Perl in a Nutshell

The purpose of this document is to introduce you to enough of the Perl language so that you can understand RWB, the Perl CGI script handed out as part of the first project, and to write your own Perl CGI scripts. In addition to the Perl book mentioned in your syllabus ("the Camel book") and the pointers provided on the course web page, there are other resources on Perl available online and in bookstores.

A Minimally typed, Context-dependent, Interpreted Scripting Language With A Heuristic Parser

The above is a basic description of Perl 5.x (we will use Perl 5.10 in this class, which is installed in /usr/bin). What does it it mean? A scripting language is a high-level language that makes it easy to glue together programs. Perl is interpreted. When you run a Perl script, it is quickly compiled into an intermediate format for the Perl virtual machine which then executes it. In Perl, variables come in a few simple types. There is no such thing as a "char", "int", "real", "float", "double", "struct foo", or whatever. Instead, the type of a variable or constant is inferred from surroundings, from the context in which it is used. This means that type errors may happen as the program is running! Finally, Perl's parser is heuristic, just like a parser for English. This means that there are many different ways of saying the same thing. In fact, there is a running joke that Perl can make sense of line noise. There is also a competition for the best Perl Poetry.

Hello World

This is the smallest Perl program:

```
#!/usr/bin/perl –w
use strict;
print "Hello World\n";
```

The first line specifes that it is the perl interpreter that should execute this code. The "-w" indicates that Perl will print warnings at run-time for type mismatches and other issues. The second line brings in a Perl library, strict.pm. Strict will prevent you from doing things that would be dangerous within certain environments, such as in a CGI script. Finally, the third line prints "Hello World". To run this program, you would place it in the file ("hello.pl", for example), and then run:

```
$ chmod 755 hello.pl
$ ./hello.pl
Hello World
$
```

The "chmod" line sets the permissions on the file to include execute permission for you, your group, and everyone. These are also the permissions you need for CGI scripts. Here is the same program, except modified to act as a CGI script:

```
#!/usr/bin/perl –w
use strict;
use CGI qw(:standard);
print header();
print start_html("Hello World");
print "Hello World\n";
print end_html();
```

To run that as a CGI script, place it in ~you/www (or ~you/public_html, depending on your server's configuration) and chmod 755 hello.pl it. Then you should be able to run it via http://339/~you/hello.pl.

Asking Perl for help

The Perl manual is available in the form of the perldoc command. Some examples of usage:

```
perldoc –h # show to use perldoc
perldoc perlfunc  # show all Perl's built-in functions.
perldoc perlop  # show all Perl's built-in operators
perldoc perlvar # show all Perl's predefined variables
perldoc perlmod # show how Perl modules work
perldoc perlform # show how Perl's output formatting tools work
perldoc perllocale # show how Perl's internationalization support works
perldoc CGI  # describe the CGI module
perldoc DBI # describe the DBI module
perldoc DBD::Oracle # describe the Oracle driver for DBI
perldoc SOAP::Lite # describe the SOAP::Lite module
perldoc IO::Socket::INET # describe Perl sockets
perldoc –f split # detailed information on the split builtin function
perldoc –q SQL # search the FAQ for references to SQL
```

You may also find www.perl.com and www.cpan.org helpful. www.perl.com is the "official" site for perl and its ports. www.cpan.org, or CPAN, is the Comprehensive Perl Archive Network, the way to find perl modules and scripts that can help you.

Adding things to Perl

CPAN provides a HUGE range of scripts and modules that can be added to a Perl installation, making it possible to do, or interact with almost anything you could imagine. It's a giant library maintained and shared by all the Perl users in the world. Perl includes an interactive interface to automatically install tools from CPAN. To start this, run

```
perl –MCPAN –e shell
```

By default, installation is to the shared directories on the system, which you can only do if you have root or administrator privileges. However, you can install your own local extensions as well, and there is documentation on the CPAN site and elsewhere for how to do this.

