The Perl Review

Volume 0 Issue 3 May 1, 2002

April showers bring May flowers

Letters to *The Perl Review* **i**

About this issue **i**

Community News **ii**

Perl Golf **iii**

Extreme Publishing: Change Happens **1**

brian d foy

Cooking Perl with flex **4**

Alberto Manuel Simões

Parroty Bits: Bit 1, The Parrot Vooms! **11**

Dan Sugalski

Finding Perl Modules **17**

brian d foy

Book Reviews **21**

Web Access http://www.ThePerlReview.com/ **Email** letters@theperlreview.com **Publisher** brian d foy **Editors** David H. Adler, Andy Lester **Technical Editors** Kurt Starsinic, Adam Turoff, Elaine Ashton **Copy Editors** Beth Linker, Jack Hattaway **Contributors** brian d foy, Alberto Manuel Simões, Dan Sugalski, Dave Hoover, Jérôme Quelin **Copyright** Format Copyright © 2002 *The Perl Review*, article content Copyright © 2002 by their respective authors.

Letters

What's wrong with Perl?

Why don't you like Perl anymore?
– *name withheld*

brian writes: *A lot of people missed the joke on last month's Python issue, and I got quite a bit of hate mail over it. Even when I explained it was an April Fools issue, they still thought something was wrong with Perl. This is still The Perl Review, and we still use Perl. :)*

Perl Modules

CPAN is an amazing resource, but... so many modules, so little time! I was wondering if you could consider starting a modules section.
– *Carlos Arenas*

brian writes: *We would certainly like to have articles which discuss modules, but we also need authors to write them, so we are limited by what authors submit. We also do not want to simply reproduce the module documentation, so any articles should show something beyond the simple documentation examples. Authors can propose articles through our website.*

About this issue

We do not have any Python in this issue, but we do have some other non-Perl things. Dan Sugalski shows us a parrot assembler, and Alberto Manuel Simões connects Perl to flex through XS. On the Perl side brian d foy explains how he finds Perl modules so you do not have the waste the time writing your own.

Perl on the web

The Perl Review
http://www.theperlreview.com – the website for this magazine with information for readers and authors

Perl Monks
http://www.perlmonks.org/ – Perl discussion forum

Randal Schwartz's magazine columns
http://www.stonehenge.com/merlyn/columns.html

available at http://www.geekstuff.com

Community News

YAPC::NA Call for Participation

http://yapc-stl.org/archives/000366.html#000366
You can submit proposals for lightning talks until June 5th. This year YAPC::NA is at Washington University in St. Louis, June 22-26.

YAPC::NA Dorm Rooms

http://use.perl.org/articles/02/04/27/1830250.shtml
$USD25 double occupancy and $50 single occupancy. Rooms are air-conditioned and will have net access, but are not on the Washington University campus. Online registration was not yet available at press time.

Upgrade your SOAP::Lite

http://search.cpan.org/search?dist=SOAP-Lite
SOAP::Lite 0.55 fixes a serious security problem that allowed SOAP::Lite to execute arbitrary Perl code on the server side, detailed in Phrack 58.

Google create search interface

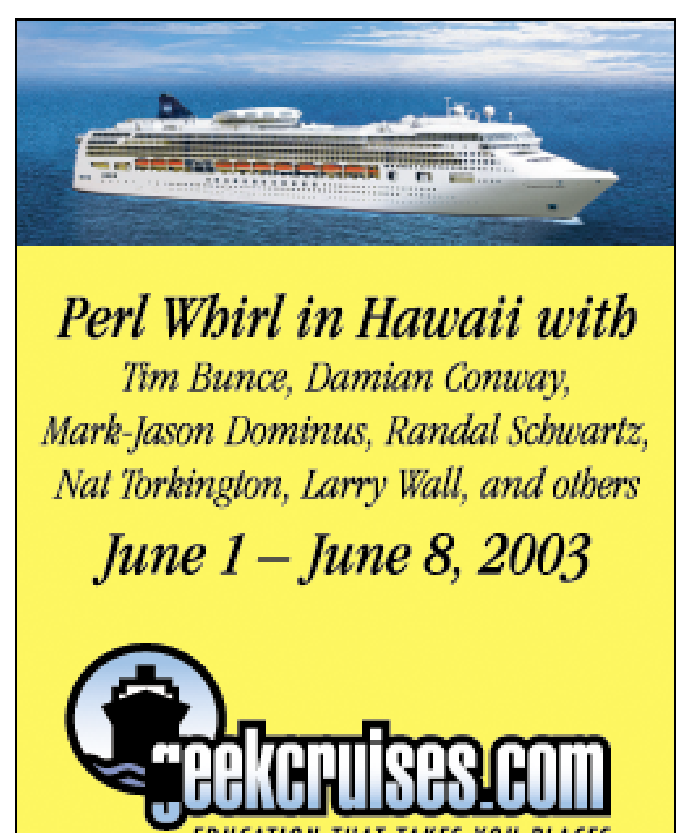
http://www.google.com/apis/
You can programmatically search Google with its new SOAP interface and the WWW::Google::Search module. You must register with Google to use the service.

Perl Conference 6 news

http://conferences.oreilly.com/os2002
Online registration is open for the O'Reilly Open Source Convention, July 22-26. If you register early for two tutorials you can get two free!

Test CPAN

http://www.perl.com/pub/a/2002/04/30/cpants.html
You can set up your computer to automatically download and test Perl modules then feed the results back to the CPAN Testing Service (CPANTS).



New books

Publishers: to have us list your book, send us a note at book_reviews@theperlreview.com.

