming which is the fundimental of all Graphical programming.

The nasties of packing, boxes and getting used to the idea of giving up control.

A basic crib-sheet of basic widgets, functions and methods will be provided. This I found last year would be vital, as the process of re-affirming syntax via the slides is too time-consuming.

Then the fun starts.

A basic application requirement will be taken from the floor, and the application will be designed using pen and whiteboard. The structure will be explained, pitfalls will be purposefully fallen into to give the attendee a good basic understanding of how to write (and how not to write) basic X11 applications.

Last year, the application was the "Dutch Auction" program which was used at the end of YAPC::Europe.

Learning from the tutorial last year... I will be stricter on answering "pedant" questions like "Why don't you use a foreach loop" instead of assigning individual assignments. (Readability)

wxPerl -- Another GUI for Perl

Jouke Visser <jouke@pvoice.org>

Abstract

wxPerl -- Another GUI for Perl

Introduction

- Background of wxWindows and wxPerl
- Terminology
- using the wxWindows documentation for C++
- Status of wxPerl

First steps

- "Hello World"
- Buttons, labels and input
- Event handling
- Leftovers

More advanced controls

- Menus
- Common Dialogs
- Notebook tabs
- Device Context
- Printing
- Drag and Drop

Building an example application

Resources

http://www.wxwindows.org, http://wxperl.sourceforge.net, tutorials on http://www.perl.com

Advanced Testing

Michael G. Schwern `<schwern@pobox.com>`

Building on the basic techniques in `Test::Tutorial`, we'll learn about how to write tests for the more difficult situations you come up against in day-to-day programming.

- Testing services and daemons
- Testing network applications
- Testing web applications with `WWW::Automate`
- Testing GUI applications
- Testing databases
- Testing failure
- Test coverage analysis with `Devel::Cover`
- Common cross-platform testing pitfalls

Practical Web Programming

Tom Phoenix

Editor's note.

There is no paper nor abstract for this tutorial. Don't despair, you can find a wealth of examples and explanations in Randal's legendary WebTechniques Perl columns [http://www.stonehenge.com/merlyn/WebTechniques/].

POE for Beginners

Jos Boumans, http://japh.nu amsterdam.pm `<kane@cpan.org>`

About POE

The following bit is stolen directly off http://poe.perl.org (all good things stem from there):

POE parcels out execution time among one or more tasks, called sessions. Sessions multitask through cooperation (at least until Perl's threads become mainstream). That is, each session returns execution to POE as quickly as possible so it can parcel out time to the next.

POE includes a high-level component architecture. Components are modular, reusable pieces of programs. They perform common, often tedious tasks so programmers can work on more interesting things.

Several POE components have been written to handle client, server, and peer networking tasks. Because of them, POE has become a convenient and quick way to write multitasking network applications.

POE supports graphical toolkits such as Tk and Gtk. It is the only Perl programming framework that does this. It also supports Curses, HTTP, and other user interfaces. POE programs can present multiple user interfaces at once.

About the Contents

Most of the material I will be using will be based on the CPAN-module `Acme::POE::Knee`, which is a simple module I wrote a while ago to demonstrate how the POE kernel dispatches events. It shows the behaviour of the 'state machine' illustrated as a pony race (or POEny, as we like to call it). The second main source of material will come from the likes of the POE article, which appeared on perl.com a while ago. The actual article can be found here: http://www.perl.com/pub/a/2001/01/poe.html. The article describes how to build a server daemon that understands TCP using POE.

About the talk

This tutorial is aimed at giving a basic understanding the workings of POE, trying to get the audience to understand how to use POE and to show them some simple programs (or POEgrams). The main obstacle in understanding POE is right at the beginning; it introduces many terms, concepts and technology that might take some time to wrap your ahead around. This tutorials aims to be your guiding light in this process. Once these basics are covered, we can go into the appicatins of POE and teach the audience how to write POEgrams themselves.

E-Mail Processing with Perl

Mark A. C. J. Overmeer, MARKOV Solutions
<mark@overmeer.net>

`Mail::Box` is currently maintained and developed under a generous grant provided by the Dutch NLnet Foundation [http://www.nlnet.nl/]

Abstract

In the last few years, e-mail developed from an application used by a relatively small scientific community into a basic means of communication for everyone. Companies have added e-mail as a new way to keep in touch with costumers, next to the traditional postal and telephone services. Friends post birthday invitations by e-mail, and the pictures of the last family meeting are broadcasted to everyone on them.

Creative new practices for e-mail include spam and spreading viruses. Jokes are not posted in plain text anymore, but preferably as PowerPoint presentations. Plain text is not nice at all: let's send HTML or -even better- full Word and PDF documents to everyone we know!

The concept of e-mail has changed, and with it the need to control that flow has increased. Filters which automatically remove received spam, virus checkers, and message content converters are more and more implemented. At the same time, mail folders grow rapidly in size. The number of messages to read daily explodes. How to cope with that?

The recently developed `Mail::Box` module is capable of handling various kinds of mail folders and replaces many older Perl modules (like MailTools) with components which are better suited for the current generation of messages. The module supports many advanced features which can help implementing the filters you need.

Some features of `Mail::Box`, `Mail::Address`, and `MIME::Types` will be discussed. A few small programs will be explained. Although Mail::Box is fully Object Oriented, only little OO is really needed to understand the examples.

Introduction

Perl is a powerful language for automating administrative processes; whether it be for system administration, customer interaction via web-pages, or reporting purposes. Over 3800 modules which can help you with your tasks are available on Perl’s Open Source archive CPAN[1]. At the moment of this writing, August 2002, about 75 of these modules are directly related to handling e-mail. This paper may help you use some of them, especially focussed on those related to the `Mail::Box` module.

Competing Implementations

Reading through the list of e-mail related modules, you get the impression of overlap: many modules implement the same functionality. For instance, there are at least five implementations for parsing a message from file into structures which can be handled by Perl. Why is there so much overlap?

There are various reasons why overlapping implementations are available in the Perl libraries – and those reasons apply to many other Open Source projects as well. They are caused by the following reasoning of programmers:

1. “The implementation is *bad*”

   Quite a few modules on CPAN are very old, and really should be replaced by modules which cover the same functionality but use improved syntax[2]. In searching for a useful module, the programmer gets a *bad* feeling about the existing code, and produces a new implementation for the same thing.

   However, this is easily said, but harder to achieve. A replacement module passes through many phases:

CPAN’s main archive can be found at http://www.cpan.org/

- not at all functional
- partly functional, still many bugs
- partly functional – alpha
- partly functional – beta
- more functionality, fewer bugs
- acceptable functionality – release
- more functional than the original module

  Although it is often easy to see *that* a module must be improved (users with little experience can see that), it requires more experience to determine *how* a module can be improved. It usually takes a lot of effort to get to the state that a rewritten module actually is *improving* the situation by provide better code.

  The conclusion must be that most modules look worse than they actually are: they are working with acceptable functionality. The qualification *bad* is only acceptable if you provide a *better* implementation.

2. “Oh, I didn’t know there was an implementation already”

   CPAN has grown too large. It is very hard to find out which modules can satisfy your needs: On which versions of Perl does it run? On which platforms does it run? How functional is the module?

   If you start coding because nothing is available on CPAN yet, after a few months of work you may see someone else publishing his implementation addressing the same problem. So: publish early and get people involved that way. However, in early stages you do not supply suffcient functionality, which drives people immediately away from your code never to come back. There is no easy answer to this dilemma.

3. “It is too complicated to use”

   The subjective feeling of *complicated* is caused by a few factors. Most complexity with modern programming languages is not in learning the language itself, but rather in learning how to use its libraries. Studying the libraries will consume most time. Some of Perl’s libraries (for instance Perl/Tk) are much larger than Perl itself!

   Large libraries easily get marked as *too complicated*, just because the user underestimates the complications involved in a correct implementation. The average user is not aware of requirements in RFCs[3], compatibility issues with broken applications, environments of other users of the same library, or even of what the real needs of his own program are.

   Of course, the author of a module can help fight the feeling of complication by providing a simple interface to the library, good documentation, examples, and support. But the module will still be large and complex.

It is obvious that modules are in various conditions, created with different objectives in mind, for different applications, and written by different programmers. In general, the e-mail related modules for Perl are too small, contain little to no documentation, and were created by many independent programmers all using their own programming style. Still people need to use them, and often they succeed. But how much effort do they have to put in it, and what is the quality of the result?

Message Parsing

Let us focus on the subject of this paper: parsing e-mail messages with Perl. At first glance, e-mail message parsing looks really simple. Current implementations on CPAN are much too complicated! Well... Yes: simple messages are simple to parse. A simple message:

```
    From: me@example.com
    To: you@somewhere.aq
    Subject: This is a demo

    This is the message body.
    Bye!
```

Perl’s syntax is under continuous development, which makes the language more powerful with every new version. Most e-mail related core modules pre-date the time when real references and Object Orientation where added to the language, hence are full of tricks to circumvent these deficiencies.

RFC=Request For Comment, the way protocols are standardized on Internet. See http://www.ietf.org/rfc.html for a full list.

A simple parser for that:

```
    my %head;
    while(<>)
    { last if $_ eq ”\n”;
        my ($name, $content) = split /\:/, $_, 2;
        $head{$name} = $content;
    }
    my @body = <>;
```

When you look at Mail::Internet – the *mother* of all mail related modules – you see that it actually is implemented this way. Really straight forward. Back in 1995 when Mail::Internet was created, e-mail message were this simple. But e-mails are certainly not that simple anymore! My cousin posts the latest pictures of her children from New Zealand by e-mail. Students post to each other PowerPoint presentations to decorate jokes with animations and sound. In recent years, e-mail changed from small single-part text-based messages into often large, multi-parted multi-media documents. The simplicity of Mail::Internet (with multi-part messages not even supported) is certainly not suffcient anymore.

Of course extensions – like the MIME::Entity family – do exist, but each solve only a few of the complications which are associated with more complex messages. Using these packages, you also need to know too much about details of the e-mail protocols. Some of the complications which e-mail related modules have to deal with:

- header lines can appear more than once in the same message header. They should be treated case-preserving. Order preserving is nice. Wrapping long lines (folding) is only permitted for some of them;
- multi-parts and nested multi-part messages are quite hard to handle. Start thinking about the complications involved in constructing them. Each message part looks a bit like a message itself, but they are slightly different composed. What about deleting one part? Does it imply that the main message’s identity is changed?
- the message body can be encoded (base64, quoted-printable, ...) for transfer, even when it contains plain text. The text can use varying character sets. What if we want to automate a reply included in a piece of our text encoded as ISO 8859-15, and the message we reply to is in UTF-8? Or it is structured in HTML?
- various folder types, some organizing the messages together in one file (like Mbox folders), others as separate files within one directory (like MH and Maildir), or separate entities on a remote machine (like IMAP and POP);
- differentiated treatment of labels which relate to messages status (is it read, replied upon, etc). Sometimes these labels are stored in the messages header as `Status` and `X-Status` lines. However, there are differences in implementation and interpretation of these header lines. Maildir encodes the labels in the filename of the message, while MH maintains a special label file per folder.
- how do we create a reply to a binary message? Or for a multi-part? Or one part within a multi-part?

Maybe this list will warn you not to create your own implementation for handling mail folders. A decent mail handling module must hide as many of the complications as possible from the users. Of course, the module implementing everything that has to do with e-mail gets large and hence harder to use and harder to learn. However, hopefully it enables everyone to create powerful applications without the need to know all the details about correct mail.

Development of `Mail::Box`

During the year 2000, I needed a module to process mail folders and discovered that the `Mail::Folder` module was not supported anymore. `Mail::Folder` is (*was*) able to manage a set of `MIME::Entity` messages in various kinds of message folders. In need of a functioning folder handler, `Mail::Box` version 1 was developed. `Mail::Box` version 1 was based on the existing `MIME::Entity` messages. The further I went with my module, the more problems I encountered with these existing modules. Most problem reports I received were the result of these old modules I used.

Then in mid-2001, I decided that all e-mail related modules were due for a big clean-up. A rewrite from the bottom, fully object oriented, intensively documented, and uniformly structured was the way to go. The result is `Mail::Box` version 2. The goal of `Mail::Box` is to be

- one consistent set of packages;
- documented on different levels of abstraction;
- easily extendible in an object oriented way;
- hiding as much nastiness about messages as possible; and
- providing strong building blocks to construct new messages, and replies, forwards, and bounces of messages.

In the next chapters, we will see a few things that `Mail::Box` can do for you. Where applicable, some other e-mail related modules will be introduced as well.

`Mail::Address`

`Mail::Address` is part of the MailTools bundle of modules. Most of the modules in that bundle are best avoided, but `Mail::Address` is a relatevely positive exception.

Use `Mail::Address` to convert text into e-mail address containing objects, or the reverse: convert objects into text. The `Mail::Address` object provides access to the knowledge in the address information.

Parsing a line with addresses

The following program demonstrates how `Mail::Address` is used to get e-mail addresses from a string:

```
use Mail::Address;
my $line = ’info@example.com (Overmeer, Mr M.)’;
my @addr = Mail::Address->parse($line);
my $first = $addr[0];
print $first->comment,”\n”; # (Overmeer, Mr M.)
print $first->name,”\n”; # Mr M. Overmeer
print $first->address,”\n”; # info@example.com
```

Different textual representations of e-mail addresses are handled. The parse method returns all addresses which are contained on the line as list. Each address is converted into a separate `Mail::Address` object. The name method is most impressive, knowing how different cultures prefer to mangle personal names and then de-mangle them.

Creating an address

The following example shows the reverse process: create an address object yourself, and then format it into a correct string.

```
use Mail::Address;
my $a = Mail::Address->new(’Mark Overmeer’,’me@example.com’);
print $a->format,”\n”; # Mark Overmeer <me@example.com>
```

`MIME::Types`

The `MIME::Types` module knows how to handle mime-types right. A MIME compliant[4] message contains information about the kind of data which is stored in the body. That information is stored in the Content-Type line. For example:

```
Content-Type: text/plain; charset=”us-ascii”
```

In this example, the mime-type of the message is text/plain. Information about the character set used to compose the text is not part of the mime-type.

Information about a mime-type

Given the string describing a mime-type, the `MIME::Types` module can provide details about it.

The MIME (Multipurpose Internet Mail Extensions) standard is described in RFCs 2045 up to 2049, and defines how the header lines are shaped. The RFCs also define acceptable content for specific lines.

```
   use MIME::Types;
   my $alltypes = MIME::Types->new;
   my $plain = $alltypes->type(’text/plain’);
   print $plain->mediaType; # text
   print $plain->subType; # plain
   if($plain->isBinary) ... # false
   if($plain->isAscii) ... # true
   print $plain->encoding; # quoted-printable
```

The module handles the case-insensitivity of the mime-type description, and provides a little more background information. As a service, it can tell you whether the body is binary, and how the data should preferably be encoded for transmission.

Information about filename extensions

The `MIME::Types` module also knows about filename extensions in relation to mime-types. Some of these notions are operating system specific.

```
   use MIME::Types;
   my $alltypes = MIME::Types->new;
   my $h = $alltypes->mimeTypeOf(”index.html”);
   print $h->simplified; # text/html
   print ”$h”; # text/html
   my $g = $alltypes->mimeTypeOf(”GIF”);
   print $g->type; # image/gif
   my $z = $alltypes->mimeTypeOf(”a.gz”);
   print $z->type; # application/x-gzip
   print $z->simplified; # application/gzip
   print $z->subType; # gzip
```

In mime-types, a leading `’x-’` indicates the use of a name which is not offcially registered. So, new names will be introduced with a `x-`, and after registration this flag will disappear. The `MIME::Types` module understands this.

`Mail::Message`

This is the first object from the `Mail::Box` module we will discuss. `Mail::Box` is all about messages which are stored in folders, but messages can also live (temporarily) outside a folder and therefore can be discussed separately first. Every MIME compliant message contains

- a header: a few lines which describe the message’s body, information about the transport of the message, and so on. This is also called the meta information about the message; and
- a body: the actual content.

In `Mail::Message`, these parts are represented by two separate objects: a `Mail::Message::Head` and a `Mail::Message::Body`. Let’s take them apart.

A message header

The header of a message contains information about the message, excluding the *payload* which is stored in the body. This information is stored in well-described lines, for example:

```
   From: you@example.com
   Date: Wed, 4 Oct 2000 14:22:35 -0400 (EDT)
   Message-Id: <200010041822.e94IMZr19712@example.com>
   To: me@home.museum
   Received: (from majordomo@localhost)
       by example.com (8.11.0.Beta3/8.9.3) id e94IMj420302
       for everyone; Wed, 4 Oct 2000 14:22:45 -0400
   Subject: Font metrics
   Status: RO
   Content-Type: text/plain; charset=”us-ascii”
   Content-Length: 1620
   Lines: 32
```

The label (*field*) of a header line is case-insensitive, but has a preferred notation: each word in the field should start with a single capital.

Some of the lines – like `Received` in the example – may be *folded* over multiple lines when they get too long to be printed nicely on a screen. This is only allowed for the well-described *structured* lines, not for other lines. Each (folded) line can have three parts:

```
            <field> : <body> ; <comment>
```

The *comment* including its leading semi-colon is optional. Although it doesn't sound that way, the comment part may contain very useful information as well: somethimes it contains *attributes*.

Get a header line

Once you have a message, you can ask about lines in the header:

```
   my $msg = ....; # get it somewhere
   my $date = $msg->head->get('date');
   print "$date"; # Wed, 4 Oct 2000 14:22:35 -0400 (EDT)
   $date->name; # Date
   $date->print; # Date: Wed, 4 Oct 2000 14:2...
```

The `$msg` is constructed some way, or contained in a folder; we'll see later how to get one. The `$date` returned is either undef (header field not found) or a `Mail::Message::Field` object. This object stringifies to the body of the header line, excluding the comment. To simplify access to the header lines, you can also directly ask the *message* for information in the header. Consider the following alternatives:

```
   my $msg = ....; # get it somewhere
   my $head = $msg->head;
   print $head->get('DATE');
   print $head->get('content-type');
   print $msg->get('DATE');
   print $msg->get('content-type');
```

It doesn't really matter which solution you take. The former will be a tiny bit faster though.

Special fields

Of course the previous section was valid, however it is much too low level. For instance, the `To` and `From` header lines contain e-mail addresses. And did you know that those lines are overruled by `Resent-To` respectively `Resent-From` lines if present?

A few fields get special treatment. Fields which start with `Content-`, like `Content-Type` and `Content-Length` are managed by the message's body (explained in the section called "Information about the body"). Fields which contain e-mail addresses have their own methods which return `Mail::Address` objects. Example:

```
   my $from = $message->from;
   print $from->name; # Mr M. Overmeer
   my @to = $message->to;
   my @cc = $message->cc;
```

When there are `Resent-` lines present, only the data in those lines is returned, as it should be. Otherwise, the usual set of lines is parsed.

Message body

The body is the `payload` of the message. `Mail::Box` stores the body of each message in opened folders temporarily in an efficient way. As user of the module, you do not to know which way: it may be kept in memory or in a file. Independent of the way it is stored, the data is available in various ways. When the message is created or read, some header lines are copied into the body object. This is especially important because we will process the body, which may change the `Content-Length` of the body, or the number of `Lines`. The changes must be reflected in the meta information about the body. Some processing steps may even modify the `Content-Type`. Often you want to decode the body to be able to use it, which means that the `Content-Transfer-Encoding` changes to none.

Getting the body

A message can supply two different versions of its body: the encoded body, and a decoded version of the body. Of course, in an application it is often more useful to ask for the decoded version:

```
    my $msg = ...; # get it somewhere
    my $encoded = $msg->body; # are you sure?
    $encoded->print; # oops, base64 encoded text?
    my $decoded = $msg->decoded;
    $decoded->print; # perfect
```

Do not worry: if the body was not encoded in the first place, then no time or resources are consumed to “decode” it. However, if there is work to be done, the `$decoded` points to a different body object, which no longer has anything to do with the message from which it came.

Information about the body

Information about the body which was originally found in the header is copied into the body object. When you have a decoded body of a message, this data may (will) differ from what was found in the header.

The following examples may be useful:

```
    print $decoded->transferEncoding;
                 # was Content-Transfer-Encoding
    print $decoded->nrLines; # was Lines
    print $decoded->type->body; # text/html
    my $type = $decoded->mimeType; # a MIME::Type!
    print $type->subType; # html
    $decoded->print # works best
        unless $decoded->mimeType->isBinary;
    $decoded->print # even simpler with
        unless $decoded->isBinary; # this shortcut
```

Constructing a message

There are a few ways to construct a new message. Think about whether the new message has a relation to an existing message or not. If so, what is that relationship? Although the methods are quite similar in use, their results can differ quite a lot.

The following message constructors are available:

`Mail::Message->read(\*STDIN)` and `Mail::Message->read(\@lines)`
: Let’s start with the worst solutions. These methods are required only when an external source delivers the message data this way. For instance, when you write a **procmail**[5] application, **sendmail** will start your program and provides the message on STDIN. You are left without options, so need to `read`.

`Mail::Message->build(data)`
: Create a message based on a set of loose data: throw in a number of header lines, and some scalars, arrays, or file-handles with text, and the message is created.

`Mail::Message->buildFromBody($body)`
: A more careful approach to construct a message: first construct a body object, and then turn it into a full message with a header. Of course, the content related header lines are constructed based on the data which is stored in the body.

`$msg->reply`
: Create a reply based on a received message. This implies that the destination of the newly created message is the originator of the source message. Some header lines keep track of this game of answering answers, and they are updated accordingly.

`$msg->forward`
: Forwarding a message implies that you want someone else to read the message as well, but at the same time wish to add some notes to the message. The resulting message will contain (a part of) the original. Some

procmail is used to process incoming mail at the moment it arrives on the system. It can be used to remove spam at arrival, or channel messages to separate folders based on sender.

header lines are automatically created as well.

`$msg->bounce`
: Use this when you need to forward a message to someone without modifying the content. Some minor modifications in the message header are made (specifically some `Resent-` headers are added), but that's all.

The next sections discuss small examples for each constructor. Look at the manual-page of `Mail::Message::Construct` for full details. The examples are taken from the `examples/` directory which is distributed in the `Mail::Box` module.

Constructing with read

To start off simple: read parses a message from a file handle, a string, or an array of lines. Be warned that all lines must end with a newline, including the last line.

General examples:

```
   use Mail::Message;
   my $msg1 = Mail::Message->read(\*STDIN);
   my $msg2 = Mail::Message->read(\@lines, log => 'PROGRESS');
   my $msg3 = Mail::Message->read(<<MSG);
   Subject: hello world
   To: you@example.com
                       # warning: empty line required !!!
   Hi, greetings!
   MSG
```

When you write your program in a way such that lines are first collected in an array, and then this `read` method is called to produce a `Mail::Message`, you are *not* on the right track. In such cases it is better to use two steps to produce the best result:

```
   use Mail::Message;
   my @lines = ... ; # contains 8bit encoded html
   my $body = Mail::Message::Body->new
     ( data => \@lines
     , mime_type => 'text/html'
     , transfer_encoding => '8bit'
     );
   my $msg = Mail::Message->buildFromBody($body);
```

Have a look at the `Mail::Message::Body::Construct` manual page to learn more.

Constructing a message in one step

Building a whole message from components is more flexible than using read: specify data and some header lines, and get a fully dressed-up message in return.

A number of missing lines will be added for you; the `Content-` lines are calculated or predicted. When more than one source for data is specified, a multi-part message is produced. When the data needs to be encoded, it will automatically get encoded.

If one `Mail::Address` or an array of them are specified for a header line, they are flattened to a nicely formatted string. A full program:

```
   use Mail::Message;
   use Mail::Address;
   my $to = Mail::Address->new('your name', 'you@tux.aq');
   my $signature = Mail::Message::Body->new(...);
   my $msg = Mail::Message->build
    ( From => 'me@home.nl'
    , To => $to
    , Cc => 'everyone@example.com'
    , data => [ "This is\n", "the first part of\n"
              , "the message\n" ]
    , file => 'myself.gif'
    , file => 'you.jpg'
    , attach => $signature
    );
```

The example above explicitly provides three header lines. Next to those, at least the obligatory `Message-ID` will be added, plus many lines about the type and size of the body. The resulting message is a multipart containing four parts: the first specified in-line, then two as external files, and the last as a prepared body object.

Constructing a message from a body

The `buildFromBody` method is related to build, but is used with a previously composed body object. This is better suited for creating (nested) multi-part messages, where more control on the produced message structure is required. The overly complicated example below shows that header lines may also be passed as `Mail::Message::Fields`. In this case, you overrule the content-type as stored in the body: don’t do that at home!

```
   my $body = $some_message->decoded; # or
   my $body = Mail::Message::Body->new(...);
   my $type = Mail::Message::Field->new
    ( ’Content-Type’, ’text/html’
    , ’charset=”us-ascii”’
    );
   my @to =
    ( Mail::Address->new(’Your name’, ’you@example.com’)
    , ’you@example.info’
    );
   my $msg = Mail::Message->buildFromBody
    ( $body
    , From => ’me@example.nl’
    , To => \@to
    , $type
    );
```

Constructing a reply

A reply is based on a source message, and by default constructs a message ready to be sent back to the author of the source message.

There are many ways to get that to work. The complicating factor is the wish to utilize parts of the source message in the reply. This is especially hard when you want to created automated replies: think about the receipt of binary or multi-parts which have to be treated differently from simple plain text messages.

An example:

```
   use Mail::Message;
   my $msg = ...; # get it somewhere
   my $pgp_key = Mail::Message::Body...
   my $reply = $msg->reply
    ( prelude => ”No spam, please!\n\n”
    , postlude => ”\nGreetings\n”
    , strip_signature => 1
    , signature => $pgp_key
    , group_reply => 1
    );
```

Please do not forget to terminate all lines in the prelude and postlude with a newline.

With the right options, `Mail::Box` tries to do its best for you, in any situation. When a reply is made on a multi-part message, it tries to figure out which part is the primary content to be used in the answer composition. When the source message’s body is binary, it will be attached, and the prelude plus postlude are combined into a nice covering text.

Constructing a forward

Once you have seen reply you have seen forward: its interface is rather the same. The forwarded message also composes a new message based on the original, but it is encapsulated differently: the whole message body is taken to be forwarded by default, not only an extract.

Also some of the header lines will be updated differently. For instance, the subject line will start with “`Re:`” for the reply, and with “`Forw:`” for the forward. Besides, while a reply knows where the message must be sent, the forward requires that information to be specified.

Constructing a bounce

Too many automated applications forward or reply their messages when they really should use bounce. This may be caused by the fact that bouncing messages was a latter addition to the MIME specifications.

Examples for bounce can only be small:

```
   use Mail::Message;
   my $msg = ...;
   my $bounce = $msg->bounce
    ( To => ’postmaster@home’
    , Bcc => \@the_whole_world
    ):
   $bounce->send;
```

This example also contains a new feature: by calling send on a message it will be... sent. The `Mail::Box` module contains packages which support various message transmission mechanisms. The easy to use `send` method tries hard to find a way which works in your environment. Of course, this is configurable as well.

Mail::Box

So far, only single messages have been discussed. Single messages are created in your program, and disappear when your program stops. To preserve messages they must be stored in folders. The `Mail::Folder` name-space is occupied by an implementation which is not maintained, so the implementation which can handle `Mail::Message` objects was named `Mail::Box`. There are many kinds of mail folders, which all differ quite substantially in how they store a message. They all employ different run-time administration of the mes-sages they contain. But in the general case you do not have to know these distictions if you are using the `Mail::Box` module.

The Mail::Box::Manager

The `Mail::Box`::Manager keeps track on all open folders, so you cannot accidentally open the same folder twice in your program. It also autodetects the folder type, which saves you work, and at the same time compiles the required modules for you. Your conclussion must be: always start with the manager. In the next example, we open a folder via the manager:

```
   use Mail::Box::Manager;
   my $mgr = Mail::Box::Manager->new;
   my $inbox = $mgr->open(’=Inbox’):
   die unless defined $inbox;
   $inbox->close;
```

As you see, no need to specify whether this is an Mbox, MH, or Maildir type of folder. When the name of the folder starts with an equals-sign (=), it will look for the name in a – folder type dependent – default directory.

Important to know: folders are opened read-only by default. Use

```
   my $inbox = $mgr->open(’=Inbox’, access => ’rw’);
```

to allow updates. When the folder is closed, it is automatically de-registered from the manager. When the folder is closed explicitly (calling method `close`), goes out of scope[6], or the program terminates, the changes will be written.

Getting the messages

Of course, the only reason to open folders is to get to the messages which are located in them. The main meth-ods for this are

```
    my $msg = $folder->message(3); # fourth
    my $msg = $folder->message(-1); # last
    my $msg = $folder->messageId($msgid); # that one
    foreach my $msg ($folder->messages) # all subjects
    { print $msg->subject, ”\n”;
    }
    print $_->subject, ”\n”
        foreach $folder->messages; # all subjects
    print scalar $folder->messages; # count
    my $msg = $folder->message(3);
    $msg->delete; # flag for deletion
    $folder->message(3)->delete; # same
```

There are many more methods available, which are described in the manual pages. The deletion of a message

A my variable only has limited visibility, the scope. When the result of opening a folder is caught in such variable, it will get destroyed with the variable at the end of its scope.

takes place when the folder is closed, not immediately.