Variables and Literals

Perl variables include scalars ($x), lists (@x), and hashes (%x). Scalars include ints, floating point values, strings, and references. The following are some examples.

```
$a=5;
$b=53.4;
$c="Hi!\n";
$cref=\$c;   # \$ creates a reference to a scalar
print $a, "\n", $b, "\n", $c, ${$cref};
```

A "reference" is similar to a C or C++ pointer. "$cref=\$c" is basically like "cref=&c" in C or C++, while ${$ref} is like "*cref" in C/C++, meaning "dereference the reference $cref and treat the result as a scalar." You can also treat the result in other ways, like an a list (@{$cref}) or a hash (%{$cref}). Of course, this only makes sense if $cref "really" points to a list or hash. You can find out what kind of thing a reference points to using ref:

```
print "\$cref points to a ",ref $cref,"\n";
```

We will discuss references as we go through this section.

A list (or one dimensional array if you like), contains a list of scalar variables and . Any combination is OK. You can also find out the index of the last element of the list.

```
@d = ($a,"Hi", 34, 10.4,$cref);
print @d;
print "The last element of the list is $#d.  The list has ".($#d+1)." elements.\n";
print "The contents are\n";
for ($i = 0; $i<=$#d;$i++) {
        print "$d[$i]\n";
}
```

You can convert lists to scalars by using join. Join concatenates list items using its first argument. We could write the above for loop as:

```
print join("\n",@d), "\n";
```

There is no notion of a list of lists.

```
@e = (@d,@d);
```

just concatenates the two input lists. A typical approach to lists of lists is to use list references:

```
@e =(\@d, \@d);
foreach $listref (@e) {
   print @{$listref}, “\n”;
}
```

You make lists using ( ) and you can also make references to anonymous lists using [ ]:

```
$e=[ [ 1, “Hi”, 3], [2], [“Wow”, “each”, “list”, “can have lots”, “of”, [“things”] ]];
```

$e is now a reference to a list that contains three references to lists. The third list contains a further reference to a sublist. Note that “$e” is a different variable from “@e” and both can coexist. The responsibility for dealing with such complex data structures as this is solely that of the program. Perl does very minimal typechecking. It’s sometimes helpful to get a complex data structure like this pretty-printed for you:

```
use Data::Dumper;
print Dumper($e);
```

Sometimes it is helpful to dereference references in more elegant ways. For example, the following two lines are equivalent:

```
print @{@{$e}[1]}[0], “\n”;
print $e->[1]->[0],”\n”;
```

The “->” is equivalent to the “->” operator in C/C++. How to interpret that is pointed at by the dereference is determined by what follows “->”

Perl includes hash tables (or “associative arrays”) as first class elements of the language. This is extremely powerful because it means you can look up data by key instead of by position as in a list:

```
%h = ( “x”=>”hi”, “y”=>”bye”);  # create hash with some default mappings
$h{“y”} = “byebye”;
$h{“z”} = [ 1, 2, 3, “Any other list stuff here”];
$h{“h”} = { “name”=>”Peter”, “favcolor”=>”black”}
print Dumper(\%h);

print “This hash table has the following keys: “, join(“, “,keys %h),
      “   and values: “, join(“, “, values %h), “\n”;
foreach $k (sort keys %h) {
  print “$k maps to $h{$k}\n”;
}
print $h{“h”}->{“name”},”’s favorite color is “, $h{“h”}->{“favcolor”},”\n”;
#if you skip the ->s, perl “figures out” what you’re likely trying to do.
#the following line has the same result!
```

```
print $h{"h"}{"name"},"'s favorite color is ",$h{"h"}{"favcolor"},"\n";
```

As you can see, you can include arrays and hashes inside hashes. You can also include both inside arrays. The combination of arrays, hashes, and references allow you to build essentially arbitrary data structures.

You can wipe out variables:

```
undef $k;
```

And you can use undef as a value, similar to NULL in SQL:

```
@x=(1,2,undef,3);
```

You can check to see if a variable is defined:

```
if (defined $k) { print $k; }
```

Variables are scoped in various ways. By default, all variables have global scope! You have to explicitly set their scope otherwise. One thing that "use strict" does is force you to do this. The most important other kind of scope is "my":

```
$k=4;  # $k is visible globally
my $k=4; # $k is visible only within this block, file, or eval.
```

There is some additional subtlety in scoping ("our", "local"), but we'll leave it at that.