Perl in a Nutshell, 2nd Edition

Ellen Siever, Stephen Spainhour, Nate Patwardhan; O'Reilly & Associates; 0-59600-241-6; June 2002

Perl & XML

Erik T. Ray, Jason McIntosh; O'Reilly & Associates; 0-596-00205-X; 224 pages; April 2002

XML in a Nutshell, 2nd Edition

W. Scott Means, Elliotte Rusty Harold; O'Reilly & Associates; 0-596-00292-0; 600 pages; June 2002

Perl & LWP

Sean Burke; O'Reilly & Associates; 0-596-00178-9; 400 pages; July 2002

Corrections

Publisher's note: Since we only publish a digital edition at the moment, we can fix errors after we virtually go to press. We had a bit of trouble with this last issue though, and we apologize for the confusion.

In last month's "Python and the Golden Mean", we messed up the Fibonacci series. Python and Paul Prescod can certainly create the series correctly, but apparently we could not. We should have used Paul's Python program instead.

We also said that *The mod_perl Developer's Cookbook* is published by O'Reilly & Associates instead of Sams Publishing.

Perl Golf

Dave Hoover & Jérôme Quelin

April's challenge came from Jon Bentley's *Programming Pearls*. Golfers had to sort into classes all anagram in the input, and to sort the classes by number of words, alphabetized by first word. For example, for the word list "the perl review lives tables viewer elvis stable ablest" the programs should produce

```
elvis lives
review viewer
ablest stable tables
```

Lars Mathiesen, March's beginner catagory winner, won the veteran category with 61 strokes:

```
map!s/
^/ /m|//||print,sort%0for map$0{0,sort/./g}
.=$_,sort<>
```

Because his solution is a single-line for loop, we will describe the right side first. The file input operator, <>, reads the entire input stream, then sort sorts words within anagram classes.

Like many other golfers, Lars used the *signature* technique described by Bentley. The signature for both "elvis" and "lives" is "eilsv". The signatures become hash keys and the anagrams their values. In common Perl, it might look like

```
$hash{ join '', sort split // } .= $_
```

In Perl Golf, this reduces to

```
$0{0,sort/./g}.=$_
```

`/./g` splits a word into characters, then sort() alphabetizes the characters. The deprecated hash key auto-join (used in Perl 4 to create multi-dimensional data structures) joins the signature. Listing a constant scalar before the split-sort forces it into list context, causing the auto-join. The right side is finished.

The %0 hash has signatures as keys and the alphabetized anagram classes as values. Each anagram still has a trailing newline.

The left side expression repeats for each word in the input stream. Fortunately, the number of repetitions needed is always less than the number of words. Each iteration of the loop sorts %0, keys and all, and feeds it to map(). This sort() remedies the secondary alphabetization of the anagram classes. The ingenious algorithm within map() won Lars the tournament:

```
!s/
^/ /m|//||print
```

Due to the ^ anchor, the first regular expression replaces the first newline in $_ with a space when there is another line below it. Bitwise OR, `|`, takes the return value of `!s///` before evaluating its right operand, the empty regular expression. An empty regular expression repeats the last successful regular expression, so bitwise OR returns true until the first regular expression is true and the second is false (i.e. the final word in the class). This is the third consecutive month that the winner used a bitwise operator.

The algorithm does not output hash keys or words without anagrams since the first regular expression never returns true for them. The algorithm traverses the hash, transforming newlines to spaces one at a time. As soon as the final word is reached, bitwise OR returns false to logical OR, `||`, which then finally prints $_. The classes with two words print ahead of classes with three words (and so on).

Marko Nippula won the beginner catagory with his 74 stroke solution.

```
-p $_{1,sort/./g}.=$_}for
(sort+map{@_=sort/.+/g;1x$#_."
 @_
 "}%_){s/\C.*
 //
```

Special thanks goes to **Peter Makholm** for administering much of April's competition. To see all 687 solutions submitted by 128 participants, along with information for May's challenge, visit our web site at http://perlgolf.sourceforge.net/ .

Extreme Publishing: Change happens

brian d foy, comdog@panix.com

Abstract

Project teams do not have perfect information at the start of a project, and requirements change throughout the project. Extreme Programming outlines some practices to reduce the negative affects of change, and *The Perl Review* uses some of these on its way to becoming a print magazine.

1 Introduction

Change happens. Customers rarely know what they really want at the beginning of a project, either from not knowing the technological possibilities or their actual needs. Development teams should not consider either of these to be faults. Things will change, so anyone who ignores that early pays for it later. Strict adherence to an early development plans often does not deliver something the customer actually wants or can use.

The development process helps customers discover things that help them better define what they really want and how they would like to do that. Development teams that expect and prepare for this change deliver more value than those who do not. The customer can redirect the development to meet his better understanding of his needs or the inclusion of better or fresher information.

Extreme Programming (XP) can alleviate the effects of change. No methodology will stop change, and no methodology magically solves all problems. XP recognizes that change is not the problem, and focuses on the things that a team protects during change—their investment. For any project a team can have an investment of time, money, thought, or other things in which they place value. When things change, their investment in those things may lose value, which is okay. However, if a team spends a week perfecting a particular Perl module and the project no longer needs it, the team may think that they wasted their time, which is not okay. Problems occur when the team values their investment over the needs of the customer, and XP tries to minimize that conflict through short release cycles, daily integration, and other things I have mentioned in previous articles. These XP practices reduce the investment and the time until the team discovers if that investment will lose value. Essentially, smaller risks have smaller losses.

1.1 Learning what we really want

The *The Perl Review* team has the same problem even though we are both the customer and the development team. We started with the vague goal of printing a magazine and delivering it to readers. As we broke the problem down, we discovered various things that we needed to do to get that far and set intermediate goals, as I discussed in an earlier "Extreme Publishing" article.

Since we expected changes in direction, we concentrated on the intermediate results, and have provided value along the way. So far, we have published four issues, tens of thousands of people have downloaded the

magazine, and authors have been able to get their message to the people. If we concentrated on the big goal to the exclusion of everything else none of that would have happened. We would probably be no closer to publication on paper than we are right now, and we still might never publish a print issue.