Moving messages around

Next to retreiving messages from a folder, you are also able to add messages to a folder. Be warned that some of these methods work on the manager, others on a folder.

```
    $mgr->moveMessage($otherfolder, $msg);
    $mgr->copyMessage($otherfolder, $msg);
    $mgr->appendMessage($otherfolder, $msg);
    $folder->addMessage($msg); # a Mail::Message
```

It is permitted to move, copy, or append more than one message at once. When the specified folder is open in the program, actions will take place on that opened folder. Otherwise, the updates are made as cheaply as possible, preferably without opening the destination folder.

When you move a message between folders of a different kind, `Mail::Box` will coerce them: adapt them to the requirements of the folder. All methods shown above return the coerced version of the supplied message.

It is also permitted to use `Mail::Internet` and `MimeEntity` messages in most of this kind of operations. This means that you can intergrate `Mail::Box`in your existing applications.

Playing with folders

As final example of the power of `Mail::Box` a small example which plays with folders:

```
    use Mail::Box::Manager;
    my $mgr = Mail::Box::Manager->new;
    my $pop = $mgr->open('pop3:user:passwd@mail.isp.net')
       or die "Failed to open POP connection: $!\n";
    my $loc = $mgr->open( 'InBox', access => 'rw'
                        , type => 'mh', create => 1)
       or die "Cannot open inbox: $!\n";
    $pop->copyTo($loc, delete_copied => 1, select => '!spam');
    $mgr->closeAllFolders;
```

In the example above, messages are moved from a remote host using POP3 to a local folder except when they are flagged to contain spam (unsolicited mail).

Further reading

This paper only briefly introduced a few of the modules which can be used to handle e-mail with Perl. It is only an introductory story, and you certainly need to read more to write real-life applications. To learn more about `Mail::Box`, you may need additional information from

- the `Mail::Box` website at http://perl.overmeer.net/mailbox, which lists more resources and tutorials;
- the mailing-list `<mailbox@perl.overmeer.net>`, to get your questions answered;
- a browsable (HTML) version of the documentation at http://perl.overmeer.net/mailbox/html/, which is also as a whole downloadable from the website;
- the `scripts/` and `examples/` directories included in the distribution.

Powerful libraries require study before they can be used, and `Mail::Box` is no exception to this rule. Sometimes they look too complex, but often that's because people underestimate the complications involved. Be glad for everything that is offered for free. Contribute with complaints and suggestions, so existing modules can improve. New implementations can only declare others 'bad' if they themselves are in a 'better' state.

As final remark: be very, very, very careful not to lose e-mail by mistakes in your program. People will hate you when e-mail gets lost.

Part II. Inside Perl

Tagmemics

An Introduction to Linguistics for Perl Developers

Allison Randal <al@shadowed.net>

This talk explores the influence, the theory of tagmemics has had on the development of Perl.

Perl 5.10

Hugo van der Sanden <hv@crypt.org>

What can we expect from Perl 5.10?

Perl 6 Prospective

Damian Conway <damian@conway.org>

Larry Wall <larry@wall.org>

This talk looks at what is known, surmised, guessed, wished for, and dreaded about Perl 6. It discusses the history, motivations, syntax, semantics, and likely idioms of the new Perl.

On Topic and Topicalizers in Perl 6

Allison Randal <al@shadowed.net>

This talk was well received at the YAPC::NA. It describes the new behaviour of $_ in Perl 6.

Inside Parrot

Dan Sugalski <dan@sidhe.org>

Parrot is a virtual machine used to efficiently execute bytecode for interpreted languages. This talk is a gentle introduction to Parrot. It covers Parrot basics and leads into programming Parrot Assembler.

Perl 5.8 At Last

Arthur Bergman <arthur@contiller.se>

Abstract

For the past two years Perl 5.8 has been one of the worst hidden skunkworks ever. Conceived approximately at the same time as the much publicized Perl 6 project, 5.8 (clandestinely disguised as 5.7) has made major enhancements to the Unicode functionality, sprouted a new user-visible threads implementation, silently assimilated several dozen modules, and created some completely new ones.

Come to hear what's new, what's changed, and what's gone. Be aware that most of this information is blatantly stolen from `perldelta` and the brain of Jarkko Hietaniemi.

Unicode

Perl 5.8 finally properly supports Unicode. It should now be much more usable than it was in 5.6.0 and 5.6.1. In fact, quoting the most excellent `perluniintro.pod`, “5.8 is the first recommended release for serious Unicode work”. The Unicode Character Database coming with Perl has been upgraded to Unicode 3.2.0, whereas 5.6.1 had 3.0.1. Most UCD files are included with some omissions due to space considerations.

The Perl Unicode model is straightforward: strings can be eight-bit native bytes or strings of Unicode characters. The principle is that Perl tries to keep its data as eight-bit bytes for as long as is possible. When Unicodeness can't be avoided, the data is transparently upgraded to Unicode. Native eight-bit bytes are whatever the platform uses (for example Latin-1); Unicode is typically UTF8.

Perl will now automatically do the right thing with regard to unicode and non-unicode strings. All functions and operators will respect the `utf8` flag. For example, it is now possible to use Unicode strings in hashes and correctly use them in regular expressions and transliterations. This has fully changed from 5.6 where you controlled Unicode support with the lexically scoped `utf8` pragma. To fully use Unicode in Perl we now have to have compiled Perl with perlio, the new IO system written by Nick Ing-Simmons, together with the new `Encode` module written by Dan Kogai, which allows different filehandles to be set to bytes, unicode or legacy encodings. `Encode` also comes with 'piconv' which is a Perl implementation of 'iconv' and 'enc2xs'. This allows you to create your own encodings to `Encode`, either from Unicode Character Mapping files or from Tcl Encoding Files. From SADAHIRO Tomoyuk comes `Unicode::Normalize` and `Unicode::Collate`, used, unsurprisingly, for normalization and collating.

Perl threads

Just like 5.8 is the first recommended release for Unicode work, it is also the first recommended release for threading work. Starting with 5.6, Perl had two modes of threading, one style called '5005threading', mainly because it was introduced with 5.005, and 'ithreads', which is short for interpreter threads. Gurusamy Sarathy introduced 'ithreads' as a step forward from the multiplicity needed to support the pseudofork implementation on Win32. However in 5.6 there was no way in controlling these threads from Perl, which has changed with the introduction of two new modules in 5.8.

The basic rule for this thread model is that all data is cloned when a new thread is created. So by default no data is shared between threads. If one wants to share data, then there is a `threads::shared` module and the new `shared` variable attribute. Controlling the new threads is done using the `threads` module. More reading can be found in the respective modules and in `perlthrtut`.

New IO

Perl can now rely on our bugs instead of on the IO implementations bugs. In Perl 5.8 we are now using PerlIO which replaces both 'stdio' and 'sfio'. The new IO system allows filters to be pushed/poped from a filehandle for doing all kinds of nifty things: the `Encode` module, mentioned earlier in the Unicode discussion, uses PerlIO to do the magic character set conversions at the IO level.

Interested parties who want to create their own layers should look at `perlapio`, `perliol`, `PerlIO` and `PerlIO::via`.

Safe signals

There will be no more random segfaults caused by signals, as we now have a signal handler that just raises a flag and then dispatches the signal between opcodes, so you are free to do anything you feel like in a signal handler (since it isn't run at async time, it isn't really a signal handler). This has potential for conflicts if you are embedding Perl and rely on signals to do some specific behaviors. If you like having a chance of a random segfault on signals you can always compile Perl with PERL_OLD_SIGNALS , but this will not be threadsafe.

New and improved modules

Perl 5.8 comes with 54 new modules, many from CPAN, as one goal has been to make it easy for `CPAN` to work out of the box. As a result, libnet and a couple of other modules have been included. We have put a lot of work into testing, so `Test::More` and that family of modules were naturally included. There was also a push to make Perl more i18n friendly, so it includes several i18n and l10n modules as well as the previously covered Unicode modules. There are a bunch of modules that provide access to internal functions like the `PerlIO` modules, `threads` module and `sort`, the new module that provides a interface to the sort implementation you are using. Finally we also thought it was time to include `Storable` in the core.

We also have included a number of updated modules. `Cwd` is now implemented in XS, which gives us a nice speed boost. `B::Deparse` has been improved to the point it is actually useful. Maintenance work on `ExtUtils::Makemaker` has made it more stable. `Storable` now supports unicode hash keys and restricted hashes. `Math::BigInt` and `Math::BigFloat` have been upgraded and bugfixed quite a lot, and they have been completed by the inclusion of `Math::BigRat` and the `bigrat` and `bignum` functions for lexical control of transparent bignumber support.

Speed improvements

Even though this release includes a lot of new features, there are some optimizations included as well! We have changed 'sort' to use 'mergesort', which came as a surprise for me since I have been told since I was a toddler to use 'quicksort'. However the old behavoiur can be controlled using the `sort` module and we even have a mystery stable 'quicksort'!

Once again we have changed the hashing algorithm to something called One-At-A-Time, so all of you who depend on the order of hashes have a good reason to fix your programs now!

'map' and 'unshift' have also been made faster.

Testing

We hope this should be the most stable release of Perl to date, as an extensive QA effort has been spearheaded by Michael Schwern, leading to several benefits. We now have 6 times the number of testcases, which test a cleaner codebase with more documentation. The Perl Bug database has been switched over to Request Tracker [http://www.bestpractical.com/rt/]. We should thank Richard Foley for his work on perlbugtron, which has now been retired. After several discussions on what a memory leak is, several memory leaks and cases of 'naughty access' have been fixed. To acheive this, we've used 'third degree', 'purify' and the most excellent opensource alternative 'valgrind'.

More numbers

Nicholas Clark, Hugo van der Sanden, and Tels have done some magic 'keeping integers as integers as long as possible' work, and when finding bugs in vendors number-to-string and string-to-number, they coded around these to increase precision. We should all be happy that 42 is now 42 and not 42.000000000000001; imagine what the aliens would do if they found out.

Documentation

I mentioned several documentation pages earlier, and they are part of the 13 new pod files included in Perl. In addition to this all README.os files have been translated into pod. Interesting to note are several new tutorials including `perlretut`, `perlpacktut`, `perldebtut`, `perlnewmod` and “a gentle introduction to Perl” `perlintro`. There is also a brand new POD specification written by Sean M. Burke in `perlpodspec`.

Depreciations

Several depreciations have occurred in Perl. In future versions of Perl, 5005threads will be gone and replaced by 'ithreads'. Pseudo-hashes will be killed, but the `fields` pragma will work using restricted hashes. Suidperl, which despite everything isn't safe, which is a bare package with unclear semantics, is also depreciated.

A few things have been removed and forbidden, such as blessing a ref to another ref. Self-tying of arrays and hashes led to some weird bugs and has been disabled; they touched some rarely tested codepaths. The [[.cc]] and [[=c=]] character classes are also forbidden because they might be used for future extensions. Several scripts that were outdated have been removed and the upper case comparison operators have also been axed.

The war of the platforms

Perl 5.8 works on several new platforms: the EBDIC platforms were regained, but sadly we lost Amiga, so any volunteers that want to make the Amiga port work again are very welcome.

Odds and Ends

There is a long list of new small changes in Perl 5.8. The biggest of these small changes is restricted hashes which can be used from the new `Hash::Util` module and lets you lock down the keys in a specific hash; this will possibly be used as a replacement for pseudohashes for the `fields` pragma.

Threads in Perl 5.8

Arthur Bergman <arthur@contiller.se>

This talk is about Perl 5.8 threading, which introduces shared variables. Hear about how threading applies to mod_perl2.0 and learn how the make existing modules threadsafe and possibly threadfriendly. XS comes last, so that fearsome people can drop off for their own threads.

Part III. Applications

ELEKTRA Peripheral Simulation

A Perl Application Program

Wolfgang Laun, ALCATEL Austria AG
<Wolfgang.Laun@alcatel.at>

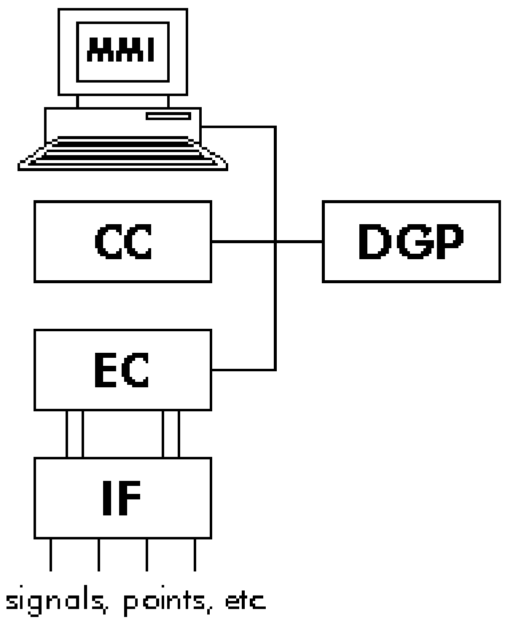
Abstract

ELEKTRA is the brand name of ALCATEL, Austria's electronic railway interlocking system. There are, at present, around 70 ELEKTRA installations in three countries. The biggest installations are in Wels, Upper Austria, (188 points) and in Salzburg (173 points).ELEKTRA is implemented in Perl.

The ELEKTRA Electronic Interlocking System

Interlocking systems are meant to control all the trackside and internal equipment of a railway station to guarantee safe and reliable operation, most importantly by establishing train and shunting routes. To achieve this goal, the ELEKTRA architecture (fig. 1) employs a set of interconnected nodes, distributing the tasks of the man-machine interface (MMI), central control (CC), element control (EC) and diagnostics processing (DGP). The application software is generic, while the configuration for some specific railway station is implemented as a set of distributed databases, one for each node.

Figure 1.



The EC handles the task of controlling all the element interfaces through a number of input and output ports. Setting an output port may turn on a light point of signal or throw over a point, whereas input ports reflect the current state of an element. While the configuration of ELEKTRA nodes is (almost) independent of the actual set of elements of some railway station, the set of element interfaces depends on the equipment that has to be controlled. Given that (even for a small station) the element interfaces, which, for safety reasons, are implemented in relay technology, would occupy several cabinets, it is not feasible to execute laboratory tests either of the ELEKTRA application software or of the databases for a specific station on a fully equipped system. Even if one would set up all the interfaces, the signals, points and other gadgets would still be missing. A full system is installed on the destination site, but it is equally impractical to use these systems for development tests. (Note that the installed system still has to undergo final acceptance and concordance tests.)

For testing of the the generic software and of a station's database we therefore simulate the peripheral equipment, including the interfaces. In addition to the simulation proper, system testers also expect a convenient environment for developing and executing the tests. And finally, there are also so-called subsystem tests, where just the MMI node and the EC node is tested all by itself, for which also some simulation and test support is required.

Requirements for the ELEKTRA Peripheral Simulation (EPS)

From the ELEKTRA architecture and from the way testing by developers, testers and data preparation personnel is done we derive a number of technical and operational requirements.

- Communication with ELEKTRA nodes is done over ethernet and TCP/IP, using socket connections. Since there may be up to 8 ECs (depending on the size of a station) and each EC consists of two independent software channels (ECA and ECB) the number of connections is variable. Messages from ECs arrive asynchronously.
- There has to be a graphical user interface (GUI) representing a subset of the station elements. This GUI must provide a feature for easily changing between the "free" and "occupied" state of some track or point, and it should permit all other state manipulations, including the instillation of interface errors.
- The set of elements varies with the country-specific implementations of the ELEKTRA system. The simulation tool must adapt to varying sets of elements, and the subsystem test tool must be able to handle several different protocols to the MMI and EC subsystems.
- Some subsystem tests require the assembly of binary data in the target system representation in a portable way.
- The simulation programs should be portable across various platforms. At present, our laboratory workstations run Solaris (on Sun Sparc) and Linux (on Intel).
- It must be possible to run system and subsystem tests driven by test scripts and without requiring any operator interaction.

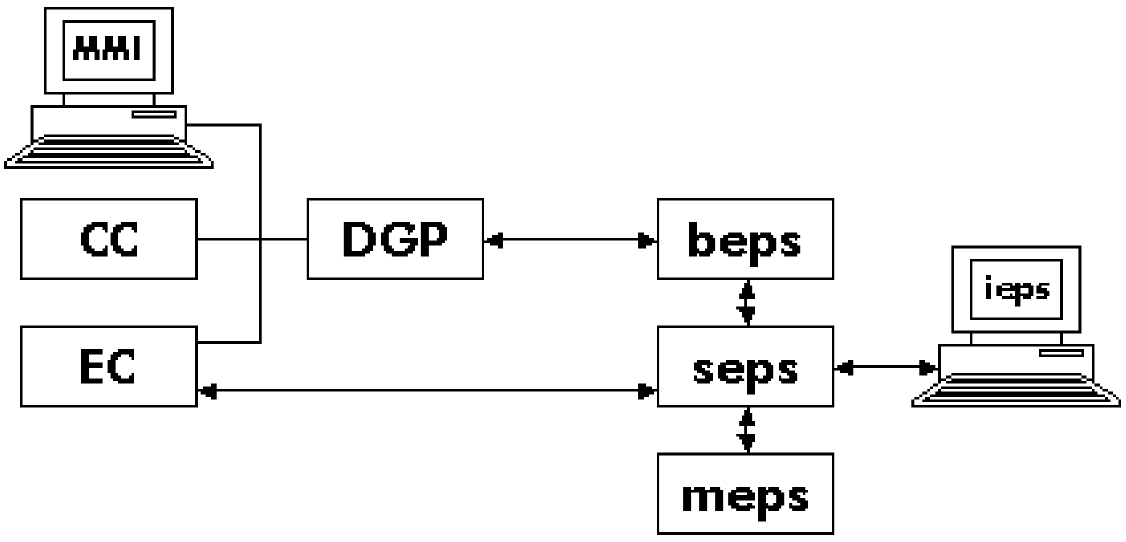
Additional requirements such as reusability, easy maintenance and object-orientedness are dictated by generally accepted software engineering rules.

The EPS Programs

A Quick Survey

The ELEKTRA Peripheral Simulation consists of a number of Perl programs (fig. 2). All of the programs - now totalling 48,000 LOC - are written in Perl 5. Individual programs may be combined as required for a certain test, and can even be started on different nodes of the laboratory network.

Figure 2.



The individual programs are:

- EPS/Simulation Server (seps). This program responds to messages from an EC (containing output port settings), from an attached EPS/Interface, EPS/Batch or EPS/Train Motion Simulator (all containing external status changes). Resulting changes are returned to the EC (input port settings) and to the EPS/User Interface.
- EPS/User Interface (ieps). This GUI program permits interactive inspection and manipulation of all element and interface states.
- EPS/Batch (beps). This is a command interpreter supporting the execution of test scripts for system and subsystem tests.
- EPS/Train Motion Simulator (meps). This program simulates the progress of a train from some start signal until it reaches a signal commanding "halt".
- EPS/Line Handler (leps). This program (which is not shown in fig.2) is a protocol converter required for communicating with an EC of the previous hardware generation which does not have an ethernet connection but provides a serial port instead.
- EPS/Track Editor (teps).This is a graphics editor for creating the track map used by the EPS/Interface program.

Let's have a closer look at the most interesting EPS programs.

The EPS/Simulation Server Program

This is the program doing the actual simulation chores. The behaviour of each element (and its interface) is modelled by a Perl package, which inherits enough from the `Element` base class to make it quite simple. As an example, here is the Austrian subsidiary signal simulation:

```
package OBBsusig;
use strict;
BEGIN {
    use vars qw( @ISA );
    @ISA = qw( Element );
}
#####
```

```
# Constructor: use default
#####
# evaluate: re-compute all voters and supervisors
#
sub evaluate($){
  my $self = shift();
  # Voters
  $self->voters();
  # Lamps
  $self->{lw_} = $self->{faultlw} || $self->{ssp} ?
                 0 : $self->{pwa} && $self->{pwb};
  # Supervisors
  my $h;
  $self->{swa} = ( $h = $self->{faultswa} ) ?
                 $h-1 : ! $self->{lw_};
  $self->{swb} = ( $h = $self->{faultswb} ) ?
                 $h-1 : $self->{lw_};
  $self->{sp} = ( $h = $self->{faultsp} ) ?
                 $h-1 : ! $self->{pif};
  # Plug-In
  $self->pluginfail( '[vs]w.*', 'l.*' ) if $self->{pif};
}
1;
```

Inherited subroutines take care of creating the object and of computing interface reactions common to all IF types. What remains to be done here is the reaction to the current state of the output ports (pwa and pwb, from ECA and ECB, respectively), and the setting of the interface's input scan lines (swa, swb, sp). Inherited subroutines are made simple by relying on data from a configuration file where all I/O lines and external states are defined. The relationship between ports (one for each channel) and the scan lines returning the port state (here: vw*) is implied by adhering to naming conventions. Here is the definition for the same signal:

```
# subsidiary/caution signal
#
Element OBB OBBsusig e_susig
Ports
  pwa re_epa_susig_port Port white lamp
  pwb re_epb_susig_port Port white lamp
Scans
  vwa B 1 e_v_a_speed1 Voter A white lamp
  vwb B 1 e_v_b_speed1 Voter B white lamp
  vwab A 1 e_v_ab_speed1 Voter A+B white lamp
  swa A 1 e_s_speed1 Supervisor white lamp
  swb B 0 e_s_speed1 Supervisor white lamp
  sp A 1 e_sc_plug_in_fail_spm_3 Plug-In LAM3
Lamps
  lw white lamp
  lw_ white lamp/blinking
Stats
  pif Plug-in-Fault LAM3
  ssp Signal voltage
```

The actual configuration of a station is defined on data files where HW addresses are assigned to output ports and input scan lines. These files are read after program start, and element objects are created accordingly. After that, seps sets up a server socket and waits for connect requests from ECs, responding with the current state of that ECs interfaces. During operation, an EC sends messages indicating the state change of some port, wherupon the corresponding evaluate method is invoked and resulting changes are returned to the EC.

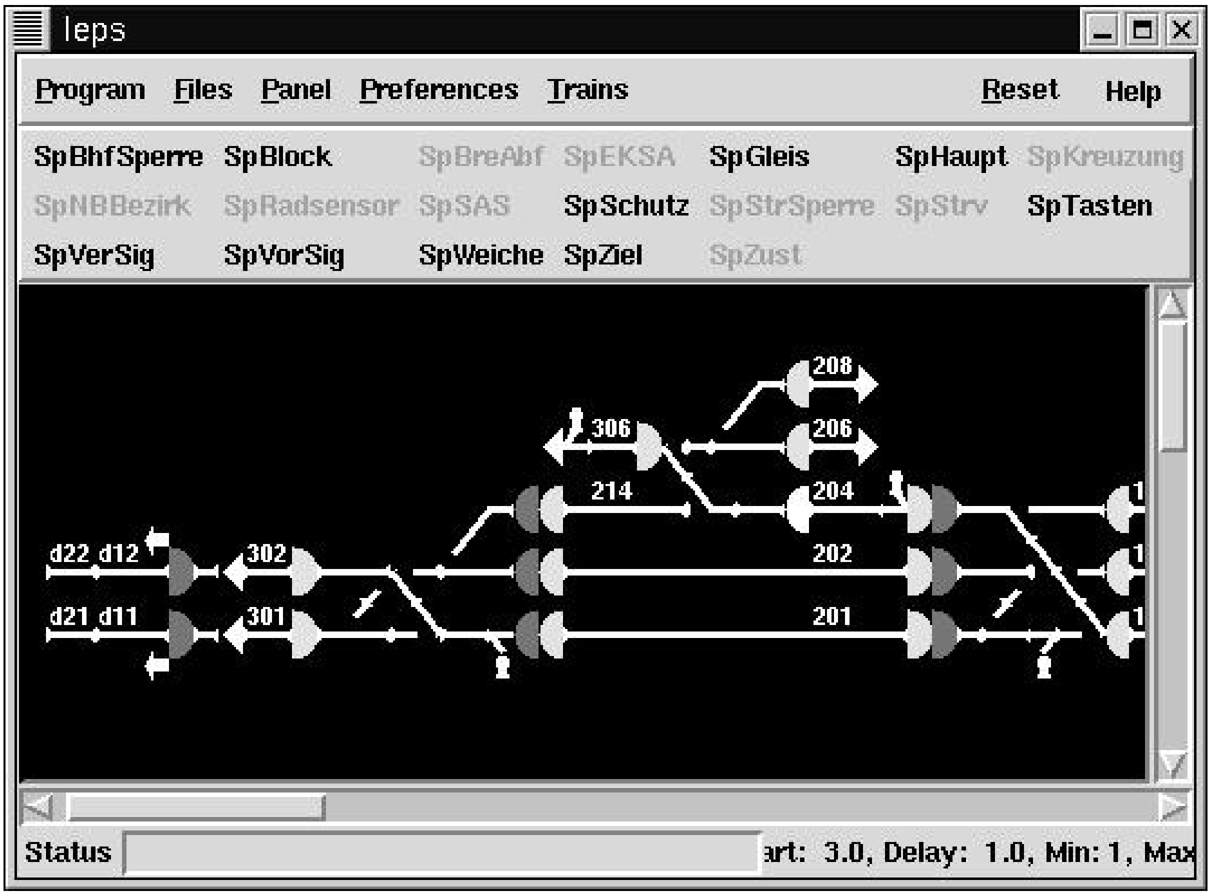
The server socket may also receive a connect request from an EPS/User Interface (ieps) program. The simulation server responds with the states of all elements. During operation, a message from ieps may request the change of some external state or indicate an interface malfunction. Again, the evaluate subroutine takes care of computing the reactions, with changes being sent to the corresponding EC and back to ieps. Similarly, an EPS/Batch program may connect and send messages indicating state changes or interface faults to the simulation server. An element status query is provided for documenting an element's state during the system test. Finally, also an EPS/Train Motion Simulator (meps) may connect. The command expects commands putting trains in front of signals (originating from ieps or beps) and responds with state changes of track relays or axle counters.

At present, seps supports the simulation of element sets for three different country applications (ÖBB, SBB and MAV). In Austria, the ELEKTRA system may also be used on top of a relay interlocking system, replacing the control desk with lamps and pushbuttons. For testing such systems, EPS provides two additional package sets for simulating ALCATEL and Siemens relay interlocking systems. (This is, essentially, the implementation of an interlocking system, although some features such as flank protection are not included.)

The EPS/User Interface Program

This is, of course, the showpiece among the EPS programs (fig. 3). Its main feature is a canvas displaying a track map showing (at least) the tracks, points, crossings, main and shunting signals of a station. Track- related elements change their colour to red when they are "occupied", points show their current position, and signals change when they alternate between the "halt" and some "drive" aspect.

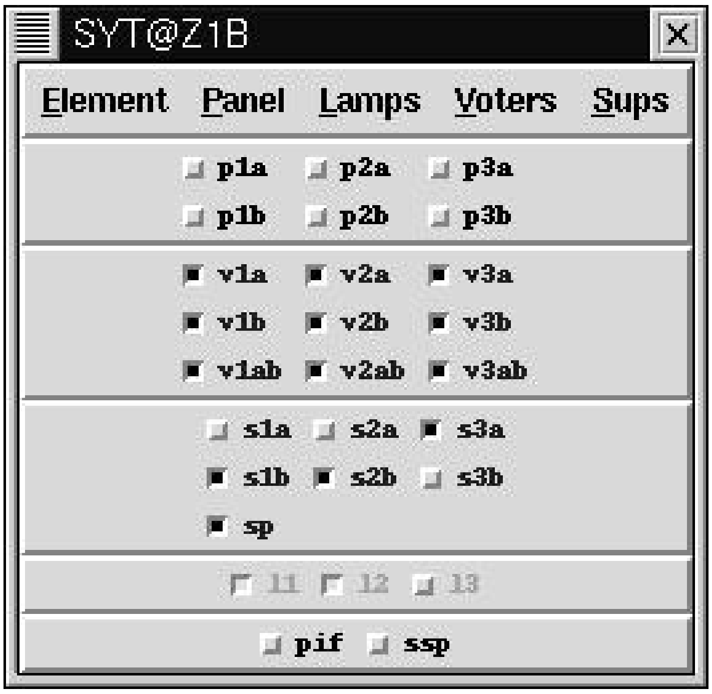
Figure 3.



Clicking mouse button 1 on a track-related element toggles between the "free" and "occupied" states. Clicking on a signal places a train in front of a signal, with its further movement to be handled by EPS/Train Motion Simulator. Other status changes can be made readily available by using a "status box", which is a button widget in combination with some checkbuttons mapped to some element's states.

Clicking with another mouse button on any element, or selecting an element from one of the element lists, creates a pop-up window showing the element state in full detail, indicating the state of all ports, scan lines, lamps and other external states. (Fig. 4 shows the pop-up window for the Swiss shunting signal.) With more than 100 different elements around, programming as many pop-up windows individually had to be avoided. Thus, pop-up windows are created by a single Perl/Tk subroutine that interprets the element definitions (as shown in the preceding section) and creates frames and checkbutton widgets accordingly.

Figure 4.