You can learn more about variables using perldoc perlvar, and about references using perldoc perlref.

Working with Lists and Hashes

A lot of power in Perl comes from the tools for handling lists and hashes, and the fact that these data types are built into the language as primitives.

You can manipulate a list using the [] operator as you saw before:

```
@a=(1,2,3);
$a[0]=33;
print $a[2];  # prints "3"
```

You can assign whole lists:

```
@tempa = @a;
```

You can also grab sections of the list using the , and .. operators:

```
@b=@a[1 .. 2, 0, 0..2];  # sets b to (2,3,33,33,2,3)
```

You can push and pop items from the back of the list to make it into a stack:

```
push @b, “hi”, “bye”;  # b is now (2,3,33,33,2,3,”hi”,”bye”)
$val = pop @b; # $val is “bye”, b is (2,3,33,33,2,3,”hi”)
push @b, $val; # b is now (2,3,33,33,2,3,”hi”,”bye”)
```

You can also shift from the front. The combination of shift and push give you a queue:

```
$val = shift @b;  # val = 2 and b is now (3,33,33,2,3,”hi”,”bye”)
```

You can also push items onto the front of the list using unshift:

```
unshift @b, $val;  # b is again (2,3,33,33,2,3,”hi”, “bye”)
```

The splice function lets you remove a segment of the list and replace it with a new list:

```
splice @b,1, 5, “cow”, “moo”;  # b is now (2,”cow”,”moo”,”hi”,”bye”)
```

You can reverse your list;

```
@rb = reverse @b;
```

You can translate between lists and scalars using join and split. Join takes a string and a list. It returns a string that contains the elements of the list concatenated using the input string:

```
$out = join(“, “, @b);  # $out=”2, cow, moo, hi, bye”
```

Split does the reverse. It takes a string and splits it into list elements based on a regular expression (more about regexps later):

```
@c = split(/, /,$out); # @c now has the same contents as @b;
```

You can sort your list:

```
@sb = sort @b;
```

If you want, you can supply your own comparison function:

```
@sb = sort { $a > $b } @b;
```

{$a > $b} is an an anonymous chunk of code. Such anonymous functions are first class objects in Perl. You can even construct them at run-time!

Another place where anonymous functions are extremely handy is with map and grep. map applies a function to every element of a list:

```
@c = map { “$_ hi” } @b;  # @c= (“2 hi”, “cow hi”,”moo hi”,”hi hi”,”bye hi”);
```

The variable “$_” is the implicit scalar input argument to the function. More about implicit variables later. Grep is similar, except that it will only return those elements where the function returns true. The following are equivalent:

```
@c = map { $_ if /cow/ } @c;
@c = grep /cow/ @c;
```

This returns the unique words (whitespace separated) in @c:

```
map { $k{$_}=1 } @c;
print join(“, “, sort keys %k);
```

Hashes, or associative arrays are straightforward. To create a hash:

```
%hash = ( “name” => “Peter”,
            “lame reference in other handout is to”=>
                 “Tiberius Claudius Drusus Nero Germanicus” );
```

To read and write values:

```
print $hash{“name”};
$hash{“what handout”} = “the oracle one”;
```

To get the keys and values of the hash:

```
@k = keys %hash;
@v = values %hash;
```

To delete a key and its value:

```
delete $hash{“name”};
```

To iterate through all the key, value pairs:

```
while ( ($key, $value) = each %hash) {
        print “$key = $value\n”;
}
```

Notice that Perl lets you return lists! For more info, look at perldoc perlvar.