1.2 Responding to new developments

Almost none of the publishing problems we face with this or the next few issues have anything to do with technology, unlike the problems we had with the initial issues. We have pretty much figured out what we want to do with LaTeX and learned to better use the tools that go along with that. Now we need to focus on the magazine bits, and this is where extreme publishing gets interesting.

We started with a small group of Perl people who spend most of their time *not* publishing a magazine. Indeed, Perl programming is the only thing that qualifies us to work on *The Perl Review.* Now that we have solved most of the technical issues, we have to solve the publishing and business ones—that is, the ones mostly dull to technically-oriented people, or, at best, the ones with which most of us have the least experience.

Although I do not discuss specific problems, I can say that they are the sort that once somebody sees them along with our solution, they would wonder why we did not see it much sooner. In some cases the editors told me about them ahead of time and I missed the significance (and now humbly accept the "I told you so"), and others where I failed to provide the proper direction or motivation.

To solve some of these new problems and explore the emerging questions in our plans I have given myself an assignment. Next issue I will have some sort of message about what I think this journal should be, and perhaps more importantly, what I do not want it to be.

2 The future

A lot of people have been asking me about my plans for *The Perl Review.* So far I have done everything specified in my plan—I publish articles and distribute a PDF file, but people want to know about after that—the stuff I have not specified. Sometime in the future, which most techies might call Real Soon Now, I want to try something in print, although I might not distribute it. I have to work out some of the business issues first.

2.1 Closer to print

We have done fairly well with LaTeX although we still have some rough spots. Several people have suggested various commercial typesetting programs like Frame, but I think that we can do this completely with free tools. Our current organization—a loose band of volunteers—would be much less effective if each person had to pay $1000 to participate.

We have to do more work up front to use LaTeX but with each issue we solve more and more problems, and pay less and less attention to the tools and more attention to the content.

So far, we have been able to do this repeatedly for four months, but we have not had to actually print the magazine, which adds more things to do and longer lead times between finishing an issue and putting it into the hands of the readers. I do not think we are ready for that added complexity.

We are getting closer to a print magazine, and we provide you with an intermediate result—our digital version, along the way. We have progressed as far as I had hoped for this stage, but we still have some things to work out.

I have had a lot of good advice and encouragement from Jon Orwant, the first publisher of *The Perl Journal*, but also several caveats. He has quite a bit of experience publishing a real magazine and has not been afraid to share it with me to let me know what I am getting into. Most of our problems publishing a print magazine has nothing to do with the magazine itself, but with the various regulatory bodies and accounting requirements, and I am not ready to deal with those yet. Jon's success does not guarantee my success.

2.2 Creating a business

Once we decide to commit an issue to paper, we have to start giving various people money. The printers, post office, and Staples want money, but so far we have avoided spending anything by using open source tools and the generosity of others. Once we decide to spend money, we need a variety of business devices to control and account for the money, including bank accounts, checkbooks, and the taxman.

If we start to take in money, I want to be able to compensate people who help bring that money in. Our volunteer staff certainly like Perl and have been willing to give quite a bit of their time to help publish the magazine, but I certainly cannot count on that indefinitely, and I cannot work for free forever, either.

I am in the middle of dissolving Perl Mongers, Inc. as part of its agreement with Yet Another Society to concentrate Perl advocacy activities with them, and until I finish that process, I am reluctant to start another business. I already have a pile of forms and filings for New York State and the federal government, and although I have no problem handling that for one business, I do not want to spend all of my day doing them because I am running two businesses.

I promised myself that I would not even think about turning *The Perl Review* into a business until I wrap up the remaining Perl Mongers business, and I expect to complete most of that within the next month. Expect better information in a future article.

3 Conclusion

Change happens, and *The Perl Review* expects it. We take small risks which reduce our investment in any particular direction so we can change to meet our developing needs as we better define what we want to do. Taking some of the principles from XP helps us deliver more value sooner and more frequently than waiting until we can print a magazine.

4 References

Extreme Programming Explained, Giancarlo Succi & Michele Marchesi, Addison Wesley, 2001.

Extreme Programming Installed, Ron Jeffries, Ann Anderson, Chet Hendrickson, & Kent Beck, Addison-Wesley, October 2000.

Extreme Programming Examined, Giancarlo Succi & Michele Marchesi, Addison Wesley, 2001.

Cooking flex with Perl

Alberto Manuel Simões, albie@alfarrabio.di.uminho.pt

Abstract

Perl has a lot of tools for parser generation using Perl. Perl has flexible data structures which makes it easy to generate generic trees. While it is easy to write a grammar and a lexical analyzer using modules like Parse::Yapp and Parse::Lex, these tools are not as fast as I would like. I use the Parse::Yapp syntax parser with the flex lexical analyzer to solve this.

1 Introduction

Some time ago, I needed to make a dictionary programming language parser faster. The original programmer had written the parser with a patched Berkeley Yacc (byacc), which can generate Perl from a grammar specification like common yacc. The only difference resides in the semantic actions, written with Perl. He wrote the lexical analyzer with simple Perl regular expressions. This combination works, but loading the complete dictionary took about 27 seconds, which was too long for me. I combined Perl with flex to cut the parsing time in half.

2 Parser structure

When I write a program in any language and a compiler interprets source code, a parser is involved. It takes the code I write, splits it out as strings of characters with a special meaning—tokens—and then tries to match these token sequences with a grammar. The process has two parts—lexing and parsing.

A lexical analyzer, or lexer, splits a program into pieces called tokens. Regular expressions match each token, which the lexer returns one by one to the caller program. Commonly, it returns the token type with the string which corresponds to the matched token.

Another program, a syntactic analyzer, or parser, puts these pieces together to check if their sequence makes any sense. The parser uses a grammar, or a set of rules, to construct a sentence and check if all the necessary tokens are used, and that no more are needed. It builds a tree of the program structure to generate or interpret code.

Traditional tools for generating parsers in C are lex and yacc, or their GNU variants, flex and bison. When using Perl, people implement parsers in many ways.