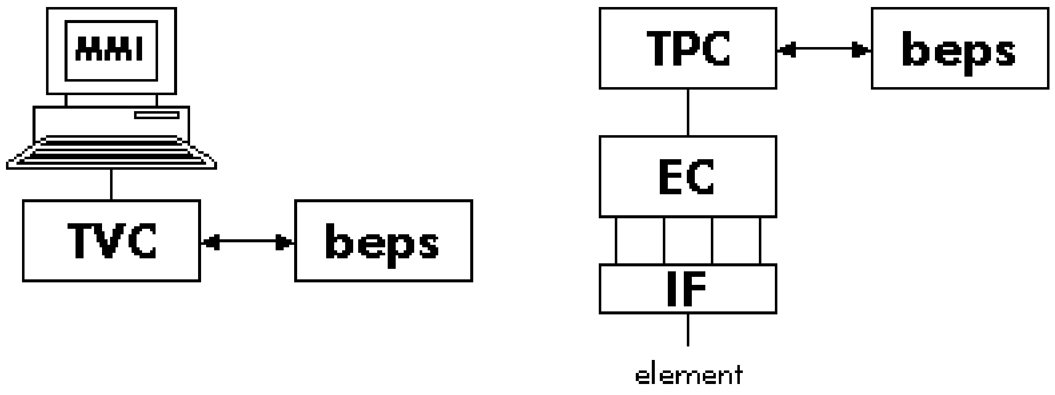
The EPS/Batch Program

EPS/Batch (beps) is a command interpreter featuring typical functions such as

- symbol assignment and substitution
- canned command sequences ("macros")
- conditional jumps
- log and error file handling

beps is used in connection with the EPS/Simulation Server as well as stand-alone, for the MMI and EC subsystem tests (fig. 5). Common to all test configurations is the requirement to communicate with an ELEKTRA node (DGP, TVC, or TPC) handling messages to be forwarded to some other node and returning data to be logged as test results. The problem is, however, that the messages and the data returned differ with respect to country and test setup. This was the incentive for implementing the generic "extend" command which will require a Perl file containing a set of additional command implementations and a subroutine capable of parsing and logging the returned test results.

Figure 5.



Implementing the command interpreter in Perl turned out to be a pleasant experience, even though backward compatibility required the implementation of some quirks and turns that were inspired by features readily available with VMS/DCL.

A pretty feature added to beps is the status window, which is implemented as a separate Perl/Tk program spawned from beps. The Perl/Tk program is fed status information through a pipe, and continuosly displays information such as the current working directory, the last command executed, the stack of active macro invocations, etc.

Perl and EPS

Why Perl?

Several years ago, a colleague gave me the documentation of Perl 4. Having mastered awk, I saw no compelling reason for turning to Perl. At some later time, when I became responsible for supporting the SW development environment for ELEKTRA, I was beginning to look for a way of implementing GUIs on UNIX. I began experimenting with Tcl/Tk and decided that the Tk part was great but that Tcl was somewhat Spartan. Soon after I discovered that Perl 5 had arrived, with lots of new features and, above all, with a Tk port available, and I decided to give it a try. After completing a few Perl programs, the need for porting and extending the then existing ELEKTRA simulation environment, part of it written in Ada (on VAX/VMS), and the GUI part being a Borland Pascal Program (on Windows 3.1) to UNIX became my biggest problem. Having gained some confidence in Perl (including the Tk package), it was Sriram Srinivasan's book "Advanced Perl Programming" which finally encouraged me to implement it all, from scratch, in Perl. (This book also contains the description of "Jeeves", a template driven generator, which helped to avoid a considerable amount of work, elsewhere in the ELEKTRA project.) Today I can say that the decision turned out well.

Some Highlights on Perl Features

It has been said that Perl is a language for getting your job done. Needless to say, even with Perl a large program still requires a lot of effort. But sometimes you feel that Perl is giving you some extra benefit. This section mentions a few instances of this happening in the EPS project.

The use and require functions.

Being able to extend a program's code or data at runtime with some dynamically selected file is a valuable asset. In EPS this is used to load a set of element simulation packages, depending on the "country" environment for which the simulation is being run, avoiding useless loading of foreign simulation packages. Another place where "require" was put to good use is in connection with the EPS/Batch command "extend", to load a set of command extensions. And, last but not least, "require" is a fast method for loading large bulk data, typically prepared with `Data::Dumper`. This is used in a couple of places for defining the network of elements making up a station and the set of defined train and shunting routes permissible on that network. (Note that an interlocking system's operator may not command arbitrary routes.)

The eval wand.

Being able to compile and execute snippets of data containing Perl code at runtime has helped in a few places. First, there is the EPS/Batch interpreter, whose expression language for symbol assignment and conditionals is pure Perl, with all symbols going into some package namepace. Parsing of expressions, evaluation, symbol table management - imagine what you'd have to implement in some other language! Even a "show symbols" command is implemented easily by accessing the `%package::` hash.

Messages containing object data that are passed between EPS programs contain that data in Perl syntax, typically in the `key=>value`list form. This makes sending and receiving independent from the actual contents, since the data is self-describing. Moreover, one does not have to worry about formatting and parsing routines, and textual representation avoids all potential pitfalls of a binary representation.

The pack animal.

One of the subsystem tests requires the generation of a considerable variety of binary messages containing all sorts of numeric, bit and character string data to be sent to the test target. Since new message variants are apt to appear every now and then, the definition of new messages should be as easy as possible. The solution combined (somewhat hidden) pack and the Perl class concept, with classes being defined *dynamically* (rather than statically with some `package` command). The code snippets below should give you an idea how this works:

```
use constant BYTE => 'c';
use constant INT => 's';
use constant LONG => 'l';
use constant BINARY => 'a';
use constant ASCII => 'A';
use constant ASCIZ => 'Z';
Frame->Extend( 'X25', # creates subclass X25
   [ 'len', UBYTE ], len => \&Frame::Length,
   [ 'hg', UBYTE ],
   [ 'snd', BINARY x 4 ],
   [ 'rcv', BINARY x 4 ] );
X25->Extend( 'X25mel', # creates sub-subclass X25mel
  snd => $mel_sender_betr_0, rcv => $mel_empfaenger_vc,
  [ 'elem', UINT ] );
```

The constants (named according to data types in the ELEKTRA programming language) are synonyms for pack codes. The `Frame` base class provides the method `Extend`, which may be called to create subclasses (`X25` and `X25mel`). Array reference arguments define (additional) fields of a message frame, and other arguments are used for setting field values. To create the packed message, the `Pack` method may be called for some subclass, with arguments to override field values:

```
X25mel->Pack( elem => 42 );
```

Other Experiences and Conclusion

The biggest problem with Perl is the complexity of the language. Mastering Perl (if one may ever claim having done that) does take a considerable amount of time. Moreover, with the language constantly evolving, maintenance and upgrading to new Perl versions makes you worry whether all of your Perl code is still stable on all platforms... Still, the quality assurance work being done by p5p is impressive, and I am pleased to say, that actual bugs never were much of a problem while the EPS implementation was under way. (One problem with an obscure pack code could be solved by a workaround, and the fix in the Perl runtime code was soon available. Responses from p5p to a couple of other bug reports were always quick and helpful.)

As far as Perl/Tk is concerned I can only say that it met all of my expectations, even though some of the trickier things required a little try and effort. I'm still adding to a collection of Perl/Tk how-tos.

Besides EPS, a number of other tools supporting the ELEKTRA software development were implemented in Perl, which has by now become the prime implementation language for this purpose. We have set up training program teaching Perl to some of our SW engineers and testers. We feel that Perl will continue to be of great value to our department.

Case Study: Psychometric Testing with Perl

Karen Pauley, Kasei Ltd. <karen@kasei.com>

Psychometric tests are now a standard part of many recruitment processes. Traditionally these are hand-scored by qualified psychologists, at great expense and with a long delay between taking the test and getting the results. We were approached by a company who wanted to produce these tests on-line, so that companies could get instant results from candidates who sit the test.

They wanted a system that they could use to set up different tests for different clients, enter the scoring rules (which could be different for different types of tests), and have the system produce reports on each candidate who sits the test. This was to be web-based and browser-independent, and the 'look and feel' of each test should be easily customisable to look like each client's own web site.

And, of course, it had to be completed in several weeks at minimal cost.

In this case study we'll look at:

- What to do when you have to build a system in a hurry.
- Making the CPAN work for you.
- Reducing your workload by 90% by using `Class::DBI` and `Template::Toolkit` as 2 legs of an MVC system.
- Reducing the remainder of your workload by another 80% by building a simple, generic, extensible, reusable web-based-forms controller.
- Managing customer expectations with eXtreme Programming.
- How a heavyweight Configuration Management System (Aegis) can help speed the development process up dramatically.
- How to build a $20,000 web site in an afternoon.

NSRL Distributed Hashing with Sexeger Improvements

Douglas White, National Institute of Standards and Technology
<nsrl@nist.gov>

Abstract

The National Software Reference Library [http://www.nsrl.nist.gov] (NSRL) is designed to collect software from various sources and incorporate file profiles computed from this software into a Reference Data Set (RDS) of information. The RDS can be used by law enforcement, government, and industry organizations to review files on a computer by matching file profiles in the RDS. This will help alleviate much of the effort involved in determining which files are important as evidence on computers or file systems that have been seized as part of criminal investigations [Fisher]. The harvesting of file profiles from software is accomplished by multiple computers in a loose parallel computing environment. The implementation of the environment is discussed, with attention to performance improvements using reverse regular expressions known as "sexeger".

Introduction

When software destined for the National Software Reference Library arrives in our library room, it is placed on a "new arrival" shelf. The software is taken from the new arrival shelf and the information such as application name, version, manufacturer, etc. is entered into our database, and a unique identifier is allocated to the software package, as well as identifiers for each piece of media in the package. A label is placed on the box or case, and media are marked with indelible ink. Once labeled, the software may then be placed on a shelf in the actual library, or while it is presently off the shelf having been labeled, batching can occur.

The batching process takes files from the original media and places them on our file server. Once batched, file profiles are constructed for every file on the media. These profiles are comprised of information that allow unique identification of every file. Part of the profile information is a set of hash values [RFC1321] [FIPS 180-1] that are computed condensed representations of a file [Hash] [Pieprzyk]. The hashes are the most popularly used features of the NSRL RDS and this paper focusses on improvements in the hashing process verification time by using reversed regular expressions.

Environment

We made the NSRL hashing process modular, and we use a parallel method of hash calculation. After software media have been batched an array of computers perform the hash calculations. There are six 166 MHz Windows 98 computers that perform the distributed hashing work. We call these six machines our hashing "constellation." This term was chosen because the term "cluster" is widely used to designate a parallel configuration of computers much more integrated than is found in our project.

Distribution of Hashing Work

We have a linux computer on our network that is the master of the distributed hashing process. It uses cron to run a Perl script every hour [cron]. The master Perl script connects to a database server and runs a query to find out if a piece of media has been batched and requires hashing. If so, the master script starts a hashing process (also a Perl script) on an available constellation computer.

The heart of the NSRL hashing process is an executable called WALKER.EXE. This is a compiled C implementation of the CRC32, MD4, MD5 and SHA-1 algorithms. The hashing process applies WALKER.EXE to a list of files to generate hash codes for each file and merges the WALKER.EXE output with the list of files, checking for anomalies. This merge allows a precise match of hashes to the location of the original file. Any error messages, format inconsistencies, etc. are identified, logged and reconciled if possible. Valid data will continue through the hashing process steps, invalid data will result in messages to administrators notifying an incomplete data set from this media. We want the process to continue with as much valid data as possible and we will investigate invalid data later.

Many of the consistency checks use reversed regular expressions known as sexeger, popularized by Jeff Pinyan (japhy) [sexeger] [RevRegEx]. When we implemented sexeger, we noted an average 55% reduction in time used to perform consistency checks, and a 32% reduction in time overall in the environment when we implemented sexeger. This choice was probably the best improvement we made in our coding of the environment.

The bulk of the sexeger code is used in finding and comparing filename extension strings (.tar, .zip, .jpeg) at the end of long strings and in traversing absolute directory path strings in reverse to identify parent-child relationships in media structures.

Performance Metrics

We retained the timing data from the automated hashes of 92 pieces of media using a previous consistency check algorithm, and we rehashed those 92 pieces of media using a new algorithm that included reversed regular expressions, or sexeger. During early implementation, timing was taken in increments of seconds; for consistency this was repeated on later runs. Here are the timing results for the old and new algorithms:

Old Consistency Checks:

Hashes performed: 812049
Average number of hashes per media: 8826
Effective hashes/s: 74

Code section	Seconds	% of run time
initialization:	18	0%
walker hash:	11010	7%
consistency:	80044	56%
find/extract:	49124	34%
total time:	140984 (39h)	100%

The times above clearly identify the consistency checks as the initial location for attempting performance tuning.

New Consistency Checks (with sexeger):

Hashes performed: 812049
Average number of hashes per media: 8826
Effective frame="all" hashes/s: 75

Code section	Seconds	% of run time
initialization:	18	0%
walker hash:	10795	11%
consistency:	35830	37%
find/extract:	50127	51%
total time:	96770 (27h)	100%

After implementing sexeger, a clear 55% reduction in consistency check time and a 32% overall execution time reduction was observed.

During the months of April and May 2002, we rehashed the entire NSRL software collection. The new hashes calculated were verified against the previous RDS to ensure that the new algorithms were not perturbing our results. The rehash allowed us to view the process end-to-end under reasonable stress. In six weeks, we batched and hashed 1583 pieces of media. This period included days where 70 CDs were batched in a 9 hour period, and it included days when all staff were on travel. On average, slightly over 1,000,000 files were hashed each week. Here are the timing results over the entire collection:

Total Collection:

Hashes peformed: 6423307
Average number of hashes per media:4057
Effective hashes/s: 34

Code section	Seconds	% of run time
initialization:	306	0%
walker hash:	184724	11%
consistency:	620288	37%
find/extract:	872811	52%
total time:	1656605 (460h)	100%

Conclusions

Measurement of time spent in major sections of the hashing execution allowed us to identify the worst performance. Implementation of counter-intuitive algorithms brought considerable time savings to the process. We found that the modularity resulted in an eventual ten-fold gain in productivity. During the months of April and May 2002, the environment showed the ability to hash and verify 1,000,000 files each week. In the current configuration, when five staff batched during the workday, the six 166 MHz constellation PCs hashed slightly faster than the work accumulated.

There are several improvements we can make to increase productivity. An increase is needed because the process is very close to saturation, and we have seen that software released on DVDs (more popular now) will extend the average time a constellation CPU is performing one task. A finer unit of timing is needed in benchmarking in the future, probably using the Benchmark and `Time::HiRes` modules. We will be seeking improvements in the archive extraction code and in the database manipulation code. We will be evaluating the `File::MMagic` module for internal verification of file types.

Biography

Douglas White has worked at the National Institute of Standards and Technology since 1987. His experiece has covered distributed systems, distributed databases and telecommunication protocols. He has written programs in many areas, including real time biomonitoring, real time video processing, web site/database integration, system administration scripts and network monitoring scripts. He holds both a B.A and M.S. in computer science from Hood College.

Bibliography

[Fisher] G. Fisher. 2001. *NIST ITL Bulletin, "Computer Forensics Guidance"*. http://www.nsrl.nist.gov/itlbulletin.html.

[RFC1321] R.L. Rivest. 1992. *IETF RFC1321, "The MD5 Message Digest Algorithm"*. http://www.ietf.org/rfc/rfc1321.txt.

[FIPS 180-1] U.S. Department of Commerce. 1995. *NIST FIPS 180-1, "Secure Hash Standard"*. http://www.itl.nist.gov/fipspubs/fip180-1.htm.

[Hash] T. Boland. G. Fisher. 2000. *"Selection Of Hashing Algorithms"*. http://www.nsrl.nist.gov/documents/hash-selection.doc.

[Pieprzyk] J. Pieprzyk. B. Sadeghiyan. 1993. *"Design of Hashing Algorithms"*. ISBN 0-387-57500-6.

[cron] P. Vixie. 1993. *Cron man page*. http://unixhelp.ed.ac.uk/CGI/man-cgi?file.

[sexeger] J. Pinyan. 2000. *"sexeger"*. http://www.perlmonks.org/index.pl?node=sexeger.

[RevRegEx] P. Sergeant. 2001. *"Reversing Regular Expressions"*. http://www.perl.com/pub/a/2001/05/01/expressions.html.

ELRIC - An Intelligent Autonomous Agent

Richard Jelinek, PetaMem, s.r.o. `<rj@petamem.com>`

Abstract

This paper introduces a system called ELRIC, that has been constructed for extensive natural language processing and that is about to show incremental natural language understanding abilities. Todays communication infrastructure and technology allow easy generation and mass transportation of data, which in turn can only be properly analyzed by humans.

This imbalance between generation and analysis shows the need for machine augmented text analysis, or - even better - autonomous text analysis and processing of an agent according to his masters (presumably a human) wishes.

Introduction

Todays technology provides the possibility to generate much text in one of the current natural languages and to send this text to many people. Because this possibilities are even easy to achieve, many people generate much text and send it to many people. This results in many people obtaining far too much text every day and - because of the global village - night.

There are two major and extreme strategies how to cope with all these texts. The first is to simple ignore all of it and just focusing on some peers (lets generalize: information sources) that are reliable for not polluting your input. The second is to be very conscious and to thoroughly analyze everything that comes across your input, reading it from a to z. What if there's an interesting piece of information hidden in it?

Because these strategies are extreme, probably no one will stick only to one of them[7]. Most of us end up in some kind of mixed strategy: Using mail filtering software, fast scanning subjects and happily deleting texts in hitherto unknown encodings.

As another example take some big company delivering its products that need extensive documentation to several countries with different languages. Each change in the product that requires a change in the documentation will require a change in all languages. Therefore either increasing the time to market of that product or lowering the quality of the documentation. Sometimes both.

In this paper we will introduce a system that has been designed to cope with this kind of problems. It is a major project in natural Language Understanding (NLU), it is being made by a very small company (3 developers) and it is coded in Perl.

Strategy Two

Strategy one in our mail filtering example above is probably impractical, as you probably don't even know who all of your potentional peers are and thus block some of them a priori. Strategy two would be ideal, but who has the time to dwell through all this vast amounts of data - most of them being irrelevant?

For some participants of the new media mass data transportation system (referred to as The Internet), it is out of question if they should adopt strategy two. Companies with at least some interest in customer relationship simply must process incoming messages. But how? Either you have to answer to an email with some standard text or you have to drive a whole call-center with dispatchers and supporters to process - probably even answer these incoming messages.

Lets consider some email like `info@ibm.com`. Would you - as a customer - write some query there? Probably

except 3l33t geeks that decide for one or another...

not, because you know what this means. A 300 000 employees global company will not be able to read, classify and process the x thousand mails that come in each day, in various languages, about various topics (consider a pre-sales question for ViaVoice in Spanish and a support question for a zSeries mainframe). A tremendous task - even more so if the subject is a ``Help me!!!"

But there are potentional customers in there and some already existing customers, that urgently need help and if they don't get it, they probably won't be customers for long. All major companies probably do one or more of the following things to augment their support staff:

- spam filtering/procmail
- automatic language recognition and forwarding
- automatic response generation (pre constructed texts)
- help-desk with cut&paste functionality for supporters

There are commercial and non-commercial products for spam filtering such as SpamAssassin `(www.spamassassin.org)` using Bayesian Filtering for content in addition to the rules for blocking of from addresses or some regexps for subjects.

Unfortunately, the deeper text analysis itself is currently bound to humans only. This imposes several constraints on this services also: Supporters may have the relevant technical competence, but not the relevant language this competence is needed in.

And last but not least you cannot assume, that a single person[8] can keep track of all kind of allowed contents to pass the company gateways. Not to talk of security and privacy issues.

And all of the above is just about analyzing and delegating incoming data. No word about generation. So as for the support issue it seems, that employees profound in many languages and having whatever competence is needed to do a supporters job, able to work night and day at nearly no cost, being always friendly would be what global companies need.

We think, that Perl makes hard things easy and we've already done some things with it, even when we were told that they were impossible. So this is a perfect opportunity: Create an intelligent agent, that is able to semantically analyze natural language text, retrieve information from it, store it, infer on the stored information, probably even make assumptions and generate natural language output according to whatever is needed.

ELRIC Architecture

The following shows some simplified architectural concepts of ELRIC. Up to now, the system is a running prototype. Because of its suitability for such work, Perl is used thoroughly for the whole implementation. It may well be, that some computational intensive parts (such as statistical NLP) will be re-implemented in C, but there seems no urgent need for that now.

or a well organized group of people

Figure 1. Top-level view of the ELRIC communication architecture

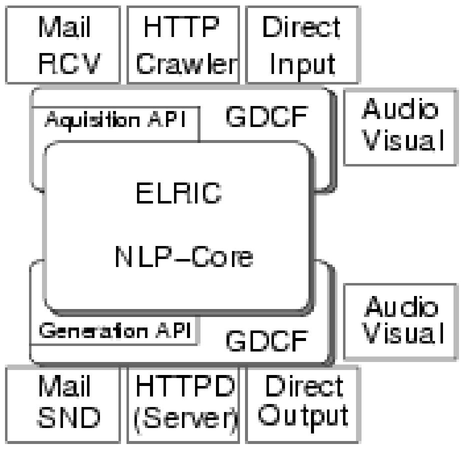


Figure 1, “Top-level view of the ELRIC communication architecture” shows the top-level view on the central communication architecture of ELRIC. It is primarily designed for communication via Email, HTTP, CLI (command line interface). The design takes future audiovisual input/output into account.\footnote{In fact, audio input is now experimental with ViaVoice and audio output is currently experimental with the Festival Speech Synthesis System (`http://festvox.org/festival/`)}

Figure 2. GDCF Architecture

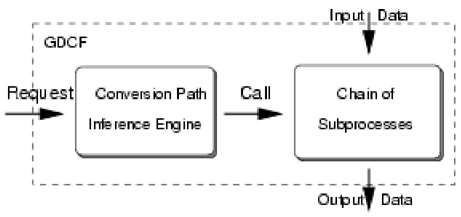


Figure 2, “GDCF Architecture” shows a greatly simplified GDCF (General Data Conversion Fabric) architecture. GDCF has a basic knowledge about the available data formats, the converters that can turn one data to another plus the options and parameters one has to give to them to do these conversions. It then (optionally) searches at startup for these converters. Depending on how many of them (and in what version) it can find, it builds up a $n x n$ conversion matrix, where n is the number of different formats. It also tries to find conversion paths by chaining several converters together for not existing conversion paths in the first place. The format it feeds to (and expects from) the ELRIC NLP-Core is EUML - ELRIC Unicode Markup Language. Which is an Unicode XML with specific Markup for NLP.

Figure 3. ELRIC NLP-Core

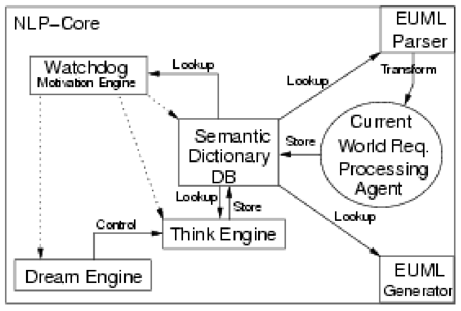


Figure 3, “ELRIC NLP-Core” shows a detailed view of the components of the NLP core. The central structure is the semantic dictionary DB. This holds declarative and procedural world knowledge information. The ``Think Engine'' is the central Inference Engine over the semantic dictionary DB. The so-called ``Dream Engine'' is a component that is controlling the ``Think Engine'' in offline-mode to reconfigure some contents of the semantic dictionary DB according to deductions made through newly acquired data. A thorough integration of newly acquired data in real-time has shown to be impossible.

Figure 4. Current World Request Processing Agent

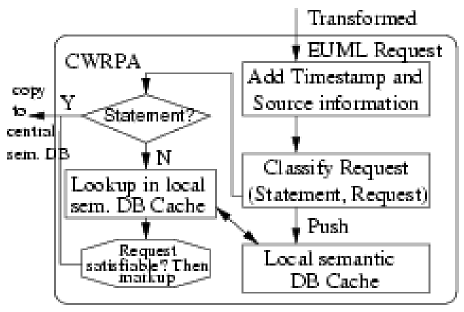

Transformed
EUML Request
CWRPA
Add Timestamp and Source information
copy to central sem. DB
Y
Statement?
N
Classify Request (Statement, Request)
Lookup in local sem. DB Cache
Push
Request satisfiable? Then markup
Local semantic DB Cache

Figure 4, “Current World Request Processing Agent” shows the Current World Request Processing Agent. This component is responsible to keep track and dispatch requests made to the system. It gets EUML data from the EUML Parser (part of the NLP Core) - which in turn got the raw EUML from GDCF - adds some basic information such as timestamp and source, then dispatches according to whether the incoming data is a simple statement (in that case the information is stored) or a request (which has to be processed by the available resources i.e. knowledge). A request may trigger some data acquiring action.

Try it.

No warranty whatsoever, but you may send an email to `elric@petamem.com` in whatever language you want, with whatever text-attachment you want containing whatever type of text you want. Please don't send any non-text attachments such as pics, mpegs, powerpoint presentations or spreadsheets. But you may send plain ASCII, html, pdf, ps, doc, rtf and so on documents. ELRIC will do its best to answer you by telling you something about your email.

But How...

...Does it actually work? Up to now, this is confidential. Mainly because company NDA restrictions and because PhD publishing restrictions. What we can tell is that:

- Knowledge is divided into declarative and procedural. The formalism for declarative Knowledge is a n-th order predicate calculus with over-saturated logic junktors and inherent 3-value logic. Quantors are also formalism-inherent.
- Procedural knowledge is represented by a turing-complete formalism. Yes - Perl.

People with a formal education in theoretic computer science will immediately object, that n-th order is not decidable and that one runs into problems with the standard halting problem.

This is true.

But both doesn't matter.

Let's say there is something someone wants you to do - probably solving the traveling salesman problem. Not knowing anything of the theoretical background you may do this for up to 4 or 5 points. You will - given a request to solve it for 20 points - either end up with a ``Don't have the time for that'' or come with a unoptimal thus ``wrong'' solution.[9]

Or actually dig into the matter and find the enormous complexity

The keyword is motivation. While executing a Perl program you always can have a supervisor watchdog that will set \$SIG\{ALRM\} and pull the plug if the timeout is reached. The time given to an execution of code is something like the motivation of the program. The faster you think, the more intelligent your thinking routines are and the more motivated you are - the more problems you may solve.

Any Hints?

You still haven't told us exactly how it does work. Just give us some hints.

True. Imagine yourself learning a foreign language. How did you do it? And in which programming language were you taught?

Given *that* ``programming language'' - what are you supposed to do with it if you need your knowledge (declarative or procedural) in some formalism?

Shoulders of Giants

We admit and are proud to be standing on the shoulders of giants. As a 4-man company it wouldn't be possible to achieve anything with this complexity within a year of work without the possibility of using technology (Perl,Linux), work (CPAN-Modules) and support (Perlmonks, newsgroups) where we may incorporate the results of thousands of man-months. Be they mythical or not.

We would like to thank the Perl community and feel an obligation to give back what we receive. By contributing spin-off technologies to CPAN, supporting authors by doing some unsexy janitors work that is needed when maintaining code we hope to support the voluntary work that is done in this field. With financial contributions to support some organizational events we hope to give our part to glue the community. And last but not least gathering more and more Perl expertise and growing up of some more hardcore Perl coders, we hope to become another oasis for camels.

Nemo: TeXmacs the Scientific Editor with Perl

Stéphane Payrard `<s.payrard@wanadoo.fr>`

Abstract

Nemo is TeXmacs with Perl. Nem, Niki, Nirc and Nile will be the Nemo tool thanks to TeXmacs and Perl.

What is TeXmacs?

TeXmacs is an interactive documentation tool with a power comparable to TeX. It does not include or use any TeX code except for font handling, neither emacs code; but, in addition to a menu-driven interface, it supports keyboard accelerators in a way comparable to emacs but somehow different too because TeXmacs has a more complex and (currently) specialized purpose.

TeXmacs is already used to write complex articles and books with complex maths, TOCs, indexes, footnotes... The current main hindrance is more sociological than technical, as editors prefer to receive Word or TeX documents.

TeXmacs is GPL, available on Unices with X-windows system. It is written in C++, currently uses Guile as an extension language and its own widget library. I have near completed a shallow port using Qtapplication event loop that will be quickly followed by a shallow Perl embedding including a Qt adaptator for POE. Watch CPAN and look for mQT and POE::Kernel::mQt within a week! After that, a complete port to Qt will be done to facilitate port toward non unix-like-systems. The Perl interpreter will eventually have more access to the TeXmacs internal structures.

What will be Nile?

The Nemo project will eventually culminate in a year or two in Nile (Nemo Interactive Literate Environment) with dynamic syntax hiliting that will support languages like per6 with an unlimited number of Unicode operators. More, Nile opens to these languages the full expressivness of mathematical notation (indices, exponent, matrices...). Think of Nile as literate IDE.

I will present TeXmacs as seen by users, will move to how to a quick review of TeXmacs internal and file format, then will explain how to extend TeXmacs using Perl. Finally I will expose the basic rudiments behind the upcoming parsing/deparsing engine behind Nile, the upcoming interactive literate environment. Steps toward Nile will be Nem (concurrent edtions over the net), Niki (a wiki), Nirc (a TeXmacs client that supports Math Notations).