Predefined variables and implicit variables

Perl includes a number of predefined variables, some of which are set implicitly at different times. The original names for these variables were kind of obtuse, so new, alternative names can also be used, provided you first "use English;" The following is a short list of some of these variables

Original	**Alternative**	**Description**
$_	$ARG	Scalar argument (for example, to map)
@_		List argument (for example, to a subroutine)
$1, $2, …		Where pattern matches from regexps with parentheses go
$.	$NR	Current line number on last used file handle
$|	$OUTPUT_AUTOFLUSH	Force autoflushing (unbuffered I/O) for output
$?	$CHILD_ERROR	Status returned by last pipe close, backtick, or system call
$!	$ERRNO	Current global error number
$$	$PID	Process ID of this process
$@		Output of last die, if any
$ARGV	$ARGV	Current file being read from
@INC	@INC	Search path for Perl modules
@ARGV	@ARGV	Command-line arguments
%ENV	%ENV	Environment variables
%SIG	%SIG	Signal handlers

You can find all of the perl predefined variables and implicit variables by checking out perldoc –f perlvar.

Memory management

Perl includes reference-counting garbage collection so there is no need to ever explicitly delete anything. If you want to do something similar, do "undef variable". All variables are heap variables, thus it is safe to return references to locally declared variables from subroutines, quite unlike C/C++.

Operators and built-in functions

Perl includes a wide range of operators and built-in functions. Essentially all the operators you're familiar with from C/C++ are available here (+, -, *, /, %, etc). In addition, there is a wide range of other operators that are imported from the bash shell language (-e, -d, etc), and a large number of mathematical operators and implicit functions from Fortran (**, sin, etc).

A wide range of useful string operators are also available. The one we will make lots of use is the "." operator, which concatenates two strings. It is also available for assignment as ".=":

```
$a="hi";
$b="bye";
$c=$a.$c;  # "hibye"
$c.=$c;  # "hibyehibye"
```

To test strings for equality and inequality, use eq, ne, gt, lt, ge, and le to do the equivalent of ==, !=, >, <, >=, and <=, which work properly only for numbers.

```
print "EQUALS HI" if ($a eq "hi") ;
print "EQUALS 5" if ($a == 5);
```

Operators can often be written in multiple ways. For example, the following are equivalent:

```
if (($a eq "x") && !($b eq "z") {}
if (($a eq "x") and not ($b eq "z") {}
```

Logical operators short circuit. This lets us say things like:

```
$#ARGV==0 or die "usage: script.pl name\n";
```

Don't forget perldoc –f perlop and perldoc –f perlfunc

Regular expressions

Perl includes built-in, first-class regular expressions. Two things every computer science student should know about languages are the theories of regular expressions and context-free grammars. Perl's regular expression system is a direct implementations of the first, and a partial implementation of the latter. This is where a lot of its power in processing textual data comes from.

In its simples form, a regular expression is a textual pattern against which strings can be matched. Regular expressions are usually demarked /:

```
/fox/ # pattern of characters "f","o","x"
/[fF]ox/ # pattern of characters "f","o","x" or "F", "o", "x"
/.ox/ # pattern of three chracters ending in "o", "x"
/.+ox/ # pattern of three or more characters ending in "o", "x"
/.*ox/ # pattern of two or more characters ending in "o", "x"
/\d+/ # one or more digits
```

/\s+\S+\s*/ # one or more whitespace characters, followed by one or more non-whitespace characters, followed by zero or more whitespace characters

Regular expression matches uses the =~ operator:

```
$a=”Now is the time for all good men and women to come to the aid of their party.”;
$b=”the quick brown fox jumped over the lazy dog.”
print “$a is a FOX statement” if $a=~/fox/;
print “$a is a PARTY statement” if $a=~/party/;
print “$b is a FOX statement” if $b=~/fox/;
print “$b is a PARTY statement” if $b=~/party/
print “$b contains some character of the alphabet” if $b=~/[a-z]+/;
```

You can search and replace:

```
$a =~ s/their party/the Democratic party/g;  # search (s=search) and  replace all
(g=global) instances of “their party” with “the Democratic Party” in the string $a.
```

You can extract parts of a match:

```
$a=~/all\s+(\S+)\s+men/;
$adjective = $1;  # $1 is the first parenthesized part of the match.  “good” here
```

You can learn more about Perl regular expressions using perldoc perlre.

