Introduction to Perl Programming

Presentation for LX865

Gregory Garretson

December 6, 2004

Originally a presentation for the
Perl Learning Group
(CL@BU: clabu.bu.edu)

Overview

I. A few introductory notes

Caveats

- I'm a linguist. I have no formal background in computer science. If you have difficult questions about computer science, ask Paul.
- I do know how to program quite well in Perl, but I am not an expert. There are many, many things you can do with Perl, and I have learned only a small subset of these.
- Having said that, feel free to ask me things if you think I can help with your exercises.

Nomenclature

- Here are some terms that are used pretty much interchangeably in this presentation. Don't let this confuse you.
  - **script = program**
  - **coding = programming**
  - **execute = run**
  - **evaluate = test**

Formatting

- Normal expository text is in this font.
- Text that represents what *you* type is shown like so:

```
perl myprogram.pl
```

- Text that represents computer output on the screen is shown like so:

```
File not found.
C:\>
```

- Things written in angle brackets should be replaced with the appropriate text:

```
perl <your program> -w
```

II. About Perl and programming

What is Perl?

- Perl is an **interpreted** programming language. (More on what that means soon.)
- Any programming language is essentially a human-friendly formalism for writing instructions for a computer to follow. These instructions are at some point translated into machine language, which is what the computer really "understands".

Programming languages vs. human languages

- Since we're all linguists, it's interesting to think for a moment about the differences between computer languages and "natural" languages:
  - Most programming languages do have recursive syntax, and thus can theoretically generate infintely complex structures, like human language.
  - Also, programming languages have "syntax" and "semantics", and something akin to word classes.
  - However, programming languages have extremely small lexicons, and very restricted syntactic rules.
  - Another big difference is that computer languages deal with very restricted domains compared to human languages, which are much more general and flexible.

Programming languages vs. human languages

- Importantly, computer programs cannot tolerate ambiguity—programs must be absolutely precise.
- Pragmatics doesn't really enter into the picture; a computer can't "figure out what you mean" in spite of what you say (i.e., no implicature).
- If there is a counter-example to this, it is probably Perl, which is extremely adept at "doing the right thing", with a limited amount of explicitness in the instructions.
- Also, Perl (like COBOL and some other languages) was designed to sound as much as possible like English, so it's relatively user-friendly. Relatively.

That said...

- One big difference between writing programs and writing other sorts of documents is that your programs must have **perfect** syntax, or they will not work. Every little comma and semicolon is critical.
- Therefore, in writing code, it is vital to be absolutely meticulous. Otherwise, you will weep and gnash your teeth and generally not have much fun.
- On the bright side, the syntax is actually really easy. Many, many times easier than in any human language.
- On the even brighter side, Perl is a very forgiving language that allows for multiple forms of expression. That means that you can say something in various ways, as long as your syntax is correct.

You and your computer

- I like to think of it this way:

  The computer is very energetic but very very stupid.

- The computer can do things incredibly fast and efficiently, but it can't do much without being told exactly how to do it.
  - Imagine the computer could get you a glass of water. Instead of saying "Get me a glass of water" (no need to be polite), you would have to say something like:

    Move to the door, travelling north-east. When you reach the door, locate the handle and rotate it counter-clockwise a quarter turn. Swing the door towards you, moving out of the way as it approaches. Then proceed through the the doorway and east to the kitchen. Locate the refrigerator...

Why write programs?

- There are many reasons to write your own programs. Some are more obvious than others.
  1) Sometimes you can't find pre-existing software to do what you need done.
  2) Sometimes it's easier and faster to do something yourself than to use pre-existing software.
  3) Sometimes you want a very high degree of control, one that you can't get with ready-made software.
  4) You don't want to have to hire someone else to work with your data, or otherwise cede control of your work.
  5) Programming can get you very close to your data: you really see them and think about them more than you might otherwise.

Why write programs?

6) Programming is an enjoyable mental challenge, like crossword puzzles or chess, provided you're not in over your head. It's a very creative activity.

7) Programming is **extremely satisfying** when it works. You discover a problem, imagine a solution, write a bunch of code, and the problem is solved. It offers a far greater sense of closure than most activities in our field.

8) Programming skills are reusable (languages are pretty similar), and programming can be useful in all sorts of domains, not just linguistic research.

9) Your friends will be *very* impressed.

Perl vs. other programming languages

- Most programming languages--such as C, C++, VisualBasic, etc.--are **compiled languages**.
  - To run a program, you create a text document with the code, run a **compiler** on it to convert it into machine code for your OS, and then run it.
- Perl in an **interpreted language**, like Java, Pascal, awk, sed, Tcl, or Smalltalk.
  - To run a program in such a language, you create a text document and tell the interpreter (or Virtual Machine) to run it as a program. The interpreter checks it, compiles it into machine code, and runs it.
  - The 2-step process of interpreted languages makes them slightly easier to work with. You can constantly check your code by running it after every change.