Application of TeXmacs

0.	TeXmacs	15 mn
1.	Translation of Perl documentation from pod to TeXmacs (Pod::TeXmacs) The generated doc is hypertext with TOC and index.	5 mn
2.	A specialized edition mode for pods A pod stylesheet and some Guile code	10 mn
2.a	An infix format for stylesheets	
3.	Using Perl as TeXmacs extension language	20 mn
3.a	A shallow port using Qt event loop	

	One can use QtApplication to get a "real" event loop while still keeping the current TeXmacs gadget based toolkit.	
3.b	Using POE to network TeXmacs Thanks to a POE::Kernel::mQt adaptator	
3.c	A complete port to Qt: a preliminary to a port to windows	
3.d	A deep Perl Embedding. Necessary to access the edition tree from Perl	
3.e	Translation of Guile code into Perl The Guile code to boot and configure TeXmacs will stay but the Guile interpretor can optionally go because this code is simple and Guile is easy to parse so can be interpreted by Perl as well. One issue: name folding.	
4.	Nemo	10 mn

4.1	Parsing/Deparsing motor
	
	This is the very nemo engine. A unique feature will be dynamic syntax hiliting (SH). With SH, how your code will be parsed will become obvious. The dynamic introduction of Unicode operator in Perl6 will make SH a must have:
	
	`<blue>a</blue><red>+</red><blue>b*c</blue>`
	`<black>a+</black><blue>b</blue><red>+</red><blue>c</blue>`
		10 mn

Q&A		5 mn
		-----
		60 mn

TeXmacs will be his own presentation tool. For further information see TeXmacs main site [http://www.texmacs.org] TeXmacs, the wiki wiki web [http://www.alqua.com/tmresources] TeXmacs with Perl, soon to come [http://nemo.mongueurs.net]

OpenMMS - Telco Industry Meets Perl

Implementing an MMS server in Perl

Jozsef Dojcsak, Leolo IT & Media Consulting GmbH
[http://www.leolo.com] <jozsef.dojcsak at leolo.com>

Abstract

Telecom software solutions seem to focus on Java, as formerly they have primarily used C++ or proprietary systems like Erlang. Though it is rare, there are Perl deployments around: some billing and customer relationship modules has already been known to be written in Perl.

Leolo Consulting implemented an SMS Processing and Routing system, the Pigeon SMS System (PSS). This system had been deployed and has been in use at a major telecom company in Germany. This system is almost entirely implemented in Perl, based on a self developed Perl class library, using advanced event driven server model and flexible communication features.

So we gained some telco experience and when the first MMS phones arrived to Europe, we decided to develop MMS Messaging Solutions. As a first step we started developing an MMS testbed system, naturally in Perl...

This paper will give you a short introduction to Multimedia Messaging Service, also mentioning some related technologies and uncovering implementation details of our MMS solutions.

About MMS

Multimedia Messaging Service, or MMS for short, is an instant messaging service for the mobile environment with rich content, including media types like image, audio and text.

MMS Message

An MMS message is not just a text message with multimedia attachments, it offers more: a presentation language, which describes the layout and the sequence of appearance. (can be imagined as a limited Powerpoint presentation)

The current versions of MMS uses a subset of SMIL (Synchronized Multimedia Integration Language) as presentation language. For more information on SMIL, see http://www.w3.org/AudioVideo/.

An MMS message encapsulates all media and presentation parts in one MIME multipart message. This multipart message is compatible with standard internet email formats, though it is binary encoded as defined in the WAP specifications.

MMS Communication

MMS uses the Wireless Application Protocol (WAP) as a bearer technology. This means that MMS is a WAP Application, using the technology and infrastructure of existing WAP systems. (well, upgrade to WAP 1.2.1 is required in order to get compatible with MMS)

In the WAP communication model, a so called WAP Gateway is used for communication with the mobile phones. This gateway translates the binary data from the phones to HTTP request and renders the textual HTTP response to binary PDUs. The Web server which gets the HTTP requests is the "Origin" server, addressed directly in the WAP browser.

MMS applies a 'store and forward' distribution model, where the mobile terminals send the message to an MMS Server, which is responsible for storing it and notifying the addressed recipients. The recipient is then responsible for downloading the message from the MMS Server.

MMS communication is achieved with MMS transactions, which in turn consist of Protocol Data Units (PDUs). Typically every requests are responded with an acknowledge PDU by the receiving party.

MMS Server

The MMS Server, or as frequently mentioned MMSC, have a central role in MMS systems. It is responsible for managing MMS client transactions, authorization, storing messages and forwarding messages to recipients. An MMSC is typically splitted into two parts: MMS Proxy-Relay and MMS Storage-Server.

The MMS Proxy-Relay has the HTTP interface for communicating with the clients via a WAP Gateway, and manages most aspects of MMS handling. The MMS Storage-Server is responsible for providing storage facilities for MMS messages until delivery to recipients or until expiration.

Background

SMS + WAP = MMS

As you might have known, WAP has been introduced to overcome the difficulties of using modern internet technologies like WWW on mobile devices. Among others, one of the important reasons to use WAP is to reduce the bandwidth requirements over the air. Due to this requirement, WAP intensively uses binary encoded messages, such as binary encoded HTTP headers, MIME multipart emails or even binary encoded XML files!

This property of WAP makes it very effective in use, but parsing or composing WAP data might require enormous efforts. Fortunately, the WAP Gateway does this job in most cases, but in case of MMS transfer, the Gateway passes the binary encoded data to the Web server untouched.

The latest WAP specification has a feature called WAP Push, which is extensively used by MMS. This WAP Push technique allows sending content to the end-user's phone. The current implementation of WAP Push is not more than a special SMS message: contains a so called User Data Header, which addresses a special port of the target phone.

MMS Solutions

Testbed MMS System

MMS is a fairly new technology, only a few, mostly "scandinavian" companies offer MMS products and solutions. We decided to develop our MMS platform but first of all, we wanted to get acquainted with this technology, so we decided to implement a "testbed" system. It was a natural selection to use Perl for this purpose (too).

Ingredients

Table 1. Testbed MMS System Components

Component	Description
Apache	We used our public 1.3.x Apache web server as MMS Server. For the ease of simplicity we used plain CGI scripts, which will be ported to mod_perl later.
CGI script used as MMS Proxy/Relay	The URL of this script should be set in the mobile phones using our system. It fairly complex, but not more than 600 lines. It implements almost all functionality of an MMS Proxy/Relay.
Filesystem directory used as MMS Storage	MMS systems should typically implement complex storage models, in order to handle quotas, expiration, authorization, etc. We decided to use the simplest possible storage model: plain directories containing messages a binary MMS files. (The storage of binary files had another advantage, we could resolve phone incompatibilities by studying these files)
WAP Push	It is implemented as a PSS::NET module, which allows SMS sending with UDH. Since WAP Push is not supported by most WAP Gateway, and typically re-

Component	Description
	quires special contract to use, we use our SMS sending application to directly create the binary SMS with the special UDH.
MMS parser and composer module	This is the heart of our system, it contains the logic and information to descramble and compose the binary MMS messages.
MMS mailbox manager	The delivered and pending MMS messages can be accessed via the Web. This application can be used to repeat notifications, and check the system status.

Experiences

The whole system was implemented in 2 weeks. Most time was spent on studying the fairly huge documentation suite and trying to resolve the phone-to-phone message sending incompatibilities.

OpenMMS

OpenMMS is an independent MMS solution for content providers and telecom companies. OpenMMS will be also available as a public service, contracted users can use it as MMSC or for distributing commercial MMS messages.

Features

- Dedicated mod_perl webserver: The HTTP interface, which is used by the WAP Gateways or the authorized users is a dedicated mod_perl based system, which forwards MMS traffic to the MMS Management Daemon.
- MMS Management Daemon: MMS communication and transaction processing is based on a more effective MMS Management Daemon. This solution allows more sophisticated working modes and higher security. The daemon is also responsible for sending out WAP Push notifications.
- Storage management: The MMS storage uses a database backend, with advanced quota and expiration management.
- Individual Postbox Access via Web: Configured users are able to see their MMS mailbox via the web, web base MMS composition and sending is also possible.
- Central Administration: The administration interface provides full control to all aspects of the MMS Server environment.
- Application Interfaces
- MMS UA Profile Handling
- Legacy phone support (with an adaptive, self-learning database)

Implementation Details

MMS Modules

For our testbed system and for OpenMMS, we have implemented generic modules for composing and parsing binary MMS messages. We have developed the MMS::Lite module, which is an all-in-one MMS module, with simplified object model and we have also built a full MMS module library with a more complex object model, which is more flexible, but not so easy to use.

The MMS::Lite module is available for download on Leolo's MMS site: www.leolo.com/mms [http://www.leolo.com/mms].

Please do not hesitate to download and use it. More MMS modules will be published soon, so stay tuned…

MMS Examples

Let us see this module in action, let us compose an MMS message:

Example 1. MMS User Agent Example

```
#
# MMS User Agent Simulator
# =======================
#
# we are going simulate what the mobile phone does...
#
use strict;
use MMS::Lite;
# this is very similar to MIME::Lite
my $mms = new MMS::Lite(
  # in the MMS::Lite module, we handle headers and content together
  From => '+36309965027',
  To => '+36309965027',
  Subject => 'test MMS',
  Parts => [
    # id => part-definition
    'start' => {
      Type => 'text/plain',
      #Data => "<smil>.</smil>",
      Path => '/data/x.smil',
      Start => 1,
    },
    'text.txt' => {
      Type => 'text/plain',
      Data => "Hello World",
      # or Path => '/data/text.txt',
    },
    'text.txt' => {
      Type => 'image/jpeg',
      #Data => $bin_image,
      Path => '/data/img.jpg',
    },
  ],
);
# let's create a binary encoded PDU:
my $mms_pdu = $mms->pdu_data(
  Operation => 'send', # 'send' is the genric name of 'm-send-req'
);
# note, if we got undef, then we might have missed a mandatory parameter
# for the given Operation
die unless $mms_pdu;
# Now we can send this data through a Wap Gateway, but we can directly
# POST this data to the so called "origin" server via HTTP POST
# (not included is this example)
# use HTTP::Request ... with Content-Type: application/vnd.wap.mms-message
# in the HTTP response we should have received an 'm-send-conf' PDU:
my $resp_mms_pdu = ...; # extracted from the HTTP request
my $resp_mms = MMS::parse_pdu($resp_mms_pdu);
die unless $resp_mms;
print $resp_mms->as_string;
```

MMS Module Internals

The MMS and related WAP specifications contain a large amount of tables containing the binary header field and attribute codes. These tables were converted with the help of some Perl scripts into hash tables and included by the MMS parser module(s).

The MMS fields are binary encoded variable length fields, the field length and thus the extraction method depends on the field value type. Fortunately, only a limited set of value types are used in the MMS messages, so we could implement a flexible solution:

Example 2. MMS Header field tables

```
# the included MMS header fields are defined as follows:
      # lookup tables for 'code' type fields
      ...
      %yes_no = (
         0x80 => 'Yes',
         0x81 => 'No',
      );
      ...
      %mms_fields = (
        'Bcc' => {
          id => 0x01,
          type => 'string',
          # Encoded-string-value = Text -string
          # | Value-length Char-set Text-string
        },
        'Cc' => {
          id => 0x02,
          type => 'string',
        },
        'Content-Location' => {
          id => 0x03,
          type => 'string',
        },
        'Content-Type' => {
          id => 0x04,
          type => 'ct',
          xref => \%content_types,
        },
        'Date' => {
          id => 0x05,
          type => 'long',
        },
        'Delivery-Report' => {
          id => 0x06,
          type => 'code',
          xref => \%yes_no,
        },
        'Delivery-Time' => {
          id => 0x07,
          type => 'delta',
          # Delivery-time-value = Value-length (Absolute-token Date-value
          # | Relative-token Delta-seconds-value)
          # Absolute-token = (Octet 128)/Relative-token = (Octet 129)
          # /Delta-seconds-value = Long-integer
        },
        'Expiry' => {
          id => 0x08,
          type => 'delta',
        },
        'From' => {
          id => 0x09,
          type => 'varstring',
          # From-value = Value-length (Address-present-token Encoded-string-value
          # | Insert-address-token )
        },
        ...
      );
      # create lookup table by ID
      my %mms_field_lookup = map { $mms_fields{$_}->{id} => $_ } keys %mms_fields;
```

For the parsing and composition of messages we have defined type-handler tables:

Example 3. MMS Header field type handler tables

```
      my %parser_type_map = (
        'code' => sub {
          my ( $dref, $xref ) = @_;
          my $value = ord $$dref;
          $$dref = substr($$dref, 1);
          return $xref->{$value};
        },
        ...
      );
      ...
      my %synth_type_map = (
        'code' => sub {
          my ( $value, $xrref ) = @_;
          my %xref = map { $xrref->{$_} => $_ } keys %$xrref;
          return pack("C", $xref{$value}) if defined $xref{$value};
        },
        ...
      );
```

Finally, the following code is used in the parsing loop:

Example 4. MMS Header parsing

```
      while( length($bin_data) )
      {
        my $field_code = ord $bin_data;
        if ( defined my $header_field=$field_lookup{$field_code} )
        {
          my $type = $mms_fields{$header_field}->{type};
          my $xref = $mms_fields{$header_field}->{xref};
          die unless defined $parser_type_map{$type};
          # skip header field
          $bin_data = substr($bin_data, 1);
          # parse value
          my $value = &{$parser_type_map{$type}}(\$bin_data, $xref);
          ...
        }
        ...
      }
```

This parsing solution was proved stable and fast, and it also allows flexible definition of forthcoming new fields or dynamic handling of MMS format versions.

Unicode handling

Since the encoding of the text part of an MMS message could be us-ascii (lower half of ISO 8859-1), UTF-8 or UTF-16, we needed a solution for handling Unicode texts in Perl. Fortunately, the CPAN module Unicode::String was perfect for this problem:

Example 5. Handling Unicode MMS Texts

```
      my %mibenum_map = (
       # MIBEnum to encoding mapping:
        3 => 'latin1',
        100 => 'utf8',
        1000 => 'utf16',
        #...
      );
      die "Unknown character encoding!"
          unless defined $mibenum_map{$content_type->{encoding}};
      Unicode::String->stringify_as( $mibenum_map{$content_type->{encoding}} );
      my $us = Unicode::String->new( $part->{content} );
      ...
```

Since some Email Clients cannot correctly handle Unicode text attachments (Netscape - Mozilla series), we need a non-unicode string, when forwarding the message to email recipients:

Example 6. Using Unicode MMS Texts

```
      # use this in a non-unicode system:
      $data = $us->latin1;
```

We look forward for the native Unicode support of perl version 5.8.0.

Drawbacks

Originator Address

You might think that many operator independent public MMS Servers will pop up in the near future. There is security constraint which limits this process: the privacy of most information of an MMS message should be well protected. Protected information is the destination address, the originator address and the message payload itself.

The Originator Address is protected so much that it is only known by the mobile phone operators. Most modern WAP Gateways could extract this information, but it is a trusted information and will only be provided to trusted (contracted) parties.

Phone incompatibility

Though MMS mobile phone manufacturers agreed to comply with common conformance specifications, slight differences can be observed between the implementation details of MMS support in mobile phones.

Discovering these differences allowed us to fix the incompatibilities...

About the author

Jozsef Dojcsak works for Leolo IT & Media Consulting GmbH as senior consultant.

Leolo Consulting

Leolo Consulting is an international company providing professional database and internet technology consulting services around Europe and California.

To read more about the MMS topic, please visit www.openmms.de [http://www.openmms.de] or www.leolo.com/mms [http://www.leolo.com/mms].

Can a Company Use Perl to Develop - and Sell - Commercial Tools?

The SANFACE Software experience

Fabrizio Sanface <sanface@sanface.com>

Helmar Wodtke

Why Perl?

At the beginning we didn't select the language.

Seven years ago I, Fabrizio Sanface, was an Unix administrator and I started to use Perl in preference to script shells (bsh, csh, ksh, etc.).

Four years ago I started to develop the first SANFACE Software tool: **txt2pdf**.

I'm not a programmer and the only language I knew was Perl.

Then, four years ago, we decided to distribute this shareware tool written in the Perl language. It was a very strange choice! Now we're very happy about a similar strange choice.

With few modifications we transformed our tool into a cgi to create PDFs on the fly on our web site [http://www.sanface.com/createpdf.html]. It was one of the first web application to make PDFs on the fly.

Using a tool similar to a library we created more complex web services like http://www.sanface.com/flash4/ferrari/.

Today everything is more simple and it's possible to develop tools using Perl, transform the Perl code into executable versions and distribute only the executable versions.

We've tested 2 tools: Perl Dev Kit by ActiveState [http://www.activestate.com] and **perl2exe** by IndigoSTAR [http://www.indigostar.com].

With Perl Dev Kit it's possible to develop using Perl and create executables for Windows, Solaris, Linux and HP-UX.

Two years ago, we selected **perl2exe** approach because:

- **perl2exe** allowed us to make executables for more operating systems (Windows, Solaris, HP-UX, AIX, Linux, BSD, Digital Unix and now Mac OS X)
- **perl2exe** allowed us to make executables from the same OS (e.g. from our Windows 2000 or also 95 we can create executable for every other OS).

We're waiting for Perl 6 news.

Probably our tools are the only tools that can make PDFs on OS's like: OpenVMS, MPE, OS/390, EPOC, etc. in the same way on every OS and with the same code. Obviously using Perl.

Another important point we're evaluating, is the possibility of making a Graphical User Interface (GUI) for our products using Perl/Tk. At the moment we've made a Visual Basic (VB) GUI for the Windows distribution. Windows customers like the GUI in Windows style.

Why commercial tools (shareware)?

Four years ago only a few software houses had products that could create PDFs. These tools usually converted

PostScript to PDF or were libraries.

A text to PDF converter was an innovative idea and we thought we could earn well selling this product.

At the beginning, we didn't understand the true market.

Our first customer was Alex project [http://www.infomotions.com/alex/] and we thought that the target of our product was to convert textual books into PDFs.

Then we understood that a lot of companies have a lot of applications that produce textual reports (old legacy applications written using cobol, ERPs, datawarehouses, DBs, etc.). Usually the companies purpose was to print the textual reports over forms and send them via normal mail or fax. Simply we gave to these companies an application that worked the same way. We gave them the possibility to design a background form (using a part of PDF syntax) and to put the data (the textual report) over it making nice PDFs.

With a simple tool their old applications can create, from simple textual reports, nice and powerful PDFs.

At the begin we wanted to give to our customers the possibility to add a lot of new features, adding to the textual reports special marks. Then we understood that the companies don't want to modify their original textual reports.

They simply need to have them in a new format that help them to print them, to send them via email, to publish them on web, to fax them. PDF is the best format to solve these problems.

We don't force the companies to add tags, we give them external files where they can set general rules using powerful regular expressions (RE).

e.g. with txt2pdf PRO is possible to use fontmark external file with syntax like

```
;;SANFACE Software;/F12;12
```

to use font /F12 point size 12 with every string "SANFACE Software" or more complex example like

```
<b>;</b>;#!btag#.*#!etag#;/F5;14
```

means: the start tag is <b>, the end tag is </b> delete the tags and use the /F5 font with point size 14 to every word (.*) inside <b>(#!btag#) and </b>(#!etag#)

Is it possible to sell tools, distributing the source code?

I'm Italian and when I talk to other Italian people they tell me:

"You're crazy. How can you think to sell a tool distributing the code? Are you not afraid that people can copy the code and the tool?"

We do not protect the registered version.

The only way we use to protect our code is the License Agreement.

We trust our customers and we trust people.

Is it the correct way or we're crazy?

Here is some data about our business:

txt2pdf is popular at C|NET [http://download.cnet.com/]: this means that at the moment it has been downloaded 23,000 times. In the same way we're present at TuCows and in every other Download site. We spent a lot of time with web marketing (try to search for e.g. **txt2pdf** or sanface in a search engine). We estimate that **txt2pdf** has been downloaded more than 100,000 times worldwide.

On the other hand we have 600 customers. 95% of our customers is located in the US.

Why a tool (why not a module)?

We've talked a lot of times about this point. This is our point of view:

Companies usually don't need modules or libraries.

When a company has to solve a problem it wants an open tool that can solve its specific problems.

Usually money isn't a big problem, time is a very big problem, to find the right person with the right knowledge is a very big problem.

The only point I was sure was: we've to make a very open (STDIN and STDOUT) server batch tool that the customer can use from any program on every OS to convert very big volumes of data in a small time.

Also, we give our customers very quick and good support.

If you've ten days and you find a very good tool but:

- It's very complex to use, to install, to whatever...
- You've a problem and the company is too slow to support you
- You need a customisation or a different feature but the company can't help or support you, you won't buy a similar product.

We know a lot of companies who bought our tools and used them like modules inside their applications (e.g. cobol and java applications).

Description of few projects

http://www.sanface.com/hfmusproject.html.

Hachette Filipacchi Media U.S., Inc. (HFM U.S.) [http://www.hfmus.com/], the New York-headquartered subsidiary of Hachette Filipacchi MÃ©dias, (the world's largest magazine publisher) began a project in 2001 to decommission our last remaining mainframe system. The single application running on it was almost 100% COBOL and responsible for acknowledging and invoicing orders places for advertising in our magazines. The project was to move it to Windows NT without rewriting it all.

A main obstacle was output. The COBOL was spitting out thousands of acknowledgements and invoices monthly on preprinted multipart forms, not to mention reams of paper reports, all using line printers and 1st-column carriage control. We sure didn't want to try to configure NT to output to a line printer! txt2pdf PRO helps us to solve the problems [http://www.sanface.com/pdf/hfmus.pdf]!

http://www.sanface.com/halifaxheraldproject.html.

The Halifax Herald Limited, one of Canada's oldest and largest independent newspapers.

Like all metro daily newspapers, the Herald publishes hundreds of display ads and thousands of classified ads every day, resulting in thousands of daily invoices and monthly statements. For years we printed duplicate bills so the Accounting Department would have file copies in the event an advertiser had questions or required a reprint.

Obviously, a significant amount of time and space was occupied handling these bills -- separating, filing, finding, re-filing -- all very manual processes. The bills themselves begin as huge ascii files that are printed on continuous pre-printed forms using IBM 6400 line printers.

In 2000, we developed a Perl-based document management system that indexed the ascii files by invoice number. This allowed the Accounting Department to search for a particular invoice or statement and display it on their screen. The system was quite convenient for them but not robust enough to allow us to stop printing and filing duplicates.

It proved that a document management system would save time, money and storage space, and allow us to better serve our customers. Managing, protecting, indexing and cross-referencing all these ascii files was a primary challenge. An Oracle database, fronted by a cgi interface using DBI/DBD and Perl was the answer. The other major concern was the integrity of ascii files -- they're easy to alter. This made PDF desirable. Customers, auditors and taxmen could agree that PDFs were exact replicas of original bills.

How to convert our ascii files to PDF? A quick search of the web turned up txt2pdf PRO. Right out of the box it produced perfect PDF versions of our bills. Of course they were still plain text, hard to read on the screen and painful to print on line printers.

txt2pdf PRO's overlay/underlay capabilities provided a perfect solution -- underlays exactly replicating our preprinted forms, color-matched and complete with the Herald logo. Now the PDFs look just like the bills we print, making them easy to read on the screen and suitable for reprinting on laser printers.

While evaluating txt2pdf PRO we discovered it also converted all our large reports accurately to PDF. These reports are hundreds of pages thick and many are just for historical record. (http://www.sanface.com/pdf/heraldbill.pdf)

Why does the Perl community not like commercial tools?

The Perl community is the only freeware community (we know) that doesn't like commercial tools. It's impossible to announce a new commercial tool version in a Perl newsgroup and in Perl site.

It's impossible to publish a Perl commercial code at CPAN.

This is also true for tools like our **txt2pdf** that is shareware and we distribute with the Perl code.

Our point of view is: the Linux community is grown up because it's based on freeware core but it allows commercial tools and partnerships.

We think we're in an very strange position.

We're probably one of the few companies that use Perl to develop its tools, but Perl community doesn't like our work.

We publish everywhere in our site we're using Perl and we'd like the Perl community to use our knowledge and our projects with big companies (http://www.sanface.com/projects.html)

The market and other communities like Linux, OpenVMS, MPE, cobol like our tools.

We think we're showing that

"a company can use Perl to develop (and sell) commercial tools".

We'd like other companies to do the same with the help of the Perl community

We hope this paper is the first step!

Extreme Perl

The Horror That Is SelfGOL

Damian Conway `<damian@conway.org>`

In this talk I dissect the SelfGOL program: an obfuscated, self-aware, viral quine that can:

- self-replicate,
- rewrite other Perl programs to allow *them* to self-replicate,
- detect un-rewritable Perl programs,
- execute itself or other Perl programs as cellular automata of arbitrary size (to play Conway's "Game of Life"),
- animate any short text as a cycling marquee banner.

SelfGOL accomplishes these feats in under 1000 bytes of standard Perl, without importing any modules, and without using a single **if**, **unless**, **while**, **until**, **foreach**, **goto**, **for**, **next**, **last**, **redo**, **map**, or **grep**.

To do all that in under 1K of code, it relies on some extreme programming techniques, and on many of the obscure backwaters of the Perl syntax. This talk explores both.

In other words, it's everything you never wanted to know about Perl, and would have been afraid to ask.

Part IV. Internet

Creating Dynamic Sites in a Flash with the Template Toolkit

Casey West <casey@geeknest.com>

If your boss has ever asked you to build a web site, you've no doubt been subjected to this declaration: "You need to build the best web site ever built by anyone in the history of web site building, with all the whiz bangs, doo dads and gizmos that make marketing drones drool, with half the fat. Plus, you should have been done yesterday; this will come up in your performance review." Oh sure, they never come right out and use those words, but that's what they mean.

I'm going to show you how simple it is to put together such a production quickly and easily. Using a combination of the Template Toolkit and mod_perl - through the Apache::Template module - you have the reusability factor of Server Side Includes (SSI) and the dynamic nature of mod_perl and CGI. Apache::Template is very easy to configure, and through the techniques I'll demonstrate, you'll be creating web sites right quick. With any luck, you'll be able to cut your caffiene intake as well as make your boss happy.

Extending the Template Toolkit

Mark Fowler <mark@twoshortplanks.com>

This talk covers various techniques for extending the Template Toolkit. It starts with a quick overview of how the Template Toolkit works, moving onto discussion of developing Plugins, Filters, and adding Virtual Methods. It moves onto discuss using views effectively to render existing objects and datastuctures and concludes showing this technique applied to XML.

How to Build Large Scale Websites/Web Applications with Embperl 2.0

Axel Beckert <beckert@ecos.de>

This talk shows how to build large scale websites or web applications with Embperl. It discusses the concept of separation of application logic, display logic, content and layout and how this separation can be achieved with Embperl 2.0.

First, the talk shows the importance of application objects and how they are created, how to break up whole pages into components and how Embperl's inheritance model is working and what are its benefits.

It then continues with a short introduction into Embperl's processing pipeline, which allows the creation of component output within multiple transformations, giving the possibility to include nearly every data source (like e.g. POD, XML, RSS, SSI, CGI scripts or plain text) and rendering techniques (like e.g. JSP, ASP or PHP).

The talk will comprise embedding Perl into HTML, using static XML, generating XML dynamically, doing XSLT transformations and using XSL-FO to generate HTML, XML, PDF, text and various other output formats.

It will also touch other features being necessary to build large scale websites like session handling, caching, database access and internationalization.

This talk will be similar to Gerald Richter's Embperl talk given at OSCON 2002. (See http://conferences.oreillynet.com/cs/os2002/view/e_sess/2716)

Introduction to `Net::DNS`

Casey West <`casey@geeknest.com`>

This introduction to `Net::DNS` will give an overview of the most useful features of `Net::DNS`. This module does more than DNS lookups and I'll give you a taste of it's crunchy goodness.

How to Integrate Anything with SOAP

Jonathan Stowe <gellyfish@gellyfish.com>

SOAP started life as the Simple Object Access Protocol, one of the productions of the W3 Consortium's XML Protocol working group. It defined a relatively lightweight means of doing Remote Procedure Calls (RPC) using XML over a pre-existing network transport protocol. The 1.2 version of the specification somewhat downgraded the acronym SOAP as features were added which probably meant that it not quite so Simple anymore and the RPC 'Object Access' simply became one of the ways in which the protocol be used for communication between systems.

Perl has, for quite some time, had good support both for XML processing and for handling a variety of common network transport protocols and thus was in a good position from the start to be able to provide good support for SOAP and so was able to take its deserved reputation as an excellent 'glue language' further beyond the boundaries of a single system or application than it previously had done and into the arena of application integration - previously the preserve of expensive and complicated middleware.

This will talk will include a a brief introduction to the history of and the rationale behind SOAP and the reasons why Perl was in a good position to be an early language for which a toolkit was available to build SOAP applications.

The remainder of the talk will focus on the use of Perl and SOAP in the integration of diverse applications and will include discussion of:

- Reusing Perl code in non-Perl applications
- Extending the capabilities of non-Perl applications to do things that are only sensible to write in Perl
- Why I love .NET and BizTalk server even though Microsoft are still bad and evil.
- The choice of Perl SOAP Toolkit and a comparison with some non-Perl ones.
- Why you might want to use the 'Document Style' rather than RPC and how you might do this.
- Why you might not want to look into the inside of `SOAP::Lite` if you value your sanity :)
- What are "Web Services" and how are they different to SOAP?
- The reason to choose between the various transport mechanisms.