Control flow

Perl includes all control flow constructs that you might expect from C or C++: if/then/else, for loops, while loops, do while (do “until” here), and goto. It also includes list iteration constructs from csh, like foreach. One thing that you may have also noticed by reading this document are conditional statements:

```
print “Hi” if $do_hi==1;
```

Loop bodies can include “next”, “redo”, and “last”, all of which can refer to a specific loop:

```
foreach $i (keys %x) {
        next if $i=~/meta/;  # do next key if it’s a meta key
        last if $x{$i}=~/$ourvalue/; # leave the loop if we find the value we need
}

OUTER: foreach $i (@list1) {
INNER:          foreach $j (@{$i}) {
                        redo OUTER if $i=$j;
                }
         }
```

Switch statements are not a part of Perl. However, if you “use Switch;”, you’ll get something similar to a C/C++/Java switch statement. It will not be particularly fast.

You can raise an exception in Perl using “die”:

```
if ($#ARGV!=0) {
         die “usage: munge_file.pl filename”;
}
```

also often written as

```
$#ARGV==0 or die “usage: munge_file.pl filename”;
```

The Perl equivalent to throw/catch in C++ or Java is die, eval, and $@:

```
eval { some code that may call die};
if ($@) {
        print “There was an error: $@”;
} else {
        print “There was no error”;
}
```

Eval is incredibly powerful, btw. Using eval, you can take any string, even one that you construct at run-time and make Perl interpret it as if it was part of your program!

```
$statement=”print \”Matches\” if (VAR =~/STRING/)”;
$statement =~s/VAR/\$a/g/;
$statement =~s/STRING/hello/g;
eval $statement;
```

Think about what this means for a minute: you can write a program that writes programs for you.

You can find out more about Perl control flow using perldoc perlsyn.

Writing and calling subroutines

Perl subroutines are really just named blocks of code that can return values

```
sub Hello {
        return (“Hello”,”Goodbye”);
}
```

You call them as you might expect:

```
@results = Hello();
```

```
# $results[0] = “Hello” and $results[1] = “Goodbye”
```

Notice that a subroutine can return a scalar, a list, or a hash! Arguments are pass by value. If you want to modify an argument, you need to pass a reference to it.

The arguments to any subroutines are simply a list with the special name of @_:

```
sub PrintArgs {
	my @args = @_;
	for (my $i=0;$i<=$#args; $i++) {
		print “Argument $i is $args[$i]\n”;
	}
}
```

Any arguments to a call to a subroutine are interpolated into a single list:

```
PrintArgs(1,2,3,(‘a’,’b’,’c’)); # same as PrintArgs(1,2,3,’a’,’b’,’c’)
```

A common trick is to make named arguments using a hash reference:

```
sub PrintArgsHash {
	my $hashref = shift;  # implicitly, shift works on @_
	foreach my $key (keys %{$hashref}) {
		print “The argument named $key is “,$hashref->{$key},”\n”;
	}
}

PrintArgsHash({foo=>5, bar=>6, baz=>”Hiyah”});
```

You can also create anonymous subroutines (unnamed):

```
$subref = sub { print “hi\n”; };
…
&$subref();
```

You can use anonymous subroutines to implement closures. In a closure, a subroutine returns a new subroutine that can be called later to continue some ongoing process.

You can learn more about Perl subroutines using perldoc perlsub.

Using modules

You can make Perl code a part of a library, or “module”, in a very straightforward way. Here is of a perl module. Notice that it has the extension .pm:

```
$ cat pm.pm
package pm;
return 1;
```

```
sub Hello {
	print "Hello\n";
}
```

The first line indicates that the code in this file is a part of the “pm” package. The value returned (1) is used to indicate to Perl that the module has been successfully loaded. The Hello subroutine is exported from the package. The general convention is to put all the code associated with a given package name into a file with the same name, suffixed with pm. Package names can be hierachical. For example, we might have Microblog, Microblog::Display, and Microblog::Display::Tree, which we would place in the files Microblog.pm, Microblog/Display.pm, and Microblog/Display/Tree.pm. Perl looks for .pm files in the list of directories given in the PERLLIB environment variable, and in the current directory.

This is an example of a file that uses the pm module:

```
$ cat pt.pl
#!/usr/bin/perl -w
use pm;
pm::Hello();
```

Here we “use” pm to get Perl to load and ready the pm module. We then refer to elements in the pm module as in the last line.