3 Why flex

On my first approach I converted the grammar to a full Perl parser. I could have chosen tools like Parse::RecDescent, Parse::YALALR, or others, but I chose Parse::Yapp because its syntax and functionality is very similar to the old yacc. In a half an hour I had a new, working version of the parser, but had only reduced the time for the parsing task by one or two seconds. This small speed-up occurs because Parse::Yapp creates full Perl programs, taking advantage of Perl facilities.

Next, since I could not make the syntax analyzer quicker, I tried to change the lexical one. I wanted to use Parse::Lex, but since it used Perl regular expressions I decided to find another option. I really like Perl, but started thinking about a C implementation for the Parser. That would take a long time but I heard about the XSUB concept and started working on a flex specification to glue with Parse::Yapp.

I could have implemented this glue using different techniques. I could write the lexical analyzer returning integers, where each one represents a different grammar terminal symbol, or I could return a string with the name of the symbol. While the second method can be slower than returning integers, the grammar is more legible, so I went with it.

I implemented my flex analyzer and glued it with Parse::Yapp. It ran in about half the original parsing time. Perl is very flexible, and flex is very fast. I can create simple and fast parsers using both of them together.

4 The Recipe

To demonstrate what I did I use something a lot more simple than my original problem, although I illustrate all of the major points in the process.

I want to create a parser for simple arithmetic problems like "1 + 2". The lexer will break the string into tokens ("1", "+", "2"), perform the arithmetic operation, and return the result.

4.1 Writing the Grammar

I need to write two things—the grammar and the lexical analyzer. I like to start with the grammar, although that is personal preference.

The Parse::Yapp syntax is like yacc. I can write a grammar for arithmetic expressions, which I put in a file I name myGrammar.yp shown in code listing 1.

Code Listing 1: myGrammar.yp

```
1  %token NUMBER NL
2
3  %left '*' '/'
4  %left '-' '+'
5
6  %%
7  command: exp NL  { return $_[1] }
8            ;
9
10 exp: exp '+' exp { return $_[1]+$_[3]}
```

```
11     | exp '-' exp { return $_[1]-$_[3]}
12     | exp '*' exp { return $_[1]*$_[3]}
13     | exp '/' exp { return $_[1]/$_[3]} #forget x/0
14     | number      { return $_[1]}
15     ;
16
17 number: NUMBER   { return $_[1] }
18           ;
19 %%
```

4.2 Writing the Lexical Analyser

Next, I write the lexical analyser in C. I create the file named myLex.l in which I put the instructions for lexing the source, shown in code listing 2.

Code Listing 2: myLex.l

```
1  %{
2  #define YY_DECL char* yylex() void;
3
4  char buffer[15];
5
6  %}
7
8  %%
9  [0-9]+   { return strcpy(buffer, "NUMBER"); }
10
11 \n       { return strcpy(buffer, "NL");     }
12
13 .        { return strcpy(buffer, yytext);   }
14
15 %%
16 int perl_yywrap(void) {
17   return 1;
18 }
19 char* perl_yylextext(void) {
20         return perl_yytext;
21 }
```

The first three lines define the prototype for the lexical analyzer. I return strings instead of the integers returned by default. I have to put these strings somewhere. In this example, I allocate a char array named buffer where I will put the token information.

The next section is a normal lexical analyzer, returning the name of the token or the character found.

The perl_yywrap and perl_yylextext glue the parser to the lexical analyzer. The perl_yywrap function restarts the parsing task while perl_yylextext accesses the text which matched the regular expression through perl_yytext, which flex creates for me.

4.3 Writing the glue

Now I have the two main tools and I am only missing the glue. The easiest way I can do this is creating a module for my parser. I start with h2xs which creates most of the module structure and files for me.

```
h2xs -n myParser
```

I have to edit some of the files that h2xs creates, and add the files that I just created. I need to add some XSUB code to myParser.xs to the lexer in myLex.l to Perl, add a line to the typemaps file, and adjust the Makefile.PL to correctly compile everything.

I copy the flex source to myParser/myLex.l and the Parse::Yapp grammar to myParser/myGrammar.yp to the module directory.

Parse::Yapp expects a yylex function that returns a pair—the name of the token and the matched text—so I add a perl_lex function to myParser.pm, shown in code listing 3. I wrote perl_yylextext in myLex.l (code listing 2), and flex generates perl_yylex for me.

Code Listing 3: The perl_lex subroutine in myParser.pm

```
1 sub perl_lex {
2   my $token = perl_yylex();
3   if ($token) {
4         return ($token, perl_yylextext())
5   } else {
6         return (undef, "");
7   }
8 }
```

I also need an error-handling function in myParser.pm. My error function in code listing 4 will simply print the token read and the token it expected if it encounters an error.

Code Listing 4: An error message for unrecognized tokens

```
1 sub perl_error {
2   my $self = shift;
3   warn "Error: found ",$self->YYCurtok,
4         " and expecting one of ",join(" or ",$self->YYExpect);
5 }
```

Once I have that done, I edit myParser.xs file to map the C functions into Perl ones. I leave the code that h2xs generated alone, and add another header file with the others.

```
#include "myLex.h"
```

At the end of the file I add the parts that connect my lex functions with the functions that I call from Perl, shown in code listing 5. The spaces and newlines are significant. The first line of each function is the return type, followed by a line with the name of the function. Then I have two lines showing Perl where to get the return value from the C functions.

Code Listing 5: Function glue in myParser.xs

```
char*
perl_yylex()
        OUTPUT:
                RETVAL

char*
perl_yylextext()
        OUTPUT:
                RETVAL
```

I create the myLex.h file, shown in code listing 6, which holds the prototypes for the functions in myLex.l.

Code Listing 6: Lexer function prototypes in myLex.h

```
char* perl_yylex(void);
char* perl_yylextext(void);
```

I have to map data types from C to Perl. Integers are trivial and handled directly by perl, but I also have pointers to characters. I create a file named typemap, shown in code listing 7 that translates pointers to characters to the built-in T_PV Perl type.