There will be working examples demonstrated in Perl and C++ (and possibly C# or Java) although this will be at a higher level than a tutorial.

This should serve both as an introduction to using SOAP with Perl for the uninitiated and also as a demonstration to people already familiar with it some further ways that it can be used.

Part V. Perl Quality

Creepy Featurism

Mark Overmeer <mark@overmeer.net>

'Featurism' has a bad name: adding extensive features (functionality) to your code. More features means more code; more code means more complexity. More featurs will reduce the overall speed, and increase the chance on bugs.

However, consider the alternative as well. Less features force users to program more around your code. And not everyone is an as gifted programmer as you are! Besides, where other people start filling-up the holes you let, they stumple on unexpected problems and bother you with questions. With some bad luck, they post modules extending your code, which break each time you upgrade your work.

It is hard to select between features to implement and features to skip. However, if you decide to include a functionality in your module, then you should do it in the full extend: with documentation, tests, and examples.

This talk takes a simple feature as example -- "search for a string in an e-mail message" --, and shows what implications this has for an Object Oriented program (you do not really need to know the details on Object Oriented programming) It shows how you can add complex features, and still keep the code relatively straight forward and testable.

Tales Of Refactoring: Remaking MakeMaker and Other Horrors

Michael G. Schwern <schwern@pobox.com>

Refactoring is the process of gradually making code better without changing its functionality. It can be used to redesign existing systems, or to cleanup and modernize old ones. Herein will be related Tales Of Refactoring In Perl. How to go about it, what works, what doesn't and some particularly nasty pitfalls to avoid. We'll see how reworking an existing code base can often succeed where a grand rewrite will fail.

We'll look at:

- The modernization of Test::Harness and MakeMaker.
- The relationship between testing, documentation and refactoring.
- Untangling messy code.
- Some social aspects of refactoring.
- The refactoring tools available in Perl.
- Lots and lots of my mistakes and experiences.

Testing and Code Coverage

Paul Johnson <paul@pjcj.net>

Abstract

Hopefully, most of us are aware of what testing is and how it relates to the software development process. It is not my intention to provide an overview of how to test Perl code, but rather to focus on how using code coverage can help in that process.

What is code coverage?

Simply put, code coverage is a way of ensuring that your tests are actually testing your code. When you run your tests you are presumably checking that you are getting the expected results. Code coverage will tell you how much of your code you exercised by running the test. Your tests may all pass with flying colours, but if you've only tested 50% of your code, how much confidence can you have in it?

There are a number of criteria that can be used to determine how well your tests exercise your code. The most simple is statement coverage, which simply tells you whether you exercised the statements in your code. We will examine statement coverage along with some other coverage criteria a little later.

When working with code coverage, and when testing in general, it is wise to remember the following quote from Dijkstra, who said:

> Testing never proves the absence of faults, it only shows their presence.

Using code coverage is a way to try to cover more of the testing problem space so that we come closer to proving the absence of faults, or at least the absence of a certain class of faults.

In particular, code coverage is just one weapon in the software engineer's testing arsenal.

Where does code coverage fit into testing?

Black box testing

Black box testing considers the object being tested to be a black box, that is no knowledge of the internals of the object being tested should be known to the test. The test will simply use the interface provided by the object and will ensure that the outputs are as expected. Should the internals of the object change slightly, or even radically, the test should still pass provided that the interface remains constant.

White box testing

White box testing looks inside the object being tested and uses that knowledge as part of the testing process. So, for example, if it is known that a certain variable is designed to take values within a specific range, a test might check what happens when that variable is given values around its extremes.

Code coverage

Code coverage is a white box testing methodology, that is it requires knowledge of and access to the code itself rather than simply using the interface provided. Code coverage is probably most useful during the module testing phase, though it also has benefit during integration testing and probably at other times, depending on how and what you are testing.

Regression tests are usually black box tests and as such may be unsuitable for use with code coverage. But often, especially in the Perl world, module, integration, regression and any other tests you might perform all use the same test code, just at different times.

How do people normally write tests?

The flippant answer to that question is "they don't". Unfortunately, that answer is the correct one in far too many cases. I'm not going to evangelise about the importance of writing tests; people who don't write tests have no use for code coverage and, in fact, are unable to use code coverage until they do write a test suite.

Sometimes people quickly write small tests to check the major functionality of their code. This is a sanity check and, whilst it has its place, it cannot replace a proper test suite.

"How's the project coming along?"

"It's 99% done, I just need to test it."

If tests are written after development is completed there is no way to gauge the quality of the code while it is in development. What happens if the tests show major implementation or design flaws? Or even minor ones? It is much better to have tests available to be run as the code is developed. Some people propose that tests should be written before the implementation. This has a lot of merit in certain situations. For the purposes of code coverage, it is often better to write a test while the code it is testing is being developed, or at least to write more tests at that time.

No matter how well you write your code, you will get a bug report sooner or later, even if it's only from yourself. That's a good time to write a test. You can use it to reproduce the bug, make sure you've fixed it and, when the test is added to the test suite, make sure it doesn't come back again.

Code coverage metrics

A number of different metrics are used determine how well exercised the code is. I'll describe some of the most common metrics here. Most of the metrics have slight variations and synonyms which can make things a little more confusing than they need to be. While I'm describing each metric I'll also show what class of errors it can be used to detect.

Statement coverage

Statement coverage is the most basic form of code coverage. A statement is covered if it is executed. Note that a statement does not necessarily correspond to a line of code. Multiple statements on a single line can confuse issues - the reporting if nothing else.

Where there are sequences of statements without branches it is not necessary to count the execution of every statement, just one will suffice, but people often like the count of every line to be reported anyway, especially in summary statistics.

This type of coverage is relatively weak in that even with 100% statement coverage there may still be serious problems in a program which could be discovered through the use of other metrics. Even so, the first time that statement coverage is used in any reasonably sized development effort it is very likely to show up some bugs.

It can be quite difficult to achieve 100% statement coverage. There may be sections of code designed to deal with error conditions, or rarely occurring events such as a signal received during a certain section of code. There may also be code that should never be executed:

```
if ($param > 20)
{
    die "This should never happen!";
}
```

It can be quite difficult to achieve 100% statement coverage. There may be sections of code designed to deal with error conditions, or rarely occurring events such as a signal received during a certain section of code. There may also be code that should never be executed:

It can be useful to mark such code in some way and flag an error if it is executed.

Statement coverage, or something very similar, can also be called statement execution, line, block, basic block or segment coverage.

Branch coverage

The goal of branch coverage is to ensure that whenever a program can jump, it jumps to all possible destinations. The most simple example is a complete if statement:

```
if ($x)
{
    print "a";
}
else
{
    print "b";
}
```

Full coverage is only achieved here only if $x is true on one occasion and false on another.

Achieving full branch coverage will protect against errors in which some requirements are not met in a certain branch. For example:

```
if ($x)
{
    $h = { a => 1 }
}
else
{
    $h = 0;
}
print $h->{a};
```

This code will fail if $x is false (and you are using strict refs).

In such a simple example statement coverage is as powerful, but branch coverage should also allow for the case where the else part is missing, and in languages which support the construct, switch statements should be catered for:

```
$h = 0;
if ($x)
{
    $h = { a => 1 }
}
print $h->{a};
```

100% branch coverage implies 100% statement coverage.

Branch coverage is also called decision, arc or all edges coverage.

Path coverage

There are classes of errors which branch coverage cannot detect, such as:

```
$h = 0;
if ($x)
{
    $h = { a => 1 };
}
if ($y)
{
    print $h->{a};
}
```

100% branch coverage can be achieved by setting S<($x, $y)> to S<(1, 1)> and then to S<(0, 0)>. But if we have S<(0, 1)> then things go bang.

The purpose of path coverage is to ensure that all paths through the program are taken. In any reasonably sized program there will be an enormous number of paths through the program and so in practice the paths can be limited to those within a single subroutine, if the subroutine is not too big, or simply to two consecutive branches.

In the above example there are four paths which correspond to the truth table for $x and $y. To achieve 100% path coverage they must all be taken. Note that missing elses count as paths.

In some cases it may be impossible to achieve 100% path coverage:

```
a if $x;
b;
```

```
  c if $x;
```

50% path coverage is the best you can get here. Ideally, the code coverage tool you are using will recognise this and not complain about it, but unfortunately we do not live in an ideal world. And anyway, solving this problem in the general case requires a solution to the halting problem, and I couldn't find a module on CPAN for that.

Loops also contribute to paths, and pose their own problems which I'll ignore for now.

100% path coverage implies 100% branch coverage.

Path coverage and some of its close cousins are also known as predicate, basis path and LCSAJ (Linear Code Sequence And Jump) coverage.

Condition coverage

When a boolean expression is evaluated it can be useful to ensure that all the terms in the expression are exercised. For example:

```
  a if $x || $y;
```

To achieve full condition coverage, this expression should be evaluated with $x and $y set to each of the four combinations of values they can take.

In Perl, as is common in many software programming languages, most boolean operators are short-circuiting operators. This means that the second term will not be evaluated if the value of the first term has already determined the value of the whole expression. For example, when using the `||` operator the second term is never evaluated if the first evaluates to true. This means that for full condition coverage there are only three combinations of values to cover instead of four.

Condition coverage gets complicated, and difficult to achieve, as the expression gets complicated. For this reason there are a number of different ways of reporting condition coverage which try to ensure that the most important combinations are covered without worrying about less important combinations.

Expressions which are not part of a branching construct should also be covered:

```
  $z = $x || $y;
```

Condition coverage is also known as expression, condition-decision and multiple decision coverage.

Time coverage

OK, this isn't really code coverage at all, it's profiling of a sort, but while we're seeing what code gets exercised, why not just see how long it takes for it to be exercised? Maybe it will show up some problems with the algorithm being used, or something.

Documentation coverage

No, this isn't really code coverage either, but documentation is important, right? So let's try and remind people to write some, at least something about each function in the public API, anyway.

How to use code coverage?

So we are all enthused about using code coverage to help ensure that our software is well tested. How do things work in practice?

Let's assume that you have successfully run your test suite with your code coverage tool. (I know that is a very big assumption, but anything else is beyond the scope.... You examine the output from your tool and it tells you that you have achieved 86% statement coverage. Not bad, but hardly stellar. You look at a more detailed report and notice that a couple of methods have never been called. You have two options: write tests for the methods, or decide they are not needed and delete them. Well, I suppose you've always got the option to bury your head in the sand, but let's assume that's not an option we'll take. You decide to write some tests which test these methods, your code coverage increases, your test suite is strengthened, and the world is a happy place.

Unfortunately, it's not always quite this easy. For example, you might notice that a certain missing else clause is never exercised, so you decide to write a test to take that branch, but when you run the test the program performs incorrectly. It may be an error in the implementation, or even in the specification. Maybe no one had even considered what should happen for the test you just wrote.

Sometimes it is impossible to write a test to exercise some code. This can also show errors or omissions in the implementation or specification. For example, the following code hasn't been fully thought through:

```
$x = 0;
print "a" if $x;
```

Given that the use of code coverage can find errors even in the specification, it makes sense to use it as early as possible in the development process. It also makes sense to run your tests regularly with code coverage turned on. It is probably a good idea to have a cover target in your makefile, or to do something similar so that it is easy to get code coverage data.

Here's a suggested way to use code coverage as part of the development process.

- Decide you need a new program / module / method / subroutine / hack.
- Write tests and documentation.
- Write the code.
- Run the tests.
- Check the coverage.
- Add tests or refine the design and code until coverage is satisfactory.

This may or may not be more fun than your current method, depending on your current method, your definition of fun and how much you enjoy bug reports and maintenance. It's also something of an ideal, and we have already established that we don't live in an ideal world. However, simply striving for the ideal, even though we may not actually reach it, will bring its own reward.

What coverage metrics to use and what percentages to aim for?

It's usually a good idea to start with the most simple metrics and move on to the more powerful ones later. In practise this means starting with statement coverage. Then, when you are happy with the coverage you have achieved there, move on to branch coverage, then to condition and path coverage.

Whilst it would be nice to be able to achieve 100% coverage for all the criteria, that is probably not a sensible goal for all but the smallest of projects. So what is a sensible goal? Well, it depends. It depends on the goals of project you are working on. It depends on the cost of failure. It depends on what the software or hardware will be used for and by how many people. It depends on how late into the project you start using code coverage. It depends on the priorities of the people you work with or for. It depends on the way the software was designed and written. It depends on the code coverage tool you are able to use. It just depends.

But, in general, you'll want statement coverage to be way up there. For each statement that is uncovered, you'll want to understand why. If it is a statement that should never be executed then your coverage tool may provide a way to flag that, and even to report if it is ever executed. That way, statement coverage can approach 100%, but in any case you'll probably want to be aiming for 95% at least.

Branch coverage is unlikely to be as high as statement coverage. You might be satisfied with 90% branch coverage, especially if you understand what happened to the other 10%. Path coverage will probably be lower still, but this figure will almost certainly be at the mercy of what, if anything, your coverage tool defines a path to be. The value of condition coverage you achieve will also be dependent on how this metric is defined by your coverage tool. It will also, to a large part, depend on the complexity of the expressions you are using.

When using code coverage, as with every other part of the software engineering process, it is important to be pragmatic. Rightly or wrongly, when I write software at home I have different aims to when I program at work. I even have different aims in the different projects I work on. This applies to the use of code coverage as much as it applies to anything else.

Danger, Will Robinson! Danger!

So, what's the downside? Code coverage isn't all that new, why isn't everybody routinely using it everywhere? Well, some of the reasons are the same as for why people aren't using a host of good ideas, tools and methodologies in the software development process. But let's look at problems specific to code coverage.

To start with, a lot of people just don't know about code coverage. Even people who really should know about it may not. And even if people do know about it, they may not fully understand it, or may not see the benefits it provides. So the first hurdle may be to get agreement, if not enthusiasm, from all concerned for the use of code coverage.

Then you have to find a code coverage tool. Unfortunately, this is not always as easily as it sounds, even if you have a corporate wad to wave around. Good code coverage tools seem to be few and far between, even for relatively common programming languages and platforms.

Once you have your coverage tool, you need to learn how to use it. You need to understand its capabilities and limitations. Of course, this is no different to the use of any other relatively complex tool. That done, you will need to run your test suite using the coverage tool. This might be where things start to get interesting. If you are lucky everything will just work, and you'll get a nice report at the end. If you are unlucky you may hit a host of problems.

Because code coverage is a white box testing technique, it is necessary for the coverage tool to look at and understand your source code. It is possible that some of your code may not be recognised by, or may not work correctly with the coverage tool. Different tools will use different methods for calculating the code coverage. One method is to instrument your source code by adding extra code to it, and then executing the program containing this extra code. It is possible that this will produce different results to your original code. Of course, these are bugs or at least deficiencies in the coverage tool, but did I mention that this world is not always ideal?

If the coverage tool does understand and work with your code, you will have to accept an overhead for using code coverage. This is primarily a time overhead, but the calculation and storage of code coverage data will obviously require a certain amount of resources too. The overhead will vary between tools, based on what coverage criteria they are checking, the way they calculate the coverage and how efficient they are. The time overhead will probably be anywhere between 1.2 times for an efficient tool checking only statement coverage to 20 times or more for a less efficient tool checking many coverage criteria. This might not be too important if it only takes a minute to run your test suite, but if it takes a week then there's a chance that the coverage data won't be available until after the release.

So now you have run your test suite with your coverage tool and you have a nice report telling you which parts of your code weren't exercised. If ignorance is bliss, what does that mean for your coverage report? Improving code coverage can be a major undertaking, but then so can any serious drive to improve quality.

When you have improved your tests and are satisfied with the level of code coverage you are achieving, remember that the quality of your product is not guaranteed, merely improved. Do not rely exclusively on code coverage as a measure of product quality.

What does code coverage not do?

Your code coverage tool will have various limitations in the coverage it performs. One would hope that statement and branch coverage would be almost universally available and consistent, but a number of coverage tools handle only statement coverage. Condition and path coverage, where available, will almost certainly not provide complete information, especially in the case of path coverage. Other coverage criteria may or may not be catered for.

Data coverage

Consider the following program:

```
  my $input = int shift;
  my @squares = (0, 1, 5, 9);
  if ($input < 4 && $input > -4)
  {
      print $squares[abs $input];
  }
```

I can get full code coverage by running three tests, with the input values -4, 0 and 4 (for example). Of course, that won't help a bit when someone wants to know what two squared is.

Some form of data coverage might come in handy here, checking that you have accessed each of the values in @squares. But then again, maybe you are already doing that, and the values you are checking for come from the same source that was used during implementation. No form of coverage can help you there.

Regular expressions

Or whatever they're called now. They might not be regular and they might not be expressions, but they just might be little languages all of their own that deserve to be tested as much as anything else. Regular expressions have their own version of statements, branches, paths and conditions, and that's before we even start thinking about embedded Perl.

Some code coverage tools

On to the specifics. What can we use to get code coverage information for hacking Perl and perl? And XS, since that sort of falls in the middle.

`Devel::Coverage`

`Devel::Coverage` was written by Randy J Ray. Version 0.1 was released on 1st September 1997 and version 0.2 was released on 17th July 2000. It is described as alpha code, but seems to work reasonably well. I was going to mention something about the code being stable and reference Jarrko's .sig, but Randy popped up on p5p recently promising a new release soon.

`Devel::Coverage` interfaces with the Perl debugger in order to calculate the code coverage. This means that it is restricted to statement coverage only, and it also makes it fairly slow, but it does mean that there are no dependencies outside of the perl core.

It is simple to use:

```
perl -d:Coverage script_name arg1 arg2 ...
```

and coverage data can be seen with:

```
coverperl script_name.cvp
```

`Devel::Coverage` requires perl 5.005, but unfortunately it doesn't work with 5.8.0.

`Devel::Cover`

`Devel::Cover` was written by me. Version 0.01 was released on 9th April 2001 and version 0.14 was released on 5th March 2002, but there should be a new version available by the time you read this. It is also described as alpha code.

Why did I write `Devel::Cover`, when `Devel::Coverage` was already there? Primarily, because I wanted to be able to check other types of coverage than just statement coverage. That required a fundamentally different approach to collecting the coverage data, and the infrastructure for that approach was not available until perl 5.6.1 and perl 5.7.1.

`Devel::Cover` does not use the Perl debugger, but instead it uses a pluggable runops function to gather the coverage data. Perl's runops function is a small function that does little more than loop through all the opcodes that make up your Perl program, running the appropriate functions and moving between the ops as the program dictates. It is possible for a module to replace this function with one of its own, and that is what `Devel::Cover` does. This allows for the tracking of each op as it is executed, and it is here that the coverage data are gathered.

But users don't care, in general, about the ops that perl is using to run their program. And so in a post processing phase, information about the ops is mapped back to reality using the rather wonderful B modules.

This approach may allow for a fairly low overhead, at least with the more basic criteria. It also means that it is necessary to have a compiler available to be able to compile the XS code in the module.

As we go to press, statement, time and documentation coverage are supported. Condition coverage is mostly there and branch and path coverage still need to be implemented. Documentation coverage is just a front end to `Pod::Coverage` by Richard Clamp and Michael Stevens, and is unavailable unless you have that module installed.

`Devel::Cover` is also simple to use:

```
perl -MDevel::Cover script_name arg1 arg2 ...
```

and coverage data can be seen with:

```
cover
```

The module is still a work in progress, but I think that most of the most difficult problems have been solved. Well, except for finding a name which won't cause confusion, anyway.

C code

Perl is written in C. XS code is basically C. It would be nice to be able to get code coverage information for our XS code. It would also be nice to see how well the Perl test suite tests perl. Since perl is written in C, just use your C code coverage tool to build perl and run the test suite. If you then use this version of perl to build a module containing XS code you might, if you are lucky, automatically get coverage information for the XS code. Otherwise you will have to see how to integrate your coverage tool with the module build process.

If you compile perl with gcc version 3 this process is simplified. There is a make target perl.gcov which will build a version of perl able to use the code coverage abilities of gcc.

A perennial problem with coverage tools is the back end, used to display the results. For `Devel::Cover` I decided to try to build a generic back end that could be used not only by `Devel::Cover` but also by any other coverage tool. To that end I have created a generic database format to store the coverage information, and reporting programs which read the database and display the output appropriately. `Devel::Cover` comes with a small program which can translate the gcov output into this database format.

The output from the reporting programs is not brilliant. In particular the HTML output is rather clunky. I've been hoping that someone would come along and write a really nice back end, but so far there's no one to blame but me.

Conclusions

Code coverage can help you lose weight as part of a calorie controlled diet.

Author

Paul Johnson - `<paul@pjcj.net>` - http://www.pjcj.net

I am a professional software engineer who has built commercial code coverage tools for Hardware Description Languages. I have been hacking Perl almost since the beginning, both professionally and for fun, and if I'm lucky, both at the same time. I am interested in the production of high quality software and the processes which facilitate that. I am currently enjoying living and working in Zürich.

Security in Perl Scripts

Jesus Alejandro Juárez Robles, Universidad Nacional Autónoma de México <alex@campus.iztacala.unam.mx>

Gunnar Eyal Wolf Iszaevich, Universidad Nacional Autónoma de México <gwolf@campus.iztacala.unam.mx>

Abstract

Perl is an all-purpose, easily extensible programming language that has become quite popular in the last years. Both the language's specification and its implementation are free, which has led to Perl being used not only as an independent language, but as a glue language and an embedded language as well. Perl has important characteristics that help us create safe code, but it has also many subtleties that, if we are unaware of them, can easily act against us.

Good things about Perl

Regular Perl programmers speak wonders about many aspects of this language: We like how it can be written, with flexible sintactical rules modeled after natural languages. We like the richness of the language. We like a real lot of things about this language - However, which of them are relevant security-wise? Mainly:

Non-typified variables

In Perl, we do not have to worry about declaring variables as having a specific type or magnitude. Perl will automatically take care of using the right size and type for our data, and converting it if necessary to any other type.

The same goes for arrays and hashes - When we declare an array, we do not need to state how many elements will it have, and we do not need to worry about the data types of each of them. Hashes and arrays are of variable length, Perl will not make you spend memory for yet-unused locations. Finally, while multidimensional arrays are not really natively supported in Perl, we can seamlessly and painlessly emulate them using references.

Automatic memory management

Many languages require the programmer to manually manage dynamically allocated memory. In Perl, the memory is automatically assign when it is needed, and the language's own *garbage collection* mechanisms will claim back any space which is no longer referenced (and therefore used). This means we don't have to worry about the feared *buffer overflows*, array sizes, data structures' complexity and other details. This not only makes the development time much shorter, but also helps avoid human mistakes, which are so common in tedious, repetitive processes.

High extensibility

Perl is a tremendously extensive language. There are modules already written allowing us to do practically everything, and, without bonding us to use programming paradigms which are not always practical (as Java does, making *everything* into an object, which sometimes is quite awkward and artificial), favors known and trusted code reutilization.

One of Perl's main strengths is the CPAN (*Comprehensive Perl Archive Network*), a very large repository of Perl modules covering practically every area of development. It is a very good idea to periodically check the CPAN during a project's development cycle, as perhaps we will find the answers to many problems we will run into. This will surely save us long hours of development and debugging.

Quick compilation

Although many people see this as a drawback instead of a good thing, I think this is one of the most important factors to Perl's success.

Although many people seem to think that Perl is an interpreted language, this is not true --- Perl is a true compiled language. The compilation, yes, takes place to memory. We can (using the B and O modules, and the perlcc program) compile in different ways, producing binary files, although this is not really faster than compiling our program at runtime, thanks to the highly optimized Perl compiler.

Quick compilation saves a lot of time when programming, even more while debugging. I have participated in projects rounding tens of thousands of lines (plus the included modules), and compilation time at startup was only a couple of seconds. We were able to test even our smallest changes without long delays. I cannot imagine how would we have been slowed down were we to do this project in a language such as C or Java.

Bad things about Perl

Even though I like talking about Perl (and I like even more using it), not all is perfect in this great language. There are many dangerous points which we must keep in mind. Some of them are:

A lot of simplicity and power --- Maybe too much

Perl allows us to easily interact with the operating system, and the programmer may forget to check for the results of its actions, or to correctly validate its arguments. This can lead us to very severe security problems.

Non-prototyped functions

Not typifying functions is a great quality in Perl --- As variables do are non typified, a single function can behave differently according to its arguments, to its context, and basically, to whatever the programmer thinks fit. However, although this gives great flexibility, it has two important drawbacks:

- The programmer must provide the intelligence to ensure the function was invoked with the right parameters in the right format
- makes the code harder to read and mantain

Note that there *is* a way to use prototyped functions in modern versions of Perl (see `Prototypes` in `perldoc perlfunc`), although its syntax is somewhat awkward, and is not widely used. Quoting from this document:

> Some folks would prefer full alphanumeric prototypes. Alphanumerics have been intentionally left out of prototypes for the express purpose of someday in the future adding named, formal parameters. The current mechanism's main goal is to let module writers provide better diagnostics for module users. Larry feels the notation quite understandable to Perl programmers, and that it will not intrude greatly upon the meat of the module, nor make it harder to read. The line noise is visually encapsulated into a small pill that's easy to swallow.

Objects are essentially just a patch

Starting with Perl 5 we can truly do object-oriented programming in Perl. It is, however, easy to notice that this implementation is essentialy a patch, although quite a nice one, and not a nice and clean implementation of objects. Object oriented programming in Perl usually involves many steps that we can skip in other languages (or we can even not think about them), and become that repetitive, tedious task that usually Perl helps us to avoid.

Types of Perl

Being faithful to the TIMTOWTDI philosophy (There Is More Than One Way To Do It, one of Perl's main lemmas), there are many ways for using Perl. The most common ones are:

Traditional Perl

The Perl binary is called together with the program, in order to compile and execute it. This can be made using the *shebang* syntax, giving Perl the scriptname as an argument, giving Perl the whole script as an argument after a `-e`, sending the whole script via STDIN to Perl, and probably many other ways.

Perl's compiler is highly efficient. However, compilation time does cost computer resources. If a program is to be executed frequently, compilation time can become very important.

Practically all Perl modules have been written with traditional Perl in mind, and will probably work better with it.

Perl as a module or part of another application

Perl was originally concieved as a glue language, made to work together with other applications and languages, and has therefore been converted to modules for different applications and to an embedded language. Probably the best example for this is mod_perl, which embeds Perl in the popular Apache web server.

mod_perl is an answer to the problem stated above. A copy of Perl will reside in the Apache server process. Apache, of course, will have a substantially bigger memory footprint than if it would not include Perl, and takes some extra milliseconds to initialize it. However, if a program is requested repeatedly (i.e., a frequently used CGI script), its compiled binary image will be ready in memory, and execution times will be much lower.

There are many ways of using mod_perl. Probably the most frequently used one (although it cannot use important capabilities of mod_perl) is through the Apache::Register module, which allows us to run, practically unmodified, the CGIs written in traditional style.

On the other hand, we have Mason, ePerl and other similar alternatives, allowing us -in the purest PHP and ASP style- to embed our code in HTML.

The best way, however, to take most advantage of mod_perl is to program directly Apache modules. This will give us the power to take part on each stage the server passes, starting with Apache's initialization and configuration, and on each stage of the request's handling cycle. The whole Apache API will be at our disposal if we use mod_perl. There are many things that can only be made in Perl or in C - And most of you will agree that it is usually more convenient to do them with Perl.

Languages comparable with Perl

It is not my intention to start a holy war on programming languages, but I do think that one of the best ways to define the factors that make a programming language unique is to compare it to others. Of course, comparing to each programming language would be tedious and ridiculous. I will compare it, then, with the ones it usually competes with.

Shell

Perl is a wonderful language that can help us with our system administration tasks. In this area, of course, the logical alternative would be using the different Unix shells (Bourne, Korn, C-Shell and their derivatives). This languages qualify perfectly as general-purpose programming languages, and are powerful enough to carry out almost any simple task. However, when we reach intermediate sized problems, we will quickly realize the impossibility to code them cleanly: The variables and functions are global, we do not have even the most basic data structures (and it is quite complex to implement them), and it is very difficult to mantain a modular programming style.

PHP

From its very beginning, PHP was meant to help easily design interactive Web pages (Perl became a top choice for interactive web pages practically since they were first concieved), although there are already ways to use PHP for administration scripts and other many roles. PHP clearly exhibits its heritage - Since its beginning it was meant to make better many aspects of Perl, allowing Web designers to easily do basic programming.

PHP's syntax is very similar to Perl's. Perl users will always find something lacking in PHP. PHP will, unlike

Perl, almost always appear scattered among HTML code - Perl users often frown at this practice, as we find it to be dirty.

PHP is a purely interpreted language, while Perl compiles the whole program into memory. This often makes PHP debugging slower and more complicated.

Typically, executing a PHP program is quicker than calling a CGI script, as PHP's interpreter is embedded in the Web server, and does not need to be loaded each time the program is invoked. This trend is clearly reverted if we use mod_perl with Apache, which besides boosting execution speed allows us to reach a much greater functionality and control of the Web server.