You can also use the Exporter module (perldoc Exporter) to push symbols from the namespace of a package into the namespace of the user of the package. This allows you to avoid explicitly referring to the package. For example, this is what it looks like from the package point of view:

```
$ cat pm.pm
package pm;
require Exporter;
@ISA = qw(Exporter);
@EXPORT_OK = qw(Hello);
return 1;

sub Hello {
	print "Hello\n";
}
```

The “EXPORT_OK” list is the list of symbols that will appear wherever the package is used. This is what it looks like from the user’s point of view:

```
$ cat pt.pl
#!/usr/bin/perl -w
use pm qw(Hello);
```

```
Hello();
```

Notice that we can now supply an optional list of symbols we want when we use the module. Also, notice that now that we have imported Hello, we can call it without the pm:: prefix.

Perl modules are considerably more complex than what I've shown here. In particular, I haven't told you about construction (BEGIN, INIT), checkpointing (CHECK), and deconconstruction (END). The combination of Perl modules and references is used to implement object-oriented programming in Perl.

To learn more about Perl modules, look at perldoc perlmod.

Input and Output

Way back in the day, Perl stood for "Practical Extraction and Report Language", so, as you might expect, it has powerful I/O features. I/O is done using file handles. Every Perl script as three open when it starts: STDIN, STDOUT, and STDERR. Gee, does this look a little bit like Unix or what? STDIN is the "standard input" to your program – what the user types, or what's piped in, or what Apache feeds it, etc. STDOUT is the "standard output" of your program. When you print, by default, you send stuff to STDOUT. STDERR is the standard place to send error messages. You can explicitly refer to STDOUT and STDERR when you print:

```
print STDOUT "Thank you, kind user\n";
print STDERR "Jeez, what a dork\n";
```

You can open your own files for output:

```
open(MYHANDLE,"> log.txt");  # > overwrites, >> appends
```

print on you filehandles:

```
print MYHANDLE "User $user just tried to hack the system\n";
```

and close them:

```
close(MYHANDLE);
```

By default, most output is buffered, meaning that Perl will keep collecting your print statements until it thinks there is enough accumulated data that it's worth actually printing it. You can turn this off:

```
use FileHandle;
MYHANDLE->autoflush(1);
```

Input is pretty straightforward, using the <> operator. This is how to read a line of input from STDIN:

```
my $line = <STDIN>;
```

Note that the line includes the trailing end-of-line character. You can eliminate it using chomp:

```
chomp $line;
```

What do you think this might do?

```
my @lines = <STDIN>;
```

That's right! It'll keep reading lines from STDIN until end-of-file and return them to you in one giant list. You might then be tempted to process a file something like this:

```
open(FILE,"myfile.txt"); # note lack of > or >>
my @lines=<FILE>;
close(FILE);
foreach my $line (@lines) {
        next if not $line=~/$name_searched_for/;
        chomp $line;
        my ($name, $grade, $notes) = split(/\t/,$line);
        print join("\t", $name, $notes), "\n";
}
```

However, that basically limits you to files that are smaller than memory, so the common idiom is more like:

```
open(FILE,"myfile.txt");
while (my $line = <FILE>) {
        next if not $line=~/$name_searched_for/;
        chomp $line;
        my ($name, $grade, $notes) = split(/\t/,$line);
        print join("\t", $name, $notes), "\n";
}
close(FILE);
```

If you call the operator <> on nothing, just as <>, you get the effect of scanning through all the lines on all the files passed to your script as parameters. You use this to make incredibly one-liners as decribed later.

Perl also supports binary I/O, random access, sockets, pipes, etc, etc. As you might expect. For more information, look at perldoc perlio and perldoc perlop.

Interfacing with external programs

Using just the I/O you saw in the previous section, you can already interface to external programs to get data from them or send data to them. For example:

```
open(MATLAB,"| matlab –nojvm");
use FileHandle;
MATLAB->autoflush(1);
```

This opens up matlab (--nojvm means "spare me the slow and pitiful Java GUI and just give me the command-line version"), and returns a file handle that is connected to matlab's input. This means that anything you print on MATLAB will be interpreted by matlab as a command typed by the "user". For example:

```
print MATLAB "x=normrnd(0,1,[1000 1]); plot(x);\n";
```

would make matlab generate some random numbers for you and then plot them.