Code Listing 7: The typemap file

```
char*  T_PV
```

4.4 Putting it all together

The code is now complete, but I need to modify Makefile.PL so it can compile it. The yapp command from the Parser::Yapp distribution creates myGrammar.pm from myGrammar.yp, and flex creates lex.perl_yy.c. I add an additional target to the Makefile through the MY::postamble subroutine. I also add the flex library, fl, to the library list, and name the produced library myLexer.so. Code listing 8 shows my final Makefile.PL.

Code Listing 8: My modified Makefile.PL

```
use ExtUtils::MakeMaker;

$YACC_COMMAND = "( yapp -o myGrammar.pm -m 'myGrammar' myGrammar.yp)";

# This is needed before WriteMakefile
`$YACC_COMMAND`;

WriteMakefile(
                 'NAME'           => 'myParser',
                 'VERSION_FROM'   => 'myParser.pm',
                 'LIBS'           => ['-lfl'], # This is for flex
                 'MYEXTLIB'       => 'myLexer.so', # Our lexer
                );

```

```
15  sub MY::postamble {
16  "
17  \$(MYEXTLIB): lex.perl_yy.c myParser.pm myGrammar.pm
18  \t\$(CC) -c lex.perl_yy.c
19  \t\$(AR) cr \$(MYEXTLIB) lex.perl_yy.o
20  \tranlib \$(MYEXTLIB)
21
22  myGrammar.pm: myGrammar.yp
23  \t$YACC_COMMAND
24
25  lex.perl_yy.c: myLex.l
26  \tflex -Pperl_yy myLex.l
27             ";
28  }
```

I compile my new module with the standard Perl module make sequence.

```
perl Makefile.PL
make
```

4.5 Testing

I can also test my module with the standard Perl test harness, although I have not added any real tests to test.pl.

```
make test
```

This does not really test if my module really works—only that it compiles and loads without error. I add a test, shown in code listing 9, which takes a string from standard input, parses it, and returns the result of the arithmetic operation.

Code Listing 9: My Grammar test

```
1  use myGrammar;
2  my $parser = new myGrammar;
3  my $result = $parser->YYParse(yylex => \&myParser::perl_lex,
4                                                            yyerror => \&myParser::perl_error);
5  print "The result is $result\n";
```

Now, I run `make test` again and enter "1+4*2+3", press enter, and end standard input with ˆD (or ˆZ on Windows), and I get the answer 25.

5 Conclusion

While Perl is really flexible, some tools are quicker than their Perl equivalents. I can use these tools from Perl with a little bit of work.

6 References

Manual pages:

flex(1), perlguts(1), perlxs(1), perlxstut(1)

Perl Module documentation:

Parse::Yapp, ExtUtils::MakeMaker

Parroty Bits: Bit 1, The Parrot Vooms!

Dan Sugalski, dan@sidhe.org

Abstract

In which I lure the unsuspecting reader into the wonderful world of Parrot programming. Along the way I will show you some of the features of the Parrot interpreter, and get you well on your way to writing programs that take advantage of its speed and power, before Perl 6 is even finished!

1 Introduction

In my last article I presented a broad overview of Parrot, the interpreter behind perl 6. This month I show you some programs. All you need to follow along is a working version of Parrot, a C compiler, and perl 5.005_03 or higher to build it.

Parrot is a bytecode machine, and one of the things it can do is execute bytecode that has been stored on disk. Since there is not a working perl compiler for Parrot yet, we—the Parrot Development Team—wrote an assembler.

An assembler is a program that takes a symbolic representation of machine code and turns it into actual machine code. Assembly code itself is very low level—each symbol, or mnemonic, corresponds to a single machine or bytecode instruction. That gives me complete control over the bytecode that is fed to the interpreter, which is good for testing, and it means that you do not have to wait for a working perl parser to start exercising and using Parrot.

By default, parrot assembly language files have a .pasm extension, and compiled bytecode has a .pbc extension. To assemble a program, feed it through assemble.pl, the parrot assembler.

```
./assemble.pl file.pasm --output file.pbc
```

To run compiled programs, I use the `parrot` program that was built when I configured and built parrot. I pass it the name of the compiled bytecode file as its parameter.

```
./parrot file.pbc
```

If I feed bad code into the interpreter, it will crash. Generally the language compiler makes sure that bad code is not created, but since I am doing this all by hand it is my responsibility. If things go wrong it may cause a core dump, crash, burn, or otherwise fail in spectacular ways.

1.1 A simple example

I start with the obligatory “Hello, world” program which is simple, straightforward, and to the point.

Code Listing 10: A simple program

```
print "Hello, world\n"
end
```

The `print`, as you might expect, prints a string to standard output; `end` marks the end of the program. All programs must exit with `end`—if a program falls off the end of the world the interpreter will likely crash.

1.2 Swinging without a net

When I create Parrot bytecode by hand, I work without a safety net. Well-formed Parrot programs should never crash the interpreter, and the perl compiler should never create anything but well-formed parrot code. Raw assembly, on the other hand, can create all manner of abominable code which is free to kill the interpreter.

This is on purpose—safety is expensive. Every microsecond the interpreter spends making sure some code is correct is a microsecond not spent executing that code. If the code is correct, and any code coming out of a compiler should be, the interpreter should not do this—either the code is correct, or the compiler is broken. If the compiler is broken, all bets are off anyway.

I may want to check the bytecode before executing it if I do not trust it—when I execute code from a remote source for example. The interpreter provides a safe mode during which it checks arguments to bytecode, verifies destinations of branches, and enforces resource limits.

1.3 A bit more detailed: Counting from 100 Million

Of course, `Hello, world` is not that useful. I want to try something a little more interesting—counting from 100 million down to zero. Code listing 11 shows how I wrote this in Perl.