Although PHP is quickly maturing into a serious programming language, its design shows its much humbler beginnings. Many parts of the language do not have a coherent syntax, and it's not impossible to find which parts of the language were designed by different people.

Python

Probably Python is the language with which Perl more directly "competes" with. Python shares many of Perl's characteristics, but it does have a fundamental design difference: Although one of Perl's most often quoted philosophical lemmas is TIMTOWTDI (There Is More Than One Way To Do It), Guido van Rossum (Python's creator) prefers to say that there should be only one way to do it.

Python emphasizes on code legibility, using much clearer and cleaner constructs, and often avoiding the creation of unmantainable code. Many Perl fans complain about Python's lack of naturality - Perl was not designed by a computer scientist, but rather by a linguist, Larry Wall, who modeled its creation to be as close as possible to a natural language.

Java

Java is a poorly understood concept, created at Sun Microsystems. Java's goal is to have a universally portable system, compiling source code to an intermediate form called *bytecode*, which is later compiled at runtime in the destination computer using a *Java Virtual Machine* (JVM).

Java was concieved to allow for execution of remote, unchecked code (often inside a Web browser) as safely as possible. It includes a *sandbox*, which avoids an untrusted program downloaded from the network from carrying certain actions which can jeopardize security.

In Java, everything we do must strictly follow the object oriented programming paradigm. Although it is true that OOP can be very useful for certain projects, making it mandatory for any application makes Java programming slow and tedious.

Java is unfortunately too slow for medium or large applications, and the different versions of the language in widespread use make the promise of "write once, run everywhere" largely a myth, as they are not completely compatible between each other.

JavaScript

This language's name is quite misleading - JavaScript has nothing to do with Java. JavaScript is a specific purpose language, which runs as part of a Web browser. JavaScript is strictly for client-side programming, and we typically use it to validate form input before sending it to the server to be processed, to allow for dynamic HTML pages, and similar tasks.

JavaScript unfortunately also suffers from having a large amount of not completely compatible, widespread implementations. Even though the language's specification is quite clear and -as JavaScript fans say- is quite elegant, using it is often very problematic. Many Web programmers prefer not using JavaScript's advantages in order not to be victims of its incompatibilities.

It is very common to find CGI scripts in different languages generating JavaScript code to be run at client side.

Avoiding insecure programming practices: the `strict` pragma

Pragmas are directives targeted at the Perl compiler, requesting it to act in a specific way for certain part of the code, to modify what it accepts as valid code, or how it will carry specific operations. Throughout this talk we will talk about various different pragmas, but I do think that `strict` is not only among the most useful and important pragmas available in Perl - It is also the one that better illustrates what pragmas are.

An important note about pragmas: If our program spans various files, being divided in modules or libraries, activating a pragma in one of them does not automatically activate it in the other ones. It is a very good practice to start all of our modules or libraries activating the same pragmas, so that the compiler will show a consistent behavior in all of the code.

Introduction to `strict`

`strict`'s purpose is to require us not to fall in insecure programming practices, which, although very useful sometimes, often jeopardize our code. Perl usually accepts certain ways of using variables, subroutines or references that are not really secure - and `strict` will not allow us to use them, aborting at compile time if it detects we are doing something wrong.

Yes, it is often awkward having the compiler complain and not letting us program as we like, but it is fundamental to use `strict` in any project we will use more than just a couple of times, as bad practices will surely come back at us and bite us.

2.1.1 Activating/deactivating `strict`

To ask the compiler to turn on the `strict` checks from the current point in the program on, we tell it to

```
use strict;
```

and if we want to deactivate this behavior from a certain point on, we ask the compiler:

```
no strict;
```

We can specify that we want to activate or deactivate just part of `strict`'s functionality this way:

```
use strict 'vars';
use strict 'subs';
use strict 'refs';
```

If we don't specify a mode, it implies we are referring to all of them.

For further documentation on this pragma, read `perldoc strict`.

`strict 'vars'`: Variable scoping

This mode will make Perl generate a compile time error if we try to use a variable which has not been declared with `use vars`, `my` or `our`, or called with its full name (including its namespace, i.e. `$main::var` instead of `$var`). The use of the `local` scope is disallowed too.

To better understand this, lets briefly take a look at the possible variable scopes in Perl.

It is important to note that using `strict 'vars'` will not break if we use Perl's special global variables (i.e. `$|`, `$_`, `$^W`, etc. Perl's special variables can be used independently of the `strict 'vars'` setting.

Global scope

- Default scope in Perl
- Global variables are available anywhere in the program
- If we want to use a global variable while using `strict`, we can predeclare them this way:

```
use vars qw($var1 @var2 %var3);
```

- The global scope is equivalent to explicitly specify the `main` package. This means that instead of predeclaring the variables as shown above, we can refer to them as `$main::var1`, `@main::var2` and `%main::var3`. We suggest, however, for the sake of clarity, to predeclare the variables instead of using the full names.

Global scope in a package - `our`

- This scope type is available only in Perl 5.6.0 on.
- Variables declared with `our` act as global variables inside a package. The package must have been explicitly mentioned before.

The following expressions are equivalent:

```
$myPackage::variable = 1;
```

and

```
package myPackage
our $variable = 1;
```

Additionally, they are both valid syntaxes when working under `strict 'vars'`.

Local scope

- The variables will be defined inside the actual block and in any block called from it:

```
sub func {
 local $var = 1;
 func2(); # Here $var is defined, so func2 prints 1
 print $var; # We are in the block where we defined $var, so this prints 1.
}
func2(); # Here $var is undefined, so func2 will print nothing
sub func2 {
 print $var;
}
```

- `local` scope will *not* be accepted when working with `strict vars`. In fact, `local` is often regarded as a leftover from Perl 4 - Avoid it, you don't need it.

Lexical variables with `my`

- Variables declared with `my` will only exist inside the block they were created in, and they will be destroyed (and their memory space will be reclaimed) as soon as this block is abandoned.

```
sub func {
 my $var = 1;
 func2(); # func2 cannot reach $var, which is undefined outside this block
 print $var; # We are in the block where we defined $var, so this prints 1.
}
func2(); # Here $var does not exist anymore, so func2 will print nothing
sub func2 {
 print $var;
}
```

- This is the reccommended scope for mostly anything we do in our programs - Try to stick to it :)

Why should we avoid using global variables?

- **Different modules or functions interfering with each other.**

When we program, most of us are quite predictable - How many times have you used the variables `$i`, `$tmp` or `$num`? How many other functions we may use, written by third parties, will have similar variable names? If we use global variables, we will continously have to take care not to interfere with variables used in other parts of our programs.

- **Code gets harder to mantain.**

  Using global variables means we have to document their use globally, not only to avoid de above mentioned problems, but also to allow for future extensibility. Using variables with a more limited scope allows us to document their purpose at the beginning of the function or block they exist in.

- **Better memory administration.**

  Perl's garbage collector is quite a good one - It claims back the space used by our data as soon as our data is no longer needed. In order to know if our data will be ever used again, the compiler can check if the data is still referenced by a variable - If we use global variables, the garbage collector will always find the storage space referenced, as variables will never be automatically destroyed. When using lexically scoped variables, as soon as we leave the block where they were created, the space used by them will be freed and claimed back by the garbage collector.

Behavior of the global scope when using `strict`

If we try to use globally scoped variables without declaring them or giving their whole name, Perl will generate the following errors:

```
use strict;
$var = 1234;
Global symbol "$var" requires explicit package name at programa.pl
line 2.
# We get this error code immediatly, as it gets generated at compile time.
Execution of programa.pl aborted due to compilation errors.
# This error is generated at execution time
```

`strict 'subs'`: Non-explicit subroutines

`strict 'subs'` will require every bareword to be valid function calls, to be enclosed within curly braces or to be at the left hand side of a `=>` operator. This means that the following forms are permitted:

- `$var{key}`
- `(key1 => 'value1', key2 => 'value2')`
- `function` (as long as it is a valid function name and the function has already been declared)

The following will cause a compile time error:

- `$var = value;` (unless `value` is a valid function name)
- `(key1 => value1, key2 => value2)` (unless value1 and value2 ar valid function names)

`strict 'refs'` --- Symbolic references

A quite obscure characteristic of Perl 5 is the use of symbolic references. This is quite a nice and fun concept, but dangerous enough that, as soon as it was introduced, `strict 'refs'` was added to `strict`, avoiding the abuse of this kind of references.

What are symbolic references?

Unlike real references, which we will often find in Perl, symbolic references don not point to the memory address of a variable, but only to a variable's name. The best way for explaining this strange concept is through an example, taken directly from `perldoc perlref`:

```
$name = 'foo';
$$name = 1; # Stores 1 in $foo
${$name} = 2; # Stores 2 in $foo
${$name x 2} = 3; # Stores 3 in $foofoo
$name->[0] = 4; # Stores 4 in $foo[0]
@$name = (); # Empties @foo
&$name(); # Calls the &foo() function
$pack = "THAT";
${"${pack}::$name"} = 5; # Stores 5 in $THAT::foo without the need for an eval!
```

And what is so bad about symbolic references?

- It tends to make our code harder to understand... To prove it, just take a look at the above example (and remember it was taken straight off the manual!)
- Its behavior might seem impredictible. For example, if we have a global `$val` variable, with the `'my-value'` value, and we also have a lexically scoped variable called `$val` with the `'other'` value, and from inside this block we call `$$val`, we will be refering to `$myvalue` and not to `$other`, because even though the internal `$val` is valid, only the external one appears in the symbol table.
- This behavior, in case it is truly required, can be achieved through the use of `eval` - in a *much* clearer way.

Reporting warnings

Perl has the ability to warn the programmer via the standard error (STDERR) that he might be doing something wrong, inviting him to check the code to verify that he did not make an error that can lead to an important failure under certain circumstances.

Perl has been able to report warnings since a long time ago, although its behavior has strongly changed with the introduction of Perl version 5.6.0

Reporting warnings with Perl < 5.6.0

With Perl versions prior to 5.6.0, the behavior of the `warnings` report is defined by a switch specified at runtime, or by a special global variable, according to the following rules:

- By default, the global special variable `$^W` has a false value.
- If we invoke Perl with the `-w` switch, this will make the global special variable `$^W` to have a true value. We can invoke Perl with this switch calling our program as `perl -w program.pl`, or making the first line of our program `#!/usr/bin/perl -w`.
- If we invoke Perl with the `-W` switch, this will make warnings to be always reported, ignoring the value of `$^W`.
- If we invoke Perl with the `-X` switch, this will make warnings never to be reported, ignoring the value of `$^W`.
- Having this rules, warnings will be reported whenever `$^W` has a true value.

Using lexical warnings with Perl >= 5.6.0

Under Perl 5.6, this behavior still works, but a new, more flexible and powerful behavior has been introduced: Defining the pragma `warnings`. Please refer to `perldoc perllexwarn` for further information.

Please remind that, unlike the switches mentioned above, pragmas apply only to the file where they are declared.

Activating warnings report as a pragma allows us to activate or deactivate individually the following categories of warnings (taken from `perldoc perllexwarn`): `chmod`, `closure`, `exiting`, `glob`, `io` (which is further subdivided in `closed`, `exec`, `newline`, `pipe` and `unopened`), `misc`, `numeric`, `once`, `overflow`, `pack`, `portable`, `recursion`, `redefine`, `regexp`, `severe` (subdivided in `debugging`, `inplace`, `internal` and `malloc`), `signal`, `substr`, `syntax` (subdivided in `ambiguous`, `bareword`, `deprecated`, `digit`, `parenthesis`, `precedence`, `printf`, `prototype`, `qw`, `reserved` and `semicolon`), `taint`, `umask`, `uninitialized`, `unpack`, `untie`, `utf8`, `void` and `y2k`.

Each of this categories is activated or deactivated individually. For example, if we want to activate warnings on symbols used only once and about unopened filehandlers being used, and we want to ignore the report of recursion and uninitialized values warnings,

```
use warnings qw(once unopened);
no warnings qw(recursion uninitialized);
```

Additionaly, we can raise the level of this categories to force them become fatal errors. For example, if we want Perl to die instead of just warn when we are redefining a function (this is, when we define two functions with the same name), we can use:

```
use warnings FATAL =>
qw(redefine);
```

If we define warnings report behavior using pragmas, it will have precedence over the behavior of the `$^W` variable or the `-w` switch. However, if we invoke Perl with the `-W` or `-X` switches, they will have precedence.

Handling tainted data

Most programs will not be limited to do internal data processing or data generation - Most programs will take some data set as its input and will generate another data set as its output. This is natural, but can become quite problematic and dangerous, especially when this data can affect the execution process of our program.

For more detailed information on this subject, please check the official doccumentation - `perldoc perlsec`.

What is tainted data?

Perl has a mode where it will avoid any external use we give to tainted data without cleaning it first, this means, it will complain whenever we try to do something that has an effect outside our program's execution space without having first been validated.

Data is considered to be tainted if:

- It comes from direct interaction with the user
- Environment variables
- Parameters received from a Web form

And Perl will refuse to use tainted data for:

- Using this data in any command that invokes a shell process
- Using this data in any command which modifies files or directories
- Using this data in a command that interacts with the process table

Refusing to use them means that if we want to do one of these restricted operations with tainted data, Perl will send a runtime exception, which will kill the running process (unless eval'ed).

Activating tainted data reports

Perl will follow the above described behavior if:

- When we run the program, we specify the `-T` switch, as in `perl -T program.pl`
- The first line of our program is `#!/usr/bin/perl -T`
- If the program is running with either the `SUID` or `SGID` bits on.

Once we enter tainted mode, it will not be possible to leave it - The *whole* execution of our program will be carried out with taint checks.

Detecting tainted data

To detect tainted data, we can use this function (taken from `perldoc perlsec`):

```
sub is_tainted {
    return ! eval { join ('', @_), kill 0;
1;
}
```

We can also use the `Taint` module (available at CPAN) this way:

```
use Taint;
warn "Datos sucios" if tainted ($var1,@var2, $3var3, %var4);
```

Thus avoiding possible fatal exceptions.

Certain values will always be tainted, as they come from the outside world. For example, the execution path (`$ENV{PATH}`) is received from the invoking process, and Perl cannot trust it to be safe. Any external program we execute from within Perl (with `system()`, `exec()`, `qx()`, backticks, etc. will require $ENV{PATH} to be untainted (cleaning) to allow its execution:

```
$data = <STDIN>; # $data contains tainted data, as they come from the user
if ( $data =~ /^([\w\b\d]+)$/ ) { # Accepts only the specified pattern and stores it in main
memory
    $clean_data = $1; # $1 contains the text which matched the regular expression
} else {
    die "I did not expect this condition: $data";
}
```

A warning about unconsciously untainting

After what we just said, this might sound tempting:

```
$data = <STDIN>;
$data =~ /^(.+)$/;
$data = $1;
```

However, accepting any data without checking defeats the use of Tainted checks. It is not impossible for us to want to do something like this, but it is *very* important to double- and triple-check if this is our only way out.

Unconsciously untainting data can be very harmful. Many computer security schemes of all kinds fail because they make the user believe he is completely sure, even if they are just a little help to heighten a bit the overall system security. Tainted checks are a tool for a programmer to ensure he is not forgetting to check something, and circunventing this mechanism would be completely pointless.

Suggestions to remember when using functions

There are many subtleties concerning functions and their use. Among the most important ones:

Invoking functions

If we are not using `strict`, Perl will allow us to invoke functions directly, giving only the function's name without explicitly stating that we are doing a function call. Although this seems often elegant and clean, it can lead to confusions to people reading our code, and can clash with a reserved word in a future Perl version. Functions should preferably be called as `function()` or `function($arg1,$arg2)`, indicating an argument list even if it is empty, making clear we are talking about a function call.

Many people like adding the prefix `&` to the function calls. This is a syntax inherited from old Perl versions and not needed anymore. If you choose to use this syntax, it is also very important to explicitly give the argument list even if it is empty (`&function()`). If instead we call it as `&function`, this function will inherit the argument list (`@_`) for the current function call.

Receiving parameters

It is very common for the first line of a function to be:

```
my ($var1, $var2, $var3) = @_;
```

or

```
my $var1 = shift; my $var2 = shift; my $var3 = shift;
```

This syntax is completely correct. However, if we just assume the function was correctly called, we can end up with undefined parameters, or ignored arguments. It is very advisable to check each of the received parameters, searching for undefined values, incorrect data types (i.e., a variable containing text where we should have a numeric value, or a scalar value where we expect a reference). Having incorrect data can cause erratic and hard to trace behavior.

Note on the reason of having `@_`

Many people ask why Perl uses a default array (`@_`), something not done by any other modern programming language. A simple example will better illustrate this:

In C, we declare a function like this:

```
int funct (int var1, char* var2);
```

In Python:

```
def func (var1, var2):
```

In PHP:

```
function func ($var1, $var2) { (...) }
```

And the list goes on for practically any other language. Why do we have to juggle around in Perl as we discussed on the previous section, manually assigning variables from a default array called `@_`?

Believe it or not, this is done by design. In Perl's first incarnations the lexical variables did not exist. If we were to declare a function together with its variable names, this would impact on the global namespace. Rather than polluting namespaces, the Perl architects decided to give arguments one standard and well defined name: The default array, or `@_`. In fact, if we will code a short function and this will not make our code illegible (and even more if the function will be called often), it is advisable to use the default array directly inside the functions. This way, we will not have to worry about manual creation and assignment of variables, saving some microseconds. Yes, some microseconds may be very little in most cases, but it can become quite noticeable in some circumstances.

Handling internal variables

We have talked this over a couple of times already. It is very important to make all of our variables (or at least, as many as possible) belong to the lexical scope (with `my`). This will avoid data in our function affecting other functions, and it will optimize memory usage.

Being careful when using references - `ref`

When we use references to pass arguments to or return values from a function, it is very important to check they are effectively references, and that they refer to the right data type. It is invariably better to return from the function with an error message and allow the invoking program or function to handle this condition than just raising a run-time error, leading to aborting the execution, or maybe even worse, handling incorrect data.

To timely detect these errors, we can recur to the `ref` builtin function this way:

$var	**ref($var)**
\\\$something	'REF'
\$data	'SCALAR'
\@arr or [1,2,3]	'ARRAY'
\%hash or {a=>1,b=>2}	'HASH'
\&func	'CODE'
*other	'GLOB'
$non_ref	'' (empty string, not undef)

`GLOB` references

A special reference type is a `GLOB` reference - It is not a reference to a scalar, an array, a hash or code, but rather a reference to everything.

Namespaces don't only manage the data types just mentioned. They also manage filehandles, which do not have a identifying prefix,a nd are not easy to pass between functions. If we want, for example, to pass a filehandle aas an argument, the traditional way is to do it via a global ref: `func(*STDOUT)`. This makes `func`'s first argument an object of `IO::Handle` type.

Yes, back in their day `GLOB`s were a needed alternative, and even an elegant way of doing something the language needed. However, nowadays the best use of `GLOB`s is to confuse fellow programmers. If you want to pass filehandlers between functions, it's often better to use the native object oriented implementation of `open`, through `IO::Handle` (or one of its subclasses, as `IO::File` or `IO::Socket`):

```
use IO::File;
if (need_open()) {
    my $handle = new IO::File;
    if ($handle->open('<filename')) {
        use_file($handle);
    } else {
        die "Error opening the file: ", $!;
    }
}
```

Here, `$handle` has all the attributes of a lexical variable: We can pass it to the `use_file` function as a parameter, it automatically disappears when leaving the block where it is declared (cleanly closing it, of course, thanks to the object's destructor), etc.

Returning results

When we leave a function, either after a successful or an unsuccessful operation, we should `return` a result, as simple and coherent as possible.

Every function should have a consistent way of expressing success or failure, and the function call should handle this result accordingly.

When we successfully complete a function, we should find the simplest way of returning the processed data - For example, sometimes it will be much simpler to process multiple results of a function call if they are all grouped in a single reference to an array, or better yet, to a logical structure reflecting their relationships, instead of sending them back as a collection of unrelated scalar values.

Suggestions to remember when using objects

Perl 5 introduced many important advantages over Perl 4. Maybe the most important of them is the support for object oriented programming. However, as we will soon see, Perl's OOP implementation is neither complete nor clean, if we follow the traditional, formal criteria for defining objects. Using objects correctly as they are implemented, however, can be very convenient, once the particularities are taken care of.

To work optimally with objects, please refer to `perldoc perlobj`, `perldoc perltoot` and `perldoc perlbot`.

Living with a patchy object implementation

Perl's OOP syntax often forces the programmer to write more than what he would in other languages, and for non-expert eyes, the style is not very clear, often even confusing. For example, we refer to an object's attribute by using `$obj->{attr}`, but we refer to an object's method with `$obj->meth()`. A common error is to try `$obj->attr`, which Perl will translate to a method call. In case the `attr` method exists, it will be called and its return value will be given to the caller, probably leading to unexpected behavior. If the method does not exist, this will lead to a runtime error, aborting our program execution.

In traditional object oriented languages we are used to having private and public attributes and methods. Perl invites us not to think that way. Perl invites us to make everything public. Quoting Larry Wall, I will let you enter my house, and I trust you not to steal my belongings because I trust in you, and not because I have a security surveillance system at home. Yes, there are ways of implementing private methods and attributes, but they look more like black magic and contorsionism than calls.

Checking for required and optional attributes

When we create an object, it is important to check that we have all the required parameters, and that we were not invoked with a parameter we don't know how to handle. To do so, I suggest to include in your constructor methods something like this:

```
my @needed = qw(color size type);
my %temp = ();
my %valid = ();
map {$temp{$_} = $valid{$_} = 1} (keys %$self);
$valid{$_} = 1 foreach (qw(texture temperature));
foreach (@needed) {
    if (defined $temp{$_}) {
        delete $temp{$_};
    } else {
        die "I need $_ to create the object!";
    }
}
if (my @tmp2 = keys(%temp)) {
    die "Unknown elements at object initialization: @tmp2";
}
```

Suggestions to remember when using modules and libraries

A large project can be much better managed if it is built using modules, splitting a large program into many separate and function-specific files. This has the additional advantage of making much easier code reutilization for other projects.

Besides modularizing our code, in Perl we will frequently use modules built by other people, as we have a large collection of modules, voluntary contributions from various Perl developers worldwide, organized in the CPAN.

In Perl we often talk about modules and libraries. They are very similar, but with one subtle yet important difference: A library is only a collection of functions, whereas a module is encapsulated in its own namespace, and is often design with object orientation in mind. Of course, as we will soon see, we can call a module as if it were a library or the other way around.

Packages and namespaces

It is a very common and advisable practice to use separate namespaces when we work with modules. This will help us avoid global functions and variables existing in one of our modules collide with others, defined with the same name, in the main program or in any other module.

The default namespace is `main`, and by default, all symbols (functions, variables and filehandles) in Perl are prepended with their namespace's name behind the scenes - The real name for `$var`, `&func` or `FILE` is `$main::var`, `&main::func` and `main::FILE`. If we specify a null namespace (i.e. `$::var`), Perl will translate it to an explicit call to `$main::var`.

With the `package` command, we can change the namespace where we are working, as shown here:

```
$var = 'value';
```

```
$Other::var = 250;
print $var; # 'value'
package Otro;
print $var; # 250
print $main::var; # will always be 'value'
print $Otro::var; # will always be 250
print $::var; # will always be 'value'
```

Lexical variables (those declared with `my`) do not belong to any namespace, and are not affected by packages.

Methods for the inclusion of modules and libraries

There are three methods for including code files: `do`, `require` and `use`. Files included with the first two are libraries, and files included with `use` are modules.

do

Its effect is exactly including the file's text in the exact point in execution it is called at. Quoting from the definition of `do` in `perldoc perlfunc`, `do 'file.pl'` is equivalent (although more efficient) to doing ``scalar eval `cat file.pl` ``. Each time we find a `do`, the file is evaluated again, so it should not be used inside loops.

The value returned from a file's inclusion is the one of the last expression evaluated in the file. If the file was not successfully included, the return value will be `undef`, and the special `$!` variable will have the error message. If the file was successfully read but could not be compiled, the return value will be `undef` and the error will be received in the special `$@` variable.

`do` is normally used for reading configuration files. Again, from `perldoc perlfunc`:

```
unless ($return = do '/usr/local/etc/myconf') {
    warn "couldn't parse $file: $@" if $@;
    warn "couldn't do $file: $!" unless defined $return;
    warn "couldn't run $file" unless $return;
}
```

require

`require` reaches a bit beyond what `do` does. The main differences are:

- The specified file must exist. If it does not exist, an runtime error is generatied, and Perl aborts execution.
- The last evaluated expression in the file must be true - If it is not so, the execution of the program will terminate with the message *library* `did not return a true value`. Because of this, it is very common to have `1;` as the last instruction of a library file.
- The file will be evaluated and executed only once. When we include a file with `require`, Perl checks first if its name exists already in the `%INC` hash, and if it does, skips the inclussion. When Perl finishes the inclusion, it adds the library's name to %INC.
- If the argument we give to `require` is a bareword, Perl will look for the source file in every directory specified in every directory listed in the `@INC` array, assuming a `.pm` extension, substituting the `::` characters for `/`. If we give `require` a string or a variable as its argument, although Perl does look in `@INC`, this substitutions will not be carried out.
- If the argument we give to `require` is numeric, the execution will be aborted if we are running under a Perl version lesser than the one specified.

use

This inclusion method was born with Perl 5, and was thought to be used with modules and objects. When we include a file with `use`, additional to `require`'s behavior:

- The inclusion is done at compile time, not at execution time. The code included with `use`, even if the `use` statement appears in the last line of the program, will be available from the very beginning.
- `use`'s argument must be a bareword.

- After the module's name, you can specify a list of functions to be imported into the invoker's namespace. You can request specifically not to modify the invoker's namespace by providing an empty list.
- Given `use`'s syntactic flexibility, it is also used for the activation of pragmas.
- If a module has the `unimport` function, we can free our namespace from this module's functionality using `no` *module* (*list*).

For further details, refer to `use`'s section in `perldoc perlfunc`, as well as to object specific documentation in `perldoc perlmod`.

Modules, pragmas, namespaces and the compiler's behavior

If we define a pragma in any of our modules or libraries (or in the central program), the behavior this pragma defines will not be inherited by any other file we use. Check on the documentation for each pragma to see if its effects are lexical (they apply to the whole block where they appear) or for the whole file.

Restricted compartments: The `Safe` module

If we don't trust the code for a certain program, module or library, we might want to add security restrictions to it when it is run. The module `Safe` was written for this purpose, and is part of Perl's default distribution. In this talk we will only skim it. For further details, refer to `perldoc Safe`.

`Safe`'s basic syntax is very simple - For creating a new restricted compartment, we use:

```
use Safe;
$compart = new Safe;
```

Restricted namespace

We can define a namespace to which this code will be restricted to, from which it will be denied interaction with any simbol located outside it. All data interchange between the restricted compartment and the rest of the program must be done using the default variables (`$_`, `@_` and `%_`), as well as the symbols explicitly declared when we create the restricted compartment.

The default namespace is `Safe::Root0` for the first compartment, `Safe::Root1` for the second, etc. To ask a compartment which namespace is it running in, we can use the `root` method.

We can specify the namespace we want the compartment to use by calling it this way:

```
my $comp = new Safe 'hidden';
print $comp->root;
```

This will cause the namespace for the compartment to be *hidden*.

Opcodes masks

Another way to control code execution is limiting the operations it will be allowed to carry out. If we use a restricted compartment, we can specify which Opcodes will be valid in its code, aborting execution in case an invalid opcode is requested. The default Opcode mask is `:default`, which allos all the opcodes to be executed.

This topic goes beyond the reach of this talk, for more information please refer to the manual page refering to opcodes (`perldoc Opcode`).

What will `Safe` not protect us from

As any other security tool, `Safe` is a very useful helper, but is far from perfect, and it is very important to remember its limitations. `perldoc Safe` mentions the following cases where `Safe` will not be of great help. Some of them, of course, can be avoided by prohibiting the corresponding Opcodes, but it is very difficult to find every case. Some of them simply cannot be restricted without seriously crippling Perl itself for the compartment, maybe to the point of becoming useless.

- Resource exhaustion, as simple as a `while (1) {}`.
- Code that spies on the system and sends information somewhere else
- Generating signals that affect the program's execution, taking advantage of poorly made signal handlers.

- Operations affecting the whole process, such as `chdir`
- The verification is done at compile time, so a `eval "(...)"` will not suffer from the Opcode restrictions.

The Dark Art of Obfuscation

Thomas Klausner <domm@zsi.at>

Abstract

What obsfucations are, how to create them and how to have fun with them. An online version of this talk can be found at http://domm.zsi.at/talks/obfu_yapc2002/.

What is Obfuscation?

Some possible definitions:

Obfuscation is writing un-understandable code.

The most obvious definition: Code so, that nobody can guess what your code will do without either running it (and still not knowing *how* you did it), or spending some amount of time and brain to figure it out.

Obfuscation is a waste of time.

What some people think - but I hope to show you that this prejudice is not true.

Obfuscation is like inventing crossword puzzles.

Sort of: you spend hours preparing a tricky puzzle for the entertainment of others.

Obfuscation is basic research.

Definitly. Personally, I learnd a lot about Perl while writing my own or decoding (de-obfuscating) other peoples Obfuscations.