To get data from a program, you do something like this:

```
open(LS,"ls |");
```

This runs "ls" and connects its standard output back to you. You could then iterate through the files list in the expected way:

```
while (<LS>) {
        next if not /myfile/;
        system "cat $_";
}
```

Here, we're using the implicit variable $_, which is where <LS> sends its data if no variable is specificed. It's also the implicit variable used in regular expression matching.

The system command is just a way of executing a command line as if you typed it at the shell prompt yourself. Actually, a shell sub process is started, the command is executed, and then the shell process is terminated. You can also execute commands using the backtick operator. This is especially important if you want to get all the results back into a Perl variable. For example:

```
my @files = `ls`;
```

does pretty much what you might expect – it runs ls and turns the results into a list, one list entry per output line of ls.

By the way, you may be thinking it would be very neat if you could open a program up for both reading and writing – that way you could send commands to it and get back the results. You might be tempted to write something like:

```
open(MATLAB,"| matlab –nojvm |");
```

Unfortunately, that won't work. You will need to do something like this:

```
use IPC::Open2;
use FileHandle;
open2(MATOUT,MATIN, "matlab –nojvm");
MATOUT->autoflush(1);
MATIN->autoflush(1);
print MATOUT $command;
$results=<MATIN>;
```

Note that you are responsible for synchronizing with the program you're running.

On Unix, Perl also supports lower level Unix process control and IPC mechanisms. For example, fork, exec, and wait are all available. Signal handlers are set via the %SIG hash. The Perl that comes with the free cygwin Unix emulation environment for Windows (www.cygwin.com) emulates all of this functionality.

Perl also supports interfacing to Windows IPC. In particular, on Windows, Perl can script ActiveX and COM. This means you can write a Perl script that controls Internet Explorer or Excel. To do this, you need the ActiveState Perl implementation, which is also free.

Database Access in Perl

The Perl interface for accessing databases is called DBI. There is a second handout, on Oracle, that describes how to get started using this interface. perldoc DBI will also tell you more.

CGI Programming in Perl

Earlier in this handout, I showed you what a minimal CGI program written in Perl looks like. You will also be seeing a more complex program in the form of Microblog. perldoc CGI will also tell you more.

Amazing Perl Things

Eval is an incredibly powerful tool. Its existence makes it possible for Perl programs to easily create new programs on the fly and even modify themselves as they run.

Closures are a much more powerful feature than they look at first glance. In particular, a closure includes the context (the "lexical scope") that existed when it was created. This gives you a way of elegantly maintaining state needed for, for example, implementing an iterator. The next step after closures are continuations. A continuation captures the current state of the program in such a way that we can return to it later. Think about an iterator that does recursive traversal of a tree. At any point in time, the stack contains the state of the traversal. When our iterator was called, it would return the program to the stack state that was true the last time the iterator was called. The iterator would then extract the current value it points to, take the next step in the traversal, and then save the stack for the next time it is called. That "saved stack" is the continuation – we can continue the traversal from that point anytime we want. This sounds pretty simple, but recall that the dynamic call graph of a program isn't a linked list – it's a DAG. This makes "saving the stack" subtle and and interesting. Perl supports continuations.

Perl contains the lambda calculus: http://perl.plover.com/lambda/

Perl's regular expressions are more powerful than the regular expressions you learned about in theory. In fact, Perl regexp matching is NP-complete: http://perl.plover.com/NPC/NPC-3SAT.html. You can, for example, do parenthesis-matching in Perl regexps, although it's not pretty.

Perl/Tk lets you use the nice, portable, Tk GUI library from TCL/Tk without having to use TCL.

The Perl command-line lets you easily write "one-liners" that quickly solve problems. For example, suppose I want to extract out the third column of some file:

```
perl –e 'while (<STDIN>) { chomp; split; print $_[2],"\n"; }' < file.in > file.out
```