Code Listing 11: Perl countdown

```
$x = 100000000;
do {
        $x = $x - 1;
} while ($x);
```

Code listing 12 shows the equivalent Parrot assembly code. It has four lines, and most of them introduce new things.

Code Listing 12: Parrot assembly countdown

```
           set I0, 100000000
REDO:  sub I0, I0, 1
           if I0, REDO
           end
```

The first line, `set I0, 100000000`, puts the value 100 million into integer register 0. The destination is the first argument. In all cases where something is stored, the destination is the first operand. In this case, it is an integer register, which I discuss later. In line 2, `REDO: sub I0, I0, 1`, subtracts 1 from the contents of integer register 0 and stores the results back in integer register 0. Once again, the destination comes first. The `REDO:` is a label—it gives this line of code a name that I can refer to later. In line 3, `if I0, REDO`, checks integer register 0 is true and, if so, branches to the label REDO. As far as Parrot is concerned, an integer is false if it has a value of zero, and is otherwise true. Line 4 exits the program, just like in my "Hello, World" program.

The Parrot version takes 5.78 seconds of user time for me. The perl version takes 170.9 seconds of user time, though it only takes 169.3 seconds of user time when running with `use integer;`.

1.4 Care and feeding of registers

A register is a sort of named temporary spot, a place that the interpreter can get to very quickly and with a minimum of fuss.

Parrot has a lot of registers, and four register types. In addition to integer registers, which hold a machine-native integer, it has floating point registers which hold machine-native double-precision floating point numbers, string registers which hold pointers to strings, and PMC registers which hold pointers to PMCs, or Parrot Magic Cookies, which correspond to Perl-level variables.

Parrot has 32 of each type of register, numbered from 0 to 31. When I refer to a register, I prefix it with its type—P for PMC registers, S for string registers, I for integer registers, and N for floating point number registers.

Parrot's register usage is RISC-like, and like most hardware RISC processors such as the Alpha, SPARC, or PPC, it can only do operations on data in its registers. Parrot cannot directly change data stored in an arbitrary named variable—it must first put the PMC for that variable in one of its PMC registers and operate on that.

For comparison, other virtual machines, such as Java's JVM, .NET, or other interpreters such as Perl 5, Python, and Ruby, are stack-based. They operate on elements that are on top of a stack. If I want to add two numbers together I first push them onto the stack, then call an add operation that takes the two numbers off the stack, adds them together, and puts them back on the top of the stack. Stack machines tend to have more concise bytecode because they do not need to know where things are going—they always go on the top of the stack. On the other hand, they tend to spend a lot of time twiddling with the stack, something Parrot does not have to do nearly as much of.

1.5 Going places

Programs are not that interesting if they do not make decisions and choose to go places. Even my simple example above makes a decision. Parrot has a set of comparison operators, detailed in this comparison table:

if x, dest

If register X is true, jump to label dest

unless x, dest

If register X is false, jump to label dest

eq x, y, dest
: If the contents of registers X and Y are the same, jump to label dest

ne x, y, dest
: If the contents of registers X and Y are different, jump to label dest

gt x, y, dest
: If the contents of register X is greater than the contents of register Y, then jump to label dest

lt x, y, dest
: If the contents of register X are less than the contents of register Y, then jump to label dest

ge x, y, dest
: If the contents of register X are greater than or equal to the contents of register Y, then jump to label dest

le x, y, dest
: If the contents of register X are less than or equal to the contents of register Y, then jump to label dest

All of these operators can take any pair of registers, though both registers generally must be the same type. The `if` and `unless` operators use Perl's idea of truth and falsehood for integers, strings, and numbers. If a number is zero, or a string is empty or the single character 0, Parrot considers it false, otherwise true.

Truth and falsehood for a PMC is a slightly trickier thing. Each PMC class is responsible for deciding whether a particular PMC is true or false. This may seem wishy-washy on the part of the interpreter, but I think class authors are in a better position to determine this than Parrot is.

The destination of a comparison is not an absolute location but an offset, a positive or negative integer, that indicates how far forward or back the program should go. While this theoretically limits the size of any one code segment, integers here are 32 bits wide, giving us offsets of up to 2 gigabytes. Parrot can reach code segments larger than 2 gigabytes with multiple hops. Moving forward or backward in a program by a relative amount is called a branch, and this is how the machine transfers control within a routine. A branch works regardless of where your code is, which means the interpreter can load bytecode off of disk and put it wherever is convenient.

In addition to the conditional operators, which will all branch only when their conditions are met, I can unconditionally change the flow of my program with the flow control operators.

The `branch` operator unconditionally branches forward or backward. They are often found at the very end of loops, transferring control back to the start to recheck the loop condition. Code listing 13 shows a branch at the end of the section labelled LOOP. The corresponding Perl program in code listing 14 does the same thing at the end of the while loop.

Code Listing 13: Branching in Parrot

```
1           set I0, 10
2           set I1, 0
3 LOOP: unless I0, END
4           sub I0, I0, 1
5           add I1, I1, 1
```

```
6           branch LOOP
7   END: print "Done\n"
```

Code Listing 14: Branching in Perl

```
1   my Int $i = 10;
2   my Int $j = 0;
3   while ($i) {
4          $i = $i - 1;
5          $j = $j + 1;
6   }
7   print "Done\n";
```

The `jump` op unconditionally transfers control to an absolute address. The destination is almost never a constant, since I cannot know absolute addresses when I write or assemble my program. Instead a program will generally fetch the address for a named subroutine into a register and jump to it that way.

The `jsr` (Jump to SubRoutine) and `bsr` (Branch to SubRoutine) ops are identical to the `jump` and `branch` ops, respectively, with a single exception. Before transferring control to the destination, they push the address of the next instruction onto the control stack. These are the two ops your programs will use to invoke subroutines.

The `return` op takes the address off the top of the control stack (the one that `jsr` and `bsr` put there) and transfers control there. It is used, as you might expect, to return from a Parrot subroutine.