Obfuscation is fun.

At least for me.

Obfuscation is job security.

Not.

Obfuscation is Art.

That's the definition I like most.

If you compare Obfuscation to an Art, e.g. poetry, you will notice a lot of similarities:

- Both seem needless at the first glance.
- Both *are* needless in some way, as they do not solve a "real problem"
- Poems usually adhere to some structure like rhyme, meter (e.g. hexameter) or sibil counts (e.g. haiku) - Obfuscations often adhere to similar self-restricting structures (ASCII-Art, fixed output ("JAPHs"), limited function/character set)

Obviously, as there are poems or pieces of art that a large number of people consider "bad" (although this finding may change over time), there are "bad" obfuscations. But I still consider all obfuscations Art.

One defintion of Art, by Spinoza:

> Any human creation which contains an idea other than its utilitarian purpose.

Obfuscation Techniques

Here I will describe some techniques that are usefull when writing obfuscations. Sometimes they are useful even when doing "real" programming.

Using Defaults.

Perl provides a lot of usefull default values in already obfuscated variables (`$*`, `$$`, `$,` ..). Know them. Use them.

Scrambled Data.

"10689781128104" is much harder to read than "japh"

Quoting.

qq(q{quote me: "I didn't do $it"})

Generating Code.

eval! eval!! eval!!!

ASCII-Art.

Funny-looking programs

Pre/Post-In/Decrement.

What's the result of ++$a + $a++ - --$a - $a--

Shift Push Pop.

Tormenting Arrays

Warped Flow.

"Control structures like 'if','for' and 'while' are for whimps" (Damian Conway)

OOO (Object Oriented Obfu).

How to (mis-)use Perl OO-features in Obfuscations

Outwitting strict.

use strict, but still escape it's bonds

Selfreading Code.

Parse Yerself

Busting O::Deparse.

No cheating, please

Multilinguality.

Programms that compile in more than one language

Obfuscation Disciplines

.sig.

Fit into a .sig (signature) file

JAPHs.

Print the string `"just another Perl hacker\n"`.

$a++.

Increment the value of $a by one.

Golf.

Find the shortest (in keystrokes) solution to a problem.

Quines.

A program that generates a copy of its own source text as its complete output. [from the "Jargon File"]

Self-Restricted Obfus.

Use only a small subset of availiable functions/characters/space/whatever (e.g. no '$', no ';', ..)

Command line.

Can be run directly via **perl -e "..."**

The Author

Thomas Klausner is full-time father of 2 kids, half-time Perl Hacker, sort-of DJ, bicyclist, dreadlocked, 26 years old, living in Vienna (Austria) and mildly afraid of morons in power all around the world.

Smoking Bleadperl

H. Merijn Brand <h.m.brand@hccnet.nl>

Abstract

How many hours a programmer will put in his program, and how many bugs he might have solved, and how much of the program is covered with working test code, bugs will probably always be present in the program or script.

The most recent development branch of the Perl code is called bleadperl, and it is patched by many on several parts on a daily basis. For every patch that solves a bug or extends the features of the rich language that Perl offers, other functionality might get busted or slightly broken.

Testing the latest state in as many configurations as possible on as many operating systems as possible with as many development enviroinments as possible is a great source of information to the perl5 porters. This process has been called ``smoke testing''.

Introduction

With people working on various parts of the perl core simultaniously: threading and variable sharing, IO subsystems, reported bugs, formatting, configuration and hints, etc. etc., it is likely that minor changes might break basic features and patches liked and accepted by the GNU gcc compiler are rejected by other - more rigorously ANSI - C compilers.

To catch those, it would be good to have every patch tested on all supported platforms as soon as possible. But since bleading edge perl is a development release, it is likely to be far from stable at any point in time, and thus not to be installed by the faint of heart. Core dumps are likely to occur if a snapshot is released without being tested on a variety of configurations.

`Test-Smoke`

`Test-Smoke` is a package of scripts that enables perl5-porters to receive test reports from users that have spare CPU cycles and some disk space on their `computer(s)` and who are willing to spend a few minutes in installing the package which then will run a number of tests on the most recent bleadperl situation in as many configurations as possible.

The talk will deal with the development of what has evolved to `Test-Smoke`, currently available on CPAN. It will also deal with some differences in compilers, with different OS behaviour, and how new users can participate in this valuable process.
As a side step, some of the parts in the perl source tree are explained briefly but not profound and not to show what the implications might be of what smoke testing detects.

Part VI. Lightnings and other Media

Imaging with `Imager`

Claes Jacobsson <`claes@surfar.nu`>

As we all know, an image says more than thousand words. This is a presentation on which imaging modules are available for the Perl programmer who wishes to visualize information or manipulate images. An indepth guide to the Perl module `Imager`, covering basic drawing operations to more advanced functionallity such as text and filters.

The Simulation and Visualisation of Objects in 3D Space

Greg McCarroll <greg@mccarroll.demon.co.uk>

This talk is about the simulation and visualisation of geometrical objects under their own acceleration in 3D space. It will achieve the visualisation of them via the MesaGL library. The maths involved will be very simple and there will probably be a crude collision detection model.

Send and Receive SMS Messages Using a GSM Modem

Johan Van den Brande <johan@vandenbrande.com>

The author of the GSM::SMS [http://www.tektonica.com/projects/gsmsms] module family introduces the concepts of SMS, Short Message Service. Learn how to send and receive SMS messages using a GSM modem or phone. A short introduction to SMS gateways will also be included. Following packages will be touched: Device::SerialPort, LWP and GSM::SMS.

How to Write Slow Algorithms Quickly in Perl ?

Gabor Szabo <gabor@tracert.com>

Abstract

Though some of us might think so, chomp is not only a Perl function. It is also the name of a NIM-like Combinatorial Game that was unsolved until recently. It has a solution and implementation in Maple and I am writing an implementation in Perl for educational and research purposes.

Introduction

When I went to high-school in the early 1980s in Budapest, Hungary, I used to play a game with a class mate that we called eating chocolate. We actually did not really play it as we knew that there was a winning strategy for the player that moved first but we tried to find a mathematical description for that winning strategy. For that I wrote several programs that would compute the winning positions but we did not have any results.

A few years later I bought a book called "Dienes Professzor Játékai" [DIENES] in Hungarian translation but actually I have looked only at a couple of pages in the book until recently.

Then about a year ago I decided it is time to learn how to create and upload a module to CPAN and as the explanation regarding how to get accepted in PAUSE was rather discouraging I decided I try to play safe and start with a module that probably no one else wants to develop but which can be nice to have on CPAN: `Games::NIM`. I planned to develop the module to play the game and to calculate the winning positions for NIM and later to extend to Chocolate. To my surprise I got the access and uploaded version 0.01 in December 2001 and then it got stuck at that version.

Now when I thought about attending `YAPC::Europe::2002` I decided to renew the work around `Games::NIM` and proposed a talk about complexity in algorithms in connection to that module and another module called `Array::Unique`. When the proposal got accepted I suddenly discovered that I have not much to say about the subject and have to work really hard in order to give you something worthwhile. So I started to work on `Games::NIM` again and read the book of Dienes [DIENES] about games and another very useful one called "Mastering Algorithms with Perl" [ALGORITHMS]. I suddenly discovered that the game I knew as chocolate eating game is actually known as Chomp and it is still basically unsolved. It all sounded very encouraging.

So here I am now, trying to understand what is the status of the research regarding this game and trying to put together a presentation that will not be too embarrassing by presenting trivial things.

So what is Chomp ?

The game starts with an M by N chocolate bar, in which the upper left cube is poisonous. The players take turns by selecting a cube, eating it and all other cubes below and to the right of the selected one. The player who takes the upper left square (either because that is the only one left or because he is fed up with the game and commits 'suicide') will lose the game.

An example.

In case we start with a chocolate bar that is 3x4 like this:

```
x x x x
x x x x
x x x x
```

One of the choices of player Number One is to select the square at 2,3 so the remaining chocolate will look like this:

```
x x x x
```

```
x x
x x
```

Then the second player among other possibilities can select 3,2 and reach the following state:

```
x x x x
x x
x
```

and so on till only the poisonous square remains. The player who moves next will lose the game as he has to eat the poisonous square.

It can be easily proved that at any M and N that is not 1x1the player who moves first can win the game if s/he plays well no matter what the other player does. That is the first mover has a winning strategy. (You can find a proof in Zeilberger's article [TRC]). After playing a bit you can also find the "winning strategy" for cases when N=M that is when the original chocolate is a square and it is also easy to find for N<3 when the chocolate bar has only one or two rows. Proofs for these can also be found in the above article. For all other cases (meaning when both M and N are >2 and are not equal) there is no solution yet[10]. Of course we can always use the naive approach [11] to find all the winning positions.

That's what I did in high school and that's what I did in the first versions of `Games::Chomp`.

What do we know so far ?

Reading Zeilberger's article [TRC] I found out that the mathematical world calls the "winning positions" as P-Positions (Previous player wins) and the "losing position" as N-Positions (Next player wins).

Zeilberger also explains that the number of legal positions for an MxN game is (M+N above N) which is an exponential function of N if M=O(N). So if I want to follow the naive approach and run an exhaustive search for the P-Positions it will take me a rather long time to find all the solutions as N and M are getting large. An exponential complexity is not something we like to see not even with computers . Especially when the algorithm is exponential in both time and memory.

Then he explains about a Maple [MAPLE] package he developed for symbolic computations that can compute P-Positions of the Three-rowed Chomp very quickly for all the cases when the 3rd row is not longer than 115 squares.

Based on the above article Xinyu Sun [XTRC] has improved the Maple package so it can actually compute all the P-Positions for any number of rows and columns in a reasonable time. He also shows a few real formulas that can be used to calculate the P-positions for a few cases of M and N. In another article Sun shows the Sprague-Grundy function of Chomp [SGFC]. If I understood correctly this function is a mapping from every Chomp position to a NIM position based on the theory of Sprague and Grundy that every finite impartial game can be mapped to a NIM game.

The game can be represented in several ways. One of them is the chocolate bar of NxM squares. Another representation is when the two players alternately naming a divisor of a given number n = 2**N * 3**M which may not be multiples of a previously named values.

A game of 4x5 would look like this:

1	2	4	8	16
3	6	12	24	48
9	18	36	72	144
27	54	108	216	432

[XTRC]

At least currently as I am writing this I think that there is no solution for those cases but I still have to try to understand the articles written about the subject. Read on.

The 'naive approach' is to go over all the possible positions between (1,1) and (N,M). Others call this 'brute force' though I have no idea how can these two terms mean the same thing. Of course this cannot be done by a human being without going insane so we leave it for the computers.

Actually with this represantation you can easily extend the game so n can be any positive integer number. This is similar to allowing a chocolate in n dimensions which would be a rather big chocolate I guess and which is quite hard to display on my computer.

So what is left for me and Perl ?

It seems that the big issues with Chomp are already solved (again, if I understand correctly the articles I mentioned) so what is left for me and for Perl in this story?

First of all I have a history with this game so I enjoy going back to it, reading about it and developing a module that will compute the P-positions and play the game.

Secondly, though there is a Maple Package [XTRC] that does this computation and even allows you to play with it but Maple [XTRC] itself isn't free in either meaning of the word and it is not likely that people all over the world will have it installed on their computer.

Developing Chomp in Perl will allow these people as well to play the game.

`Games::Chomp`

The module I have developed so far (version 0.03 when I am writing this) can compute all the P and N Positions and save them in a temporary file so when I run the program next time it will not need to do all the computation again. It is using brute force going over all the possible positions up to the starting position you provide. Version 0.01 was extreamly slow not only becasue of the exponentialness of the number of possible positions but also because my bad implementation. In version 0.02 it already has a little speed improvement compared to version 0.01 as I have added a few simple rules about NxN and 1xN and 2xN that were obvious.

In version 0.03 I added a capabality to represent rectangulars that were already processed without listing all the N-Positions in them that reduced memory consumption and increased speed.

I have added a script that runs benchmarks on the calculation for various NxM pairs. Right now it takes 30 seconds to solve the 3x30 table.

Future work

In the coming weeks till the presentation at `YAPC::Europe::2002` I hope to make several improvements in my own code that will increase the speed without going below the exponential complexity and possibly implement some of the results of Sun [XTRC] that might already take the complexity in the sub-exponential world. I'll also add functions to show the positions in various representatoins and probably a way to play the game at least on the command line.

I see NIM and Chomp and other similar games as educational tools like it is described in the book of Dienes [DIENES] so based on this module one can develop some kind of educational tool that will help children learn about numbers and strategy.

I also have this idea of "Perl on every desktop" but I think this should be another presentation. Anyway it can be useful to further develop my Perl module that will compute the P-positions for arbitrary Chomp tables (and we can even think about more than two dimensions !) and let you play the game. It should come with a web interface so it can be provided as a service and a Perl/Tk interface to encourage people to download Perl and the game and install them on their own computer. This module might also be useful for other people to look at the P-positions in different angles. They might come up with further ideas regarding a formula to compute them quicker. Creating this Perl module might also lead to the development of other similar models in Perl to help solving more difficult mathematical problems.

About the Author

Gábor Szabó gabor@pti.co.il [mailto:gabor@pti.co.il] Teaching Perl in English, Hebrew and Hungarian Developing Perl in Perl.

Perl Training and Development in Israel www.pti.co.il [http://www.pti.co.il/] Maintainer of http://www.perl.org.il/ the Israeli PUG. Member of the Hungarian PUG http://www.perl.org.hu/

Bibliography

[CHOMP] The documentation of the `Games::Chomp` module on CPAN. http://www.cpan.org/modules/by-authors/id/S/SZ/SZABGAB/.

[DIENES] *Dienes Professzor Játékai*. Mûszaki Könyvkiadó. 1989. The manuscripts of Zoltán Dienes were collected by Tamás Varga. Original title: "*Have Fun and Stretch Your Mind*". I could not find a reference to this book on any web site but you can see a (partial ?) list of the works of Zoltán Dienes at http://www.zoltandienes.com/..

[ALGORITHMS] *Mastering Algorithms with Perl*. Jon Orwant. Jarkko Hietaniemi. John Macdonald. O'Reilly. 1999.

[TRC] *Three-Rowed CHOMP*. Doron Zeilberger. Appeared in Adv. Applied Math. v. 26 (2001), 168-179. http://www.math.temple.edu/%7Ezeilberg/mamarim/mamarimhtml/chomp.html.

[XTRC] *Improvements on Chomp*. Xinyu Sun. Appeared in Integers Journal vol. 2 (2002). http://www.math.temple.edu/%7Exysun/chomp/chomp.htm.

[SGFC] *The Sprague-Grundy Function for Chomp*. Xinyu Sun. (In preparation). http://www.math.temple.edu/%7Exysun/chomp/chomp.htm.

[MAPLE] The maple software can be found and purchased at http://www.maplesoft.com/.

Quantum::Superpositions

Damian Conway <damian@conway.org>

This talk explains how to adapt the Perl programming language to quantum computing and vice versa, with applications for prime generation, list membership testing, maxima and minima detection, string comparison, and winning the office football pool, all in constant time without loops or recursion.

Along the way, we'll also touch on the physics of chocolate, parallel programming, motor racing, the Scottish vegan movement, ancient Latin, Mick Jagger's love life, winter sports, the secret of the humble potato, fashion modelling, anagrammatic encryption, modern German, cruelty to animals, and the antics of the 1930's chapter of Copenhagen.pm.

Part VII. Perl Basics

CPANPLUS - Beyond CPAN.pm

Jos Boumans, http://japh.nu amsterdam.pm <kane@cpan.org>

Abstract

CPANPLUS is a new and flexible method of perl module management and installation, using the Comprehensive Perl Archive Network.

About CPANPLUS

CPANPLUS is a new module. It was born in September of 2001, after that year's YAPC::Europe in response to general unhappiness about CPAN.pm. It aims to be a rewrite, and in time a replacement of the current CPAN.pm, but in a completely OO model, adding features unknown to the current CPAN.pm, such as module uninstall, multiple shells and so on. Not in the least, it aims to fix various bugs of CPAN.pm as well as being very modular in its build setup, thus allowing many plug-ins. For example, with a plug-in to the RT bug tracking system, CPANPLUS could send in automatic bug reporting in case a module failed to install.

About the talk

I have given the 90 minute talk about CPANPLUS to Amsterdam.pm and it was very well received. However, it also went into the technical depths of the module and reviewed the history of development, as well as covered the basic knowledge required to develop new applications using CPANPLUS. Although this is very useful information, it's perhaps better suited for a BOF session. In short, I could range the talk from 25 minutes, covering just the bases for the average user, to 90 minutes, covering development, guts, interfaces, features, bugs, direction and so on of the module.

Some more information about the module can be found on the project's homepage at: http://cpanplus.sourceforge.net

Contents

Depending on how much time I am given, the following topics would be discussed:

- Reason for creating CPANPLUS (25)
- Improvements of CPANPLUS vs CPAN.pm (25)
- A short description of the developers of CPANPLUS (45+)
- The user-interface 'Shell' (25)
- Sample usage of 'Shell' (25)
- The script-interface 'Backend' (25+)
- Sample code utilizing 'Backend' (25+)
- The developer-interface 'Internals' (60+)
- The merits of the various interfaces (45+)
- The current developments (45+)
- The future developments (60+)
- How to contribute to the project (60+)

ExtUtils::ModuleMaker

R. Geoffrey Avery, GlaxoSmithKline
<modules@PlatypiVentures.com>

Modules are the basic building block for structured programming in Perl. In this session you will learn how to use the `ExtUtils::ModuleMaker` facility. Learn how to automatically construct all the base files needed by a well-engineered module. Get to know the tools and strategies for converting existing code into well-engineered modules to promote code reuse and maintainability.

An online version of the slides can be found at http://www.platypiventures.com/perl/present.

Source Filters in Perl

Hendrik Van Belleghem <`beatnik@quickndirty.org`>

Abstract

This is an intermediate level introduction the source filters in Perl. It requires basic knowlege of modules. An online version of this talk can be found at http://beatnik.perlmonk.org/YAPC2002/.

Introduction

1. What are source filters?
2. Why do people write them?
3. Why do people use them?
4. How do you use them?

Writing source filters

1. How do you write them?
2. Writing source filters with Filter
   a. How to use Filter
   b. The Example
   c. Autofiltering
3. `Filter::Simple`
   a. How to use `Filter::Simple`
   b. The Example
   c. Autofiltering
4. Modules from scratch
   a. The Example
   b. Autofiltering

Dissecting existing source filters

1. `Acme::Bleach`
2. `Acme::Buffy`
3. `Filter::CBC`

The End

1. Conclusion
2. Links
3. Thanks

The `Acme::*` Modules

Leon Brocard <acme@astray.com>

The Comprehensive Perl Archive Network is the canonical source for Perl modules. The modules separated into various namespaces. One recent addition is the `Acme::` namespace, intended for use for usefulness-challenged but entertainment-advantaged modules. This talk will tour the modules in this increasingly-popular namespace and show that even the most initially-useless module can be a great inspiration. Meep meep!

Pixie

James A. Duncan <jduncan@fotango.com>

Pixie is an object oriented database that 'just works'. It uses a clever mechanism to serialize the data and a simple query language to view it. More interesting perhaps is the technique that is used to capture the objects. Of particular interest perhaps is that it results in a depth-first non-recursive event based mechanism to discover any object in any Perl data structure.

For more information perhaps see my use.perl journal [http://use.perl.org/~james/journal/], which has an entry containing an early prototype of Pixie.

When Perl is not Quite Fast Enough

or /sacrificing (?:style|maintainability) on the altar of speed/

Nicholas Clark <nick@ccl4.org>

Abstract

This is the script for my talk at YAPC::EU::2002. It's not a transcript of what I actually said; rather it collects together the notes that were in the pod source to the slides, the notes scribbled on various bits of paper, the notes that were only in my head, and tries to make a coherent text. I've also tried to add in the useful feedback I got - sometimes I can even remember who said it and so give them credit.

The slides are here [http://www.ccl4.org/~nick/P/Fast_Enough/00.html], and hopefully it will be obvious where the slide changes are.

Introduction

So you have a perl script. And it's too slow. And you want to do something about it. This is a talk about what you can do to speed it up, and also how you try to avoid the problem in the first place.

Obvious things

Find better algorithm.

Your code runs in the most efficient way that you can think of. But maybe someone else looked at the problem from a completely different direction and found an algorithm that is 100 times faster. Are you *sure* you have the best algorithm? Do some research.

Throw more hardware at it.

If the program doesn't have to run on many machines may be cheaper to throw more hardware at it. After all, hardware is supposed to be cheap and programmers well paid. Perhaps you can gain performance by tuning your hardware better; maybe compiling a custom kernel for your machine will be enough.

mod_perl.

For a CGI script that I wrote, I found that even after I'd shaved everything off it that I could, the server could still only serve 2.5 per second. The same server running the same script under mod_perl could serve 25 per second. That's a factor of 10 speedup for very little effort. And if your script isn't suitable for running under mod_perl there's also fastcgi [http://www.fastcgi.com/] (which CGI.pm supports). And if your script isn't a CGI, you could look at the persistent perl daemon, package PPerl on CPAN [http://search.cpan.org/search?mode=module&query=PPerl].

Rewrite in C, er C++, sorry Java, I mean C#, oops no ...

Of course, one final "obvious" solution is to re-write your perl program in a language that runs as native code, such as C, C++, Java, C# or whatever is currently flavour of the month.

But these may not be practical or politically acceptable solutions.

Compromises

So you can compromise.

XS.

You may find that 95% of the time is spent in 5% of the code, doing something that perl is not that efficient at, such as bit shifting. So you could write that bit in C, leave the rest in perl, and glue it together with XS. But you'd have to learn XS and the perl API, and that's a lot of work.

Inline.

Or you could use `Inline` [http://search.cpan.org/search?mode=module&query=Inline::C]. If you have to manipulate perl's internals then you'll still have to learn perl's API, but if all you need is to call out from perl to your pure C code, or someone else's C library then Inline makes it easy.

Here's my perl script making a call to a perl function `rot32`. And here's a C function `rot32` that takes 2 integers, rotates the first by the second, and returns an integer result. That's all you need! And you run it and it works.

```
    #!/usr/local/bin/perl -w
    use strict;
    printf "$_:\t%08X\t%08X\n", rot32 (0xdead, $_), rot32 (0xbeef, -$_)
      foreach (0..31);
    use Inline C => <<'EOC';
    unsigned rot32 (unsigned val, int by) {
      if (by >= 0)
        return (val >> by) | (val << (32 - by));
      return (val << -by) | (val >> (32 + by));
    }
    EOC
    __END__
```

```
    0: 0000DEAD 0000BEEF
    1: 80006F56 00017DDE
    2: 400037AB 0002FBBC
    3: A0001BD5 0005F778
    4: D0000DEA 000BEEF0
    ...
```

Compile your own perl?

Are you running your script on the perl supplied by the OS? Compiling your own perl could make your script go faster. For example, when perl is compiled with threading, all its internal variables are made thread safe, which slows them down a bit. If the perl is threaded, but you don't use threads then you're paying that speed hit for no reason. Likewise, you may have a better compiler than the OS used. For example, I found that with gcc 3.2 some of my C code run 5% faster than with 2.9.5. [One of my helpful hecklers in the audience said that he'd seen a 14% speedup, (if I remember correctly) and if I remember correctly that was from recompiling the perl interpreter itself]

Different perl version?

Try using a different perl version. Different releases of perl are faster at different things. If you're using an old perl, try the latest version. If you're running the latest version but not using the newer features, try an older version.

Banish the demons of stupidity

Are you using the best features of the language?

hashes.

There's a Larry Wall quote - Doing linear scans over an associative array is like trying to club someone to death with a loaded Uzi.

I trust you're not doing that. But are you keeping your arrays nicely sorted so that you can do a binary search? That's fast. But using a hash should be faster.

regexps.

In languages without regexps you have to write explicit code to parse strings. perl has regexps, and re-writing with them may make things 10 times faster. Even using several with the `\G` anchor and the `/gc` flags may still be faster.

```
    if ( /\G.../gc ) {
        ...
    } elsif ( /\G.../gc ) {
        ...
    } elsif ( /\G.../gc ) {
```

`pack` and `unpack`.

pack and unpack have far too many features to remember. Look at the manpage - you may be able to replace entire subroutines with just one unpack.

`undef`.

undef. what do I mean undef?

Are you calculating something only to throw it away?

For example the script in the Encode module that compiles character conversion tables would print out a warning if it saw the same character twice. If you or I build perl we'll just let those build warnings scroll off the screen - we don't care - we can't do anything about it. And it turned out that keeping track of everything needed to generate those warnings was slowing things down considerably. So I added a flag to disable that code, and perl 5.8 defaults to use it, so it builds more quickly.

Intermission

Various helpful hecklers (most of London.pm who saw the talk (and I'm counting David Adler as part of London.pm as he's subscribed to the list)) wanted me to remind people that you *really really don't want to be optimising unless you absolutely have to*. You're making your code harder to maintain, harder to extend, and easier to introduce new bugs into. Probably you've done something wrong to get to the point where you need to optimise in the first place.

I agree.

Also, I'm not going to change the running order of the slides. There isn't a good order to try to describe things in, and some of the ideas that follow are actually more "good practice" than optimisation techniques, so possibly ought to come before the slides on finding slowness. I'll mark what I think are good habits to get into, and once you understand the techniques then I'd hope that you'd use them automatically when you first write code. That way (hopefully) your code will never be so slow that you actually want to do some of the brute force optimising I describe here.

Tests

Must not introduce new bugs.

The most important thing when you are optimising existing *working* code is not to introduce new bugs.

Use your full regression tests :-).

For this, you can use your full suite of regression tests. You do have one, don't you?

[At this point the audience is supposed to laugh nervously, because I'm betting that very few people are in this desirable situation of having comprehensive tests written]

***Keep* a copy of original program.**

You *must* keep a copy of your original program. It is your last resort if all else fails. Check it into a version control system. Make an off site backup. Check that your backup is readable. You mustn't lose it. In the end, your ultimate test of whether you've not introduced new bugs while optimising is to check that you get identical output from the optimised version and the original. (With the optimised version taking less time).

What causes slowness

CPU.

It's obvious that if you script hogs the CPU for 10 seconds solid, then to make it go faster you'll need to reduce the CPU demand.

RAM.

A lesser cause of slowness is memory.

perl trades RAM for speed.

One of the design decisions Larry made for perl was to trade memory for speed, choosing algorithms that use more memory to run faster. So perl tends to use more memory.

getting slower (relative to CPU).

CPUs keep getting faster. Memory is getting faster too. But not as quickly. So in relative terms memory is getting slower. [Larry was correct to choose to use more memory when he wrote perl5 over 10 years ago. However, in the future CPU speed will continue to diverge from RAM speed, so it might be an idea to revisit some of the CPU/RAM design trade offs in parrot]

memory like a pyramid.

You can never have enough memory, and it's never fast enough.

Computer memory is like a pyramid. At the point you have the CPU and its registers, which are very small and very fast to access. Then you have 1 or more levels of cache, which is larger, close by and fast to access. Then you have main memory, which is quite large, but further away so slower to access. Then at the base you have disk acting as virtual memory, which is huge, but very slow.

Now, if your program is swapping out to disk, you'll realise, because the OS can tell you that it only took 10 seconds of CPU, but 60 seconds elapsed, so you know it spent 50 seconds waiting for disk and that's your speed problem. But if your data is big enough to fit in main RAM, but doesn't all sit in the cache, then the CPU will keep having to wait for data from main RAM. And the OS timers I described count that in the CPU time, so it may not be obvious that memory use is actually your problem.

This is the original code for the part of the Encode compiler (`enc2xs`) that generates the warnings on duplicate characters:

```
    if (exists $seen{$uch}) {
        warn sprintf("U%04X is %02X%02X and %02X%02X\n",
                     $val,$page,$ch,@{$seen{$uch}});
    }
    else {
        $seen{$uch} = [$page,$ch];
    }
```

It uses the hash `%seen` to remember all the Unicode characters that it has processed. The first time that it meets a character it won't be in the hash, the `exists` is false, so the `else` block executes. It stores an arrayref containing the code page and character number in that page. That's *three* things per character, and there are a lot of characters in Chinese.

If it ever sees the same Unicode character again, it prints a warning message. The warning message is just a string, and this is the only place that uses the data in `%seen`. So I changed the code - I pre-formatted that bit of the error message, and stored a single scalar rather than the three:

```
    if (exists $seen{$uch}) {
        warn sprintf("U%04X is %02X%02X and %04X\n",
                     $val,$page,$ch,$seen{$uch});
    }
    else {
        $seen{$uch} = $page << 8 | $ch;
    }
```

That reduced the memory usage by a third, and it runs more quickly.

Step by step

How do you make things faster? Well, this is something of a black art, down to trial and error. I'll expand on aspects of these 4 points in the next slides.

What might be slow?

You need to find things that are actually slow. It's no good wasting your effort on things that are already fast -

put it in where it will get maximum reward.

Think of re-write.

But not all slow things can be made faster, however much you swear at them, so you can only actually speed things up if you can figure out another way of doing the same thing that may be faster.

Try it.

But it may not. Check that it's *faster* and that it gives the same results.

Note results.

Either way, note your results - I find a comment in the code is good. It's important if an idea didn't work, because it stops you or anyone else going back and trying the same thing again. And it's important if a change *does* work, as it stops someone else (such as yourself next month) tidying up an important optimisation and losing you that hard won speed gain.

By having commented out slower code near the faster code you can look back and get ideas for other places you might optimise in the same way.

Small easy things

These are things that I would consider good practice, so you ought to be doing them as a matter of routine.

`AutoSplit` and `AutoLoader`.

If you're writing modules use the *AutoSplit* and *AutoLoader* modules to make perl only load the parts of your module that are actually being used by a particular script. You get two gains - you don't waste CPU at start up loading the parts of your module that aren't used, and you don't waste the RAM holding the the structures that perl generates when it has compiled code. So your modules load more quickly, and use less RAM.

One potential problem is that the way AutoLoader brings in subroutines makes debugging confusing, which can be a problem. While developing, you can disable `AutoLoader` by commenting out the `__END__` statement marking the start of your AutoLoaded subroutines. That way, they are loaded, compiled and debugged in the normal fashion.

```
...
1;
# While debugging, disable AutoLoader like this:
# __END__
...
```

Of course, to do this you'll need another `1;` at the end of the AutoLoaded section to keep `use` happy, and possibly another `__END__`.

Schwern notes that commenting out `__END__` can cause surprises if the main body of your module is running under `use strict;` because now your AutoLoaded subroutines will suddenly find themselves being run under `use strict`. This is arguably a bug in the current `AutoSplit` - when it runs at install time to generate the files for `AutoLoader` to use it doesn't add lines such as `use strict;` or `use warnings;` to ensure that the split out subroutines are in the same environment as was current at the `__END__` statement. This may be fixed in 5.10.

Elizabeth Mattijsen notes that there are different memory use versus memory shared issues when running under `mod_perl` [#item_mod_perl], with different optimal solutions depending on whether your `apache` is forking or threaded.

`=pod` @ `__END__`.

If you are documenting your code with one big block of pod, then you probably don't want to put it at the top of the file. The perl parser is very fast at skipping pod, but it's not magic, so it still takes a little time. Moreover, it has to read the pod from disk in order to ignore it.

```
#!perl -w
use strict;
```

```
=head1 You don't want to do that
big block of pod
=cut
...
1;
__END__
=head1 You want to do this
```

If you put your pod after an `__END__` statement then the perl parser will never even see it. This will save a small amount of CPU, but if you have a lot of pod (>4K) then it might also mean that the last disk `block(s)` of a file are never even read in to RAM. This may gain you some speed. [A helpful heckler observed that modern raid systems may well be reading in 64K chunks, and modern OSes are getting good at read ahead, so not reading a block as a result of `=pod` @ `__END__` may actually be quite rare.]

If you are putting your pod (and tests) next to their functions' code (which is probably a better approach anyway) then this advice is not relevant to you.

Needless importing is slow

`Exporter` is written in perl. It's fast, but not instant.

Most modules are able to export lots of their functions and other symbols into your namespace to save you typing. If you have only one argument to use, such as

```
    use POSIX; # Exports all the defaults
```

then `POSIX` will helpfully export its default list of symbols into your namespace. If you have a list after the module name, then that is taken as a list of symbols to export. If the list is empty, no symbols are exported:

```
    use POSIX (); # Exports nothing.
```

You can still use all the functions and other symbols - you just have to use their full name, by typing `POSIX::` at the front. Some people argue that this actually makes your code clearer, as it is now obvious where each subroutine is defined. Independent of that, it's faster:

`use POSIX;`	`use POSIX ();`
0.516s	0.355s
`use Socket;`	`use Socket ();`
0.270s	0.231s

`POSIX` exports a lot of symbols by default. If you tell it to export none, it starts in *30% less time*. `Socket` starts in 15% less time.

regexps

avoid $&.

The `$&` variable returns the last text successfully matched in any regular expression. It's not lexically scoped, so unlike the match variables `$1` etc it isn't reset when you leave a block. This means that to be correct perl has to keep track of it from any match, as perl has no idea when it might be needed. As it involves taking a copy of the matched string, it's expensive for perl to keep track of. If you never mention `$&`, then perl knows it can cheat and never store it. But if you (*or any module*) mentions `$&` anywhere then perl has to keep track of it throughout the script, which slows things down. So it's a good idea to capture the whole match explicitly if that's what you need.

```
    $text =~ /.* rules/;
    $line = $&; # Now every match will copy $& - slow
```

```
    $text =~ /(.* rules)/;
    $line = $1; # Didn't mention $& - fast
```

avoid use English;.

`use English` gives helpful long names to all the punctuation variables. Unfortunately that includes aliasing

`$&` to `$MATCH` which makes perl think that it needs to copy every match into `$&`, even if you script never actually uses it. In perl 5.8 you can say `use English '-no_match_vars';` to avoid mentioning the naughty "word", but this isn't available in earlier versions of perl.

avoid needless captures.

Are you using parentheses for capturing, or just for grouping? Capturing involves perl copying the matched string into `$1` etc, so it all you need is grouping use a the non-capturing `(?:...)` instead of the capturing `(...)`.

/.../o;.

If you define scalars with building blocks for your regexps, and then make your final regexp by interpolating them, then your final regexp isn't going to change. However, perl doesn't realise this, because it sees that there are interpolated scalars each time it meets your regexp, and has no idea that their contents are the same as before. If your regexp doesn't change, then use the `/o` flag to tell perl, and it will never waste time checking or recompiling it.

but don't blow it.

You can use the `qr//` operator to pre-compile your regexps. It often is the easiest way to write regexp components to build up more complex regexps. Using it to build your regexps once is a good idea. But don't screw up (like parrot's `assemble.pl` did) by telling perl to recompile the same regexp every time you enter a subroutine:

```
    sub foo {
        my $reg1 = qr/.../;
        my $reg2 = qr/... $reg1 .../;
```

You should pull those two regexp definitions out of the subroutine into package variables, or file scoped lexicals.

Devel::DProf

You find what is slow by using a profiler. People often guess where they think their program is slow, and get it hopelessly wrong. Use a profiler.

`Devel::DProf` is in the perl core from version 5.6. If you're using an earlier perl you can get it from CPAN [http://search.cpan.org/search?mode=module&query=Devel::DProf].

You run your program with `-d:DProf`

```
    perl5.8.0 -d:DProf enc2xs.orig -Q -O -o /dev/null ...
```

which times things and stores the data in a file named *tmon.out*. Then you run `dprofpp` to process the *tmon.out* file, and produce meaningful summary information. This excerpt is the default length and format, but you can use options to change things - see the man page. It also seems to show up a minor bug in `dprofpp`, because it manages to total things up to get 106%. While that's not right, it doesn't affect the explanation.

```
    Total Elapsed Time = 66.85123 Seconds
      User+System Time = 62.35543 Seconds
    Exclusive Times
    %Time ExclSec CumulS #Calls sec/call Csec/c Name
     106. 66.70 102.59 218881 0.0003 0.0005 main::enter
     49.5 30.86 91.767 6 5.1443 15.294 main::compile_ucm
     19.2 12.01 8.333 45242 0.0003 0.0002 main::encode_U
     4.74 2.953 1.078 45242 0.0001 0.0000 utf8::unicode_to_native
     4.16 2.595 0.718 45242 0.0001 0.0000 utf8::encode
     0.09 0.055 0.054 5 0.0109 0.0108 main::BEGIN
     0.01 0.008 0.008 1 0.0078 0.0078 Getopt::Std::getopts
     0.00 0.000 -0.000 1 0.0000 - Exporter::import
     0.00 0.000 -0.000 3 0.0000 - strict::bits
     0.00 0.000 -0.000 1 0.0000 - strict::import
     0.00 0.000 -0.000 2 0.0000 - strict::unimport
```

At the top of the list, the subroutine `enter` takes about half the total CPU time, with 200,000 calls, each very fast. That makes it a good candidate to optimise, because all you have to do is make a slight change that gives a small speedup, and that gain will be magnified 200,000 times. [It turned out that `enter` was tail recursive, and part of the speed gain I got was by making it loop instead]

Third on the list is `encode_U`, which with 45,000 calls is similar, and worth looking at. [Actually, it was trivial code and in the real `enc2xs` I inlined it]

`utf8::unicode_to_native` and `utf8::encode` are built-ins, so you won't be able to change that.

Don't bother below there, as you've accounted for 90% of total program time, so even if you did a perfect job on everything else, you could only make the program run 10% faster.

`compile_ucm` is trickier - it's only called 6 times, so it's not obvious where to look for what's slow. Maybe there's a loop with many iterations. But now you're guessing, which isn't good.

One trick is to break it into several subroutines, just for benchmarking, so that DProf gives you times for different bits. That way you can see where the juicy bits to optimise are.

`Devel::SmallProf` should do line by line profiling, but every time I use it it seems to crash.

Benchmark

Now you've identified the slow spots, you need to try alternative code to see if you can find something faster. The `Benchmark` module makes this easy. A particularly good subroutine is `cmpthese`, which takes code snippets and plots a chart. `cmpthese` was added to `Benchmark` with perl 5.6.

So to compare two code snippets `orig` and `new` by running each for 10000 times you'd do this:

```
    use Benchmark ':all';
    sub orig {
       ...
    }
    sub new {
       ...
    }
    cmpthese (10000, { orig => \&orig, new => \&new } );
```

Benchmark runs both, times them, and then prints out a helpful comparison chart:

```
    Benchmark: timing 10000 iterations of new, orig...
           new: 1 wallclock secs ( 0.70 usr + 0.00 sys = 0.70 CPU) @ 14222.22/s (n=10000)
          orig: 4 wallclock secs ( 3.94 usr + 0.00 sys = 3.94 CPU) @ 2539.68/s (n=10000)
            Rate orig new
    orig 2540/s -- -82%
    new 14222/s 460% --
```

and it's plain to see that my new code is over 4 times as fast as my original code.

What causes slowness in perl?

Actually, I didn't tell the whole truth earlier about what causes slowness in perl. [And astute hecklers such as Philip Newton had already told me this]

When perl compilers your program it breaks it down into a sequence of operations it must perform, which are usually referred to as *ops*. So when you ask perl to compute `$a = $b + $c` it actually breaks it down into these ops:

- Fetch `$b` onto the stack
- Fetch `$c` onto the stack
- Add the top two things on the stack together; write the result to the stack
- Fetch the address of `$a`
- Place the thing on the top of stack into that address

Computers are fast at simple things like addition. But there is quite a lot of overhead involved in keeping track of "which op am I currently performing" and "where is the next op", and this book-keeping often swamps the time taken to actually run the ops. So often in perl it's the number of ops your program takes to perform its task that is more important than the CPU they use or the RAM it needs. The hit list is

1. Ops
2. CPU

3. RAM

So what were my example code snippets that I `Benchmark`ed?

It was code to split a line of hex (`54726164696e67207374796c652f6d61`) into groups of 4 digits (`5472 6164 696e` ...) , and convert each to a number

```
    sub orig {
       map {hex $_} $line =~ /(....)/g;
    }
```

```
    sub new {
       unpack "n*", pack "H*", $line;
    }
```

The two produce the same results:

orig		new
21618, 24932, 26990, 26400, 29556, 31084, 25903, 28001, 26990, 29793, 26990, 24930, 26988, 26996, 31008, 26223, 29216, 29552, 25957, 25646		21618, 24932, 26990, 26400, 29556, 31084, 25903, 28001, 26990, 29793, 26990, 24930, 26988, 26996, 31008, 26223, 29216, 29552, 25957, 25646

but the first one is much slower. Why? Following the data path from right to left, it starts well with a global regexp, which is only one op and therefore a fast way to generate a list of the 4 digit groups. But that `map` block is actually an implicit loop, so for each 4 digit block it iterates round and repeatedly calls `hex`. Thats at least one op for every list item.

Whereas the second one has no loops in it, implicit or explicit. It uses one `pack` to convert the hex temporarily into a binary string, and then one `unpack` to convert that string into a list of numbers. `n` is big endian 16 bit quantities. I didn't know that - I had to look it up. But when the profiler told me that this part of the original code was a performance bottleneck, the first think that I did was to look at the the pack docs to see if I could use some sort of `pack`/`unpack` as a speedier replacement.

Ops are bad, m'kay

You can ask perl to tell you the ops that it generates for particular code with the `Terse` backend to the compiler. For example, here's a 1 liner to show the ops in the original code:

```
$ perl -MO=Terse -e'map {hex $_} $line =~ /(....)/g;'
```

```
    LISTOP (0x16d9c8) leave [1]
        OP (0x16d9f0) enter
        COP (0x16d988) nextstate
        LOGOP (0x16d940) mapwhile [2]
            LISTOP (0x16d8f8) mapstart
                OP (0x16d920) pushmark
                UNOP (0x16d968) null
                    UNOP (0x16d7e0) null
                        LISTOP (0x115370) scope
                            OP (0x16bb40) null [174]
                            UNOP (0x16d6e0) hex [1]
                                UNOP (0x16d6c0) null [15]
                                    SVOP (0x10e6b8) gvsv GV (0xf4224) *_
                PMOP (0x114b28) match /(....)/
                    UNOP (0x16d7b0) null [15]
                        SVOP (0x16d700) gvsv GV (0x111f10) *line
```

At the bottom you can see how the match `/(....)/` is just one op. But the next diagonal line of ops from `mapwhile` down to the match are all the ops that make up the `map`. Lots of them. *And* they get run *each time round map's loop*. [Note also that the `{}`s mean that map enters scope each time round the loop. That not a trivially cheap op either]

Whereas my replacement code looks like this:

```
$ perl -MO=Terse -e'unpack "n*", pack "H*", $line;'
```

```
    LISTOP (0x16d818) leave [1]
        OP (0x16d840) enter
        COP (0x16bb40) nextstate
        LISTOP (0x16d7d0) unpack
            OP (0x16d7f8) null [3]
            SVOP (0x10e6b8) const PV (0x111f94) "n*"
            LISTOP (0x115370) pack [1]
                OP (0x16d7b0) pushmark
                SVOP (0x16d6c0) const PV (0x111f10) "H*"
                UNOP (0x16d790) null [15]
                    SVOP (0x16d6e0) gvsv GV (0x111f34) *line
```

There are less ops in total. And no loops, so all the ops you see execute only once. :-)

[My helpful hecklers pointed out that it's hard to work out what an op is. Good call. There's roughly one op per symbol (function, operator, variable name, and any other bit of perl syntax). So if you golf down the number of functions and operators your program runs, then you'll be reducing the number of ops.]

[These were supposed to be the bonus slides. I talked to fast (quelle surprise) and so manage to actually get through the lot with time for questions]

Memoize

Caches function results.

MJD's `Memoize` follows the grand perl tradition by trading memory for speed. You tell `Memoize` the name(s) of functions you'd like to speed up, and it does symbol table games to transparently intercept calls to them. It looks at the parameters the function was called with, and uses them to decide what to do next. If it hasn't seen a particular set of parameters before, it calls the original function with the parameters. However, before returning the result, it stores it in a hash for that function, keyed by the function's parameters. If it has seen the parameters before, then it just returns the result direct from the hash, without even bothering to call the function.

For functions that only calculate.

This is useful for functions that calculate things with no side effects, slow functions that you often call repeatedly with the same parameters. It's not useful for functions that do things external to the program (such as generating output), nor is it good for very small, fast functions.

Can tie cache to a disk file.

The hash `Memoize` uses is a regular perl hash. This means that you can tie the hash to a disk file. This allows `Memoize` to remember things across runs of your program. That way, you could use `Memoize` in a CGI to cache static content that you only generate on demand (but remember you'll need file locking). The first person who requests something has to wait for the generation routine, but everyone else gets it straight from the cache. You can also arrange for another program to periodically expire results from the cache.

As of 5.8 `Memoize` module has been assimilated into the core. Users of earlier perl can get it from CPAN [http://search.cpan.org/search?mode=module&query=Memoize].

Miscellaneous

These are quite general ideas for optimisation that aren't particularly perl specific.

Pull things out of loops.

perl's hash lookups are fast. But they aren't as fast as a lexical variable. `enc2xs` was calling a function each time round a loop based on a hash lookup using `$type` as the key. The value of `$type` didn't change, so I pulled the lookup out above the loop into a lexical variable:

```
    my $type_func = $encode_types{$type};
```

and doing it only once was faster.

Experiment with number of arguments.

Something else I found was that `enc2xs` was calling a function which took several arguments from a small number of places. The function contained code to set defaults if some of the arguments were not supplied. I found that the way the program ran, most of the calls passed in all the values and didn't need the defaults. Changing the function to not set defaults, and writing those defaults out explicitly where needed bought me a speed up.

Tail recursion.

Tail recursion is where the last thing a function does it call itself again with slightly different arguments. It's a common idiom, and some languages can automatically optimise it away. Perl is not one of those languages. So every time a function tail recurses you have another subroutine call [not cheap - Arthur Bergman notes that it is 10 pages of C source, and will blow the instruction cache on a CPU] and re-entering that subroutine again causes more memory to be allocated to store a new set of lexical variables [also not cheap].

perl can't spot that it could just throw away the old lexicals and re-use their space, but *you* can, so you can save CPU and RAM by re-writing your tail recursive subroutines with loops. In general, trying to reduce recursion by replacing it with iterative algorithms should speed things up.

yay for y

`y`, or `tr`, is the transliteration operator. It's not as powerful as the general purpose regular expression engine, but for the things it can do it is often faster.

tr/!// # fastest way to count chars.

`tr` doesn't delete characters unless you use the `/d` flag. If you don't even have any replacement characters then it treats its target as read only. In scalar context it returns the number of characters that matched. It's the fastest way to count the number of occurrences of single characters and character ranges. (ie it's faster than counting the elements returned by `m/.../g` in list context. But if you just want to see whether one or more of a character is present use `m/.../`, because it will stop at the u first, whereas `tr///` has to go to the end)

tr/q/Q/ faster than s/q/Q/g.

`tr` is also faster than the regexp engine for doing character-for-character substitutions.

tr/a-z//d faster than s/[a-z]//g.

`tr` is faster than the regexp engines for doing character range deletions. [When writing the slide I assumed that it would be faster for single character deletions, but I `Benchmark`ed things and found that `s///g` was faster for them. So never guess timings; always test things. You'll be surprised, but that's better than being wrong]

Ops are bad, m'kay

Another example lifted straight from `enc2xs` of something that I managed to accelerate quite a bit by reducing the number of ops run. The code takes a scalar, and prints out each byte as \x followed by 2 digits of hex, as it's generating C source code:

```
    #foreach my $c (split(//,$out_bytes)) {
    # $s .= sprintf "\\x%02X",ord($c);
    #}
    # 9.5% faster changing that loop to this:
    $s .= sprintf +("\\x%02X" x length $out_bytes), unpack "C*", $out_bytes;
```

The original makes a temporary list with `split` [not bad in itself - ops are more important than CPU or RAM] and then loops over it. Each time round the loop it executes several ops, including using ord to convert the byte to its numeric value, and then using sprintf with the format `"\\x%02X"` to convert that number to the C source.

The new code effectively merges the `split` and looped `ord` into one op, using `unpack`'s `C` format to generate the list of numeric values directly. The more interesting (arguably sick) part is the format to `sprintf`, which is inside `+(...)`. You can see from the `.=` in the original that the code is just concatenating the converted form of each byte together. So instead of making `sprintf` convert each value in turn, only for perl ops to stick them together, I use `x` to replicate the per-byte format string once for each byte I'm about to convert. There's now one `"\\x%02X"` for each of the numbers in the list passed from `unpack` to `sprintf`, so sprintf just does what it's told. And `sprintf` is faster than perl ops.

How to make perl fast enough

use the language's fast features.

You have enormous power at your disposal with regexps, `pack`, `unpack` and `sprintf`. So why not use them?

All the `pack` and `unpack` code is implemented in pure C, so doesn't have any of the book-keeping overhead of perl ops. `sprintf` too is pure C, so it's fast. The regexp engine uses its own private bytecode, but it's specially tuned for regexps, so it runs much faster than general perl code. And the implementation of `tr` has less to do than the regexp engine, so it's faster.

For maximum power, remember that you can generate regexps and the formats for `pack`, `unpack` and `sprintf` at run time, based on your data.

give the interpreter hints.

Make it obvious to the interpreter what you're up to. Avoid `$&`, use `(?:...)` when you don't need capturing, and put the `/o` flag on constant regexps.

less OPs.

Try to accomplish your tasks using less operations. If you find you have to optimise an existing program then this is where to start - golf is good, but remember it's run time strokes not source code strokes.

less CPU.

Usually you want to find ways of using less CPU.

less RAM.

but don't forget to think about how your data structures work to see if you can make them use less RAM.

OpenFrame Application Server

James A. Duncan <jduncan@fotango.com>

OpenFrame is an application framework, designed to make programming in a structured manner easier for application developers. This talk presents some of the basic concepts of OpenFrame, the thought that went into creating those concepts, and a couple of the mechanisms that resulted.

Additionally some of the future directions of OpenFrame will be presented.

Introduction to OO Programming

There's a method to this madness

Jos Boumans, http://japh.nu amsterdam.pm <kane@cpan.org>

Abstract

The famous tutorial on object oriented Perl.

About the Contents

Most of the material I will be using is based on the OO tutorials I wrote, which can be found at http://japh.nu. I wrote these as a response to many questions on the beginners@perl.org mailinglist. Every day, at least a few questions about OO programming would be asked and many of them would have been answered if the users had read Damian's book or Advanced Perl Programming. When people inquired why they hadn't done so, the majority answered that most concepts introduced in the afore mentioned books were over their head. They wanted a simple yet clear introduction to both the vocabulary and the concept of OO Perl.

About the talk

This tutorial aims to be just that: an introduction to both the vocabulary and the concept. It requires only basic Perl knowledge, and assumes no knowledge of other languages or OO programming in general. I will attempt to divide the talk into a few chapters, aimed at not only helping the audience understand both concept and vocabulary, but also teaching them how to apply this knowledge in their first attempts at their own modules. After attending this talk, both Damian's book as well as Advanced Perl Programming are recommended literature and should no longer be as elusive.

Part VIII. Lightning Talks

WAIT@oreilly.de

Andreas J. König <andreas.koenig@anima.de>

The WAIT module is the unsung hero among all full text search engines for Perl programmers. It is a rewrite of the famous freeWAIS-sf program in Perl and XS.

From freeWAIS-sf it inherits bullet-proof algorithms from academic information retrieval background. From Perl it gets backing in portability and sophistication. XS contributes speed in time-critical areas.

http://www.oreilly.de/catalog/search.html is a proof-of-concept mod_perl application that is featuring WAIT and XSLT. When you try it out, you will notice:

- It's hard to find a query that tells you "No hits".

  Only two-letter searches may fail. Anything else that doesn't match a word in the dictionary is handed over to a second-level search engine through similarly written words.

- Even if there is no hit, you still get a helpful answer

  Queries that fit no document are turned into digram-searches through the dictionary of all possible words.

- Results are sorted well

  Watch the indicator of the relevance ranking for every hit, even for multiword searches.

- Not all words of a multiword search must be in the dictionary

  Any of the words in a multiword search may be a miss in the dictionary and yet the answers make sense. The reason for that is that search terms are generally ORed together (unlike google).

Other features you might not see on first sight, are full Unicode support (because this servlet is written for Perl 5.8.0) and incremental indexing and de-indexing (we haven't used this feature at O'Reilly because we have no control over the changes in the documents, but you can watch the incremental indexing feature at http://netlexikon.akademie.de/query).

Source code.

A working snapshot of the full source code of the O'Reilly servlet is included in the CVS version of WAIT at sourceforge.net. We would wish that the publication of the servlet encourages you to hack your own search engine.

Thanks.

We (Ulrich Pfeifer and me) would like to thank O'Reilly Germany for giving us the opportunity to experiment with 5.8.0 and WAIT in a real-world scenario.

Links.

http://www.oreilly.de/catalog/search.html - http://sourceforge.net/projects/wait/

Sex, Flies and Microarrays

Lucy McWilliam <lejm3@cam.ac.uk>

I'm a bioinformatics research student who uses (admittedly, not very exciting) Perl on a daily basis. Given the 'Science of Perl' theme, I thought it might be nice to give a simple overview of the (admittedly, quite funky) biology that Perl helps my colleagues and I accomplish.

Object::Realize::Later

Mark Overmeer <mark@overmeer.net>

ORL is a Damian-style of module, which provides wild abilities of Perl to innocent Perl programmers. ORL helps you creating "stubs" for data. Let me explain.

A few programming techniques help you separate "what" from "when". For instance code references: on a certain moment in your program you know "what" you want to execute later. A reference to that code is stored in a scalar. Later "when" time is ripe, you decide to run it simply by calling the subroutine which is stored in that scalar, not knowing what it actually contains.

Similar bahavior for Autoloading. The `AUTOLOAD` method is nearly always used to delay compilation of code to decrease start-up time of a program. By writing the `AUTOLOAD` subroutine, you defined "what" must be loaded, and Perl decides "when" it is loaded (compiled).

ORL uses a comparable trick, but in this case delaying the loading of data in stead of loading code. On a certain moment in your program's run you know "what" data you will need, but you do not need it instantaneously. The automatic loading of the data is only prepared. Later "when" time comes, you do need that data, which is then provided for directly. Maybe that moment will never come, and you saved (a lot of) effort in time and memory consumption.

ORL is based on `AUTOLOAD` as well. It uses re-blessing, and many other dirty tricks, like redefining `isa()` and `can()`. It speeds up certain kinds of programs considerably.

Bull In A China Shop, Canary In A Coal Mine

Michael G. Schwern <schwern@pobox.com>

My experiences tearing through vmsperl while knowing nothing about VMS. How this helped fixup not just VMS but all non-Unix ports. And how this may result in the first release of Perl that won't have “known failures”.

We're not Building a F******g House!

James A. Duncan `<jduncan@fotango.com>`

As workers in an industry we keep trying to find a metaphor for what we do, rather than just explaining it. We tend to be proud of our achievements inside our technical community. However outside of that tight knit group we tend to clam up when it comes to explaining what we do and how we do it to our friends, families, and random people we meet while drinking. The metaphor approach is not working. No one outside of a software background seems to have any understanding of why its hard to change something six months after the development starts primarily because they don't understand what it is we do. There are very simple measures we can and should take as software professionals (more or less) that attack this problem before it gets any more uncontrollable.

Typed Perl

Arthur Bergman <`arthur@contiller.se`>

Quick examples of a typing system using `types.pm`.

Introducing www.perltraining.org

A 7-minute lightning talk about the website

Gabor Szabo `<gabor@tracert.com>`

Abstract

http://www.perltraining.org is an annotated collection of companies and individuals providing Perl training all over the world.

Introducing the website

This talk is not really about Perl, it is more about marketing by someone who barely passed his exam in marketing when doing his MBA. So I guess it is better to skip it right now.

The website is organized according to countries and it is written in English. I started out with only a few names I found in Google but since I put up the site a lot of people asked to be listed. Now there are about 40 entities listed in 10 countries. There are 17 in the USA, 8 in the UK. Nearly 3/4 of the entries are in English-speaking countries but I belive this should not be so unbalanced. It would be great to list companies in other countries and in other languages as well.

By the time you read this the site *should be* also organized according to the (human) languages used for the training and have lists of courses provided by each company. It should also have content in those languages that are used for the training. If you are a Perl trainer or work for an organization that teaches Perl and you are not listed yet or if you teach in any language other then English and there is no content on perltraining.org in that language then this is the right time to contact me for instructions on how to add yourself to the list and how to add content in your language.

After all I cannot provide this information for all the companies in all the languages. I can barely do it for myself.

Benefits from being listed at www.perltraining.org

I guess I don't have to say much about why is it good to be listed in such a 'directory'. Your exposure will grow, you get a few more visitors to your site who are potential customers and also your ranking on Google will increase !

I also believe that having a list of competitors especially in smaller markets can provide additional help to you as one of the major concerns of customers is weather there is support for Perl in their own language and weather they'll have an option to go to another 'supplier' if you go out of business or get too expensive. As much as it sounds strange in a world where one company has some 95% market share people don't like to be locked in to one supplier. So I belive it is good for me to list my own competitors as well and I belive it is good for you too.

No surprise I have nearly failed my marketing exam.

Gábor Szabó gabor@pti.co.il Teaching Perl in English, Hebrew and Hungarian Developing Perl in Perl.

Perl Training and Development in Israel www.pti.co.il Maintainer of www.perl.org.il the Israeli PUG. Member of the Hungarian PUG www.perl.org.hu

Lessons from a failed Perl-XS project

Martin Brech <martin.brech@siemens.com>

- attempting to interface the Lotus Notes C API to Perl
- still an interesting thing to do as poor Notes Admins are still lost without a good scripting language, not to mention data import/export headaches with Notes databases
- Lotus Notes is definitely more a Win32 thing even on the server side
- Lotus Notes on Win32 needs dealing with threads
- right decision for the Perl-XS interface: steal the design of the interface from existing LotusScript class design
- wrong ansatz: tried to construct the objects on the C level for speed reasons
- lessons learned:
    - for such large APIs (600+ functions and numerous structs) *always* start with SWIG to get at the Perl level ASAP
    - Perl, XS and Win32 has been a really explosive mixture for me and ... booom ... left me shattered in pieces