1.6 Stacks

While Parrot is a register machine, stacks are still handy things to have. Many languages, such as Forth and Scheme, depend heavily on stacks, and having them is convenient when you need a place to stuff something temporarily and a full-fledged entry in a symbol table is more trouble than it is worth.

Because of this, Parrot has stacks. Six of them, in fact—a general-purpose stack for temporary storage, a control stack to store return addresses and the like, and each of the four sets of registers has a private stack.

Parrot uses the register stacks to save the contents of all the registers in one go. Since Parrot uses the registers to do things, they need to be saved across calls to subroutines. With one or two registers, the general stack is fine. Parrot has 128 registers though. Pushing them all one by one is a bit much. So, I created a savestack for each register type so I can save all the registers in one go. The `pushi` opcode, for example, pushes all the integer registers onto the integer register savestack, while the `popi` pops the top of the integer register stack into the integer registers.

Parrot uses the general-purpose stack to save individual registers, and sometimes to pass parameters to subroutines. The values pushed onto the generic stack are typed—if I save an integer register, and then try to restore it into a string register, for example, the interpreter will throw an exception.

When calling a subroutine, the caller is responsible for making sure that its registers are saved off, and subroutines are **not** required to be careful with registers, although subroutines may not leave anything on the stacks. This is a bit different than many systems, which mandate that the callee—the subroutine—is

responsible for saving the registers it uses. Like many other things in Parrot, I did this on purpose. It turns out that the only way to do efficient tail calls and tail recursion is to have the caller save off its registers. Efficient tail calls turn out to be essential for a number of perl constructs, and of course, languages like Lisp and Scheme make heavy use of them, so it turns out to be a nice win.

2 Wrapping it up

I have not talked about quite a bit of Parrot, including subroutines, lexical and global variables, string operations, or exception handling, for example. Do not worry, Parrot can do that, and I will cover it in a later installment.

3 References

PDD 6, as distributed in the Parrot source tree. (docs/pdds/pdd06_assembly.pod) All the valid opcodes are detailed in there.

The Parrot source. CVS checkout instructions and snapshots available from http://dev.perl.org.

Finding Perl Modules

brian d foy, comdog@panix.com

Abstract

The Comprehensive Perl Archive Network has over 11,000 modules in over 3,000 distributions. Any problem worth solving probably has a module, but many people do not know how to find the module that they need. In this article I show how I do it.

1 Introduction

I worked on a web project which included a customer login screen, and I found, after studying the error logs, that people often mistype their usernames. They do not mean to mistype their names, so they often do not know why the login failed. I thought that if I could take what they actually typed and, if it did not match an existing username, find something close, I could create a better interface that helped them solve their login problem while generating fewer support tickets for me.

Donuld Knuth talks about this sort of situation in *The Art of Computer Programming, vol. 3.* The simplest of names, including my own, end up in very strange forms in databases. Knuth invented a scheme he called soundex which reduces names to a short string of characters and letters such that similar sounding names—Smith and Smyth, for instance—reduce to the same string despite their different spellings, making them easy to find.

If I wanted to find similar usernames, I could have either written some code to implement soundex or used code that already existed. Given a reasonable level of functionality, I prefer to not create code that I can get somewhere else, and I can find these modules easily with a few simple practices.

2 The Perl modules list

Since I already knew about soundex, I searched for the term "soundex" in the Perl Modules list (search.cpan.org did not exist yet). I had already scanned the complete modules list, so I knew there was a top level category named Text and that it had all sorts of modules that did text manipulation and matching tasks. I also knew that there was a top level category named String. I quickly found Text::Soundex and went on my way.

Sometimes I have to be stubborn though. I may not know that there is a particular module for a task, but I refuse to believe that there is not. This stubbornness does not make it exist, but it sets my resolve in finding it.

When I started to learn Python I looked for the Python module list. The task is a bit more difficult than in Perl. Jarkko Hietaniemi did a wonderful job creating CPAN, and the Perl Authors Upload Server (PAUSE), created by Andreas Koenig, has been a big part of that success. Python's cryptically named "Vaults of Parnassus" has a lot of same benefit as CPAN, but hides itself under a too-clever metaphor. I knew that

Python must have some sort of code repository, and I did not give up when I did not find a thing called "This is the Python Code Repository". It took me a while to finally realize that the Vaults were what I wanted, and along the way I discovered many other Python things that became useful later. Getting there may be half the fun, but it is also most of the education. If I do not take the journey, I never learn the lay of the land.

The Perl Module List does not list every module on CPAN though, so sometimes I have to do a bit more work. Enter Google.

2.1 Google

Almost any question I need to ask about Perl somebody has already asked in one of the Perl newsgroups. The groups.google.com website archives all of these posts and allows me to search them in a variety of ways. Google took over this task from Deja.com, which was the first well-known searchable usenet web archive, and which I used when I originally researched this problem.

Today, when I pretended to be unaware of Text::Soundex and searched for "fuzzy match perl", the first four of 671 search results lead me to threads which mention Text::Soundex, String::Approx, and Text::Metaphone. I already knew about the first two modules, but not the third.

Good search terms make for useful search results. If I want to find something about Perl, I include the term "perl" and I usually get results about the computer language Perl. If I want to know something about LaTeX—the tool which typesets this magazine—I use the term "latex", and I get a mix of articles about the typesetting language and various fetish devices made with the chemical called latex. If I add a second term, I get even better results. In Table 1, I list the results for three different searches in groups.google.com. In search A, I get almost three million results for just "perl", and none of the top ten results relate to fuzzy matching. In search B I add the term "fuzzy" and get 11,000 results the first of which mentions Text::Soundex, but other results also talk about "fuzzy logic" which does not help me. If I add the term "match" in search C, I get 671 results of which the top five I think are useful.

Search	terms	hits	
A	perl	2,970,000	
B	fuzzy perl	11,000	first match mentions soundex
C	fuzzy match perl	671	

Table 1: Search terms and their returned hits

What if I did not know that I should use the term "fuzzy" though? Computer scientists like to use that word, and somewhere in my career I learned that term, but what if I had not? I could use other terms—approximate, close, almost. When I search for "perl approximate", the first result that Google returns is a reference to perlfaq6 which answers my question. When I search for "perl close", most search results are about the Perl built-in function close(), but the first result mentions String::Approx, which finds close matches within strings.

Sometimes I simply type a full sentence into the search box and let the search engine figure it out. If I search for "How do I approximately match a string in perl?", I get a useful reference in the first ten results—an announcement about a release of String::Approx. Once I read that message, I learn that the term "fuzzy matching" describes my problem, and I can use that to do more searches.

It took me less than 10 minutes to do all of these searches—certainly a lot less time than posting a message

to a newsgroup or mailing list and waiting for the reply.

If you still cannot find what you need, most search engines have some sort of guide that helps you get better results, and the methods you use for different engines depend on how they categorize and organize their information, so I do not discuss that here.

2.2 search.cpan.org

Before Graham Barr created a searchable interface to CPAN, I had to either look at the Perl modules list, look at a directory listing of all of the modules, or know exactly what I wanted. Indeed, as I said before, getting there is most of the education and I am sure that the experience gave me a good idea about what exists in CPAN, but now things are much easier. I can search CPAN based on the module name, author name, or a free text search of the documentation. If I search for “fuzzy match”, I get a result for String::Approx. I can read the documentation directly so I do not have to download or install the module to discover if it meets my needs.

A few other search interfaces to CPAN exist, and you might use one or another for different reasons. WAIT allows you to perform complex powerful searches, while Randy Kobes provides a search interface from the web and the command line.

3 Perldoc.com

Carlos Ramirez makes most of the Perl documentation searchable at Perldoc.com. Although I usually use search.cpan.org to look at a module’s documentation, I use Perldoc.com to search the Perl standard documentation. Now that the Perl documentation takes up over 1,000 printed pages, I no longer suggest that new Perl programmers try to scan all of it, although I think they should read the table of contents.

Carlos recently updated the search capabilities to allow searches on different versions of the Perl documentation, so I do not even have to have a recent version of Perl. If I have access to the internet, I have easy access to all of the Perl documentation that I need regardless of the particular installation of Perl.

4 Conclusion

I can find Perl modules quite easily—I am familiar with the Perl Modules List, know how to use Google, and look at the module documentation before I download the module. Instead of spending too much time looking for modules or reinventing them, I have time to write articles about finding them.

5 References

CPAN FAQ – http://www.cpan.org/misc/cpan-faq.html

Grok CPAN – http://www.cpan.org/authors/id/H/HF/HFB/grok-cpan.pdf

Perl Modules List – http://www.cpan.org/modules/00modlist.long.html

CPAN – http://www.cpan.org

CPAN Search – http://search.cpan.org

Google Groups search – http://groups.google.com

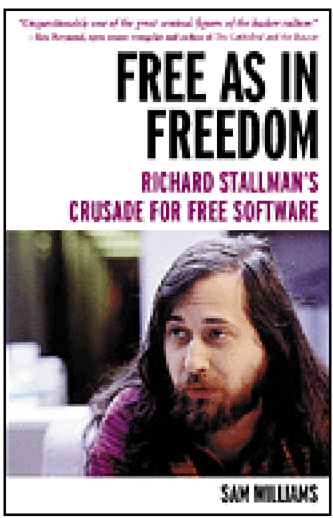
Vaults of Parnassus – http://py.vaults.ca/parnassus/

Perldoc.com – http://www.perldoc.com

WAIT – http://wait.cpan.org

Kobes Search – http://kobesearch.cpan.org

Book Reviews



Free As In Freedom: Richard Stallman's Crusade for Free Software

Sam Williams; O'Reilly & Associates; 0-596-00287-4; March 2002; 225 pages;
http://www.oreilly.com/catalog/freedom/
reviewed by brian d foy

This thin book is a quick read and a gentle and sympathetic introduction to Richard Stallman, although the author tends to be inappropriately dramatic at some points. Sam Williams has written a great supplement to other books which tell the same stories from other perspectives. I would have liked to see more in-depth research, especially from people outside of the free software or open source communities. If you do not know about Richard Stallman or free software, you might miss the bigger picture since the author focuses on Stallman, but you can read the recent works of Lawerence Lessig for more details on the General Public License, for instance.

Transact-SQL Cookbook

Ales Spetic & Jonathon Gennick; O'Reilly & Associates; 1-56592-756-7; March 2002; 282 pages;
http://www.oreilly.com/catalog/transqlcook/
reviewed by brian d foy

O'Reilly markets this book mainly to users of Sybase and Microsoft databases which use Transact-SQL, but I found this book more useful for the authors' approach to problems rather than the particular flavor or SQL they choose. If you use PostgreSQL you use or easily adapt most recipes, but MySQL users are out of luck since they often use sub-selects.

802.11 Wireless Networks

Matthew S. Gast; O'Reilly & Associates; 0-596-00183-5; April 2002; 464 pages;
http://www.oreilly.com/catalog/802dot11/
reviewed by brian d foy

This book covers just about anything you ever wanted to know about wireless networks from the low level theory to the end user configuration of software. The first half of the book explains 802.11 while the second half shows how to use it, including network configuration and tuning. Windows and Linux get their own chapters while the Macintosh gets an appendix. If you want to deploy an 802.11 on a large scale then you probably want this book as a definitive reference. If you are a causal user who just wants it to work, you probably just want to borrow it from your local wireless guru.

Host with pair Networks

- Rock-Solid FreeBSD Unix Reliability
- Edit scripts and compile programs from a secure shell
- Technical, competent support
- Perl 5.6 with over 2000 installed modules
- Free Setup when you switch to us

We've got a web hosting plan waiting for you.

www.pair.com