History of Perl

- Perl was designed in the mid 1980s by Larry Wall, then a programmer at Unisys, and self-described "occasional linguist".
- He combined useful features of several existing languages with a syntax designed to sound as much as possible like English.
- Since then, Perl has mushroomed into a powerful and popular language, with lots of modules contributed by the open-source community.
- Perl is designed to be flexible, intuitive, easy, and fast; this makes it somewhat "messy".
  - Perl has been called "a Swiss-Army chainsaw"
  - Perl is also known as "the duct-tape of the Internet"

Perl culture

- In many ways, Perl is a language for "hackers", not for computer scientists. Perl is disdained by people who prefer languages like C, which are more rigid and closer to machine code.
- Larry Wall's perspective is this:
  - "Perl is a language for getting your job done."
- Two other slogans that are key to The Perl Way:
  - "Easy things should be easy, and hard things should be possible."
  - TMTOWTDI: "There's More Than One Way To Do It."

How you get Perl

- One of the great things about Perl is that it is entirely **free**. It also runs on any platform out there. You can get the source code or a binary distribution from the web.
- The main Perl repository is CPAN: the Comprehensive Perl Archive Network
  - http://www.cpan.org
- There is a binary distribution for Windows that you can get directly from the makers, ActiveState:
  - http://activestate.com/
- Also check out these sites:
  - http://perl.com/   http://www.perl.org/

III. Perl basics

Executing a script: way #1

- There are two ways of running a Perl script: at the command line and from a text file.
- For very short scripts, you can just **type the whole program** at the command line and watch the computer run it.

```
C:\>perl -e "print 'Hello, World!'"
Hello, World!
C:\>
```

Executing a script: way #2

- For longer scripts (and ones you would like to reuse), you create a plain-text document with the program code and then tell the Perl interpreter to run this program, usually on the command line.
  - Save as "myprogram.pl":

```
#!/usr/bin/perl
print 'Hello, World!';
```

  - Then, on the command line:

```
C:\>perl myprogram.pl
Hello, World!
C:\>
```

Depending on your system, you might be able to run programs in various ways!
Try these out:

```
perl myprogram.pl
myprogram.pl
myprogram
```

Writing a script

- Perl scripts/programs are plain-text documents (generally ASCII), usually with the extension .pl, though this isn't strictly necessary.
- You can write scripts in any word processor, though I recommend that you **not** use MS Word or another processor that by default saves documents in some binary (i.e., non plain-text) format.
  - See the wiki for links to editor applications.
- It's also very helpful to use a constant-width (monospaced) font, such as **`Courier New`**, because then you will be able to line up your code easily.

Writing a script

- We generally start a Perl script with the "shebang" line, which looks something like one of these:

```
#!/usr/bin/perl
#!/usr/local/bin/perl
#!/usr/local/bin/perl5
#!/usr/bin/perl -w
etc.
```

very useful!

- This is not necessary on many systems, but when it is, it is, so it's a good habit to get into.
- The purpose of this line is to tell the server which version of Perl to use, by pointing to the location in the server directory of the Perl executable.

A first program

- Let's write a first program. Open your text-editor of choice and type the following:

```
#!/usr/bin/perl
print "Hello, World!\n";
```

- Then save the file as "hello_world.pl". Preferably, save all your scripts in the same directory. I call my directory "scripts". (I don't use "perl", because that where all the Perl files are.)
- Now we need to tell the interpreter to execute the script. We do this at the **command line**.
  - In UNIX, this is called a shell; in Windows, a command prompt; in Mac OS X, a Terminal window.

Getting to the command line

- In Windows: Click on **Command Prompt**. To get there, you may have to look under **Start->All Programs->Accessories**
- In MAC OS X: Click on **Terminal**. To get there, you may have to look under **Applications->Utilities**
- In UNIX: If you're running UNIX, you *are* in a shell, provided you're logged on. The same goes for X-Window or a similar program.

Navigating in the shell

- Note: The **prompt** may look different ways, depending on your system. Usually, it tells you what directory you're in. Note that it's often customizable. Here are some examples:

```
C:\scripts>
/Gregory/scripts$
[Some-Computer:~/scripts]gregory%
```

- You really only need to know one command to get around in the shell: cd. But it also helps to know ls (dir in Windows) and pwd.
  - Move to dir 'scripts' move up one dir

```
cd scripts          cd ..
```

Basic Perl syntax

- Perl thinks of "lines" differently from humans. For us, the most important thing is the carriage return; for Perl it is the semicolon.
- All lines of code in Perl must end with one of two things: a **semicolon** or **curly braces**:

```
print "Hello.";
sub {print "Hello."}
```

- A line of code, called a **statement**, is basically a single instruction to the computer; it says, "Do **x**." It can extend over more than one line of the page.
- Anything in { } is called a **block**. A block can contain several statements.

Comments

- Any line that begins with the character **#** is treated by Perl as a **comment** and is ignored.
- This means that you can put lots of comments for humans to read in your code, without affecting the program. This is a **very good idea**.
- The # can actually come at any point in the line: whatever comes after it will be ignored. But in that case what comes before it had better be a valid line of code!

```
# This next line sorts alphabetically
@names = sort (@names);  # () optional
```

Quote marks

- Perl makes use of three basic kinds of quote marks: single (' '), double (" "), backquotes (` `)
- You will usually want to use double quotes. As a general rule, use the others only if you have to.
- Double quotes allow **variable interpolation**. That means that this code:

```
$name = "Alejna";
print "Hello, $name!\n";
```

These are the quotes to watch!

- will print this:

```
Hello, Alejna!
```

Quote marks

- Single quotes will prevent interpolation. That means that this code:

```
$name = "Alejna";
print 'Hello, $name!\n';
```

- will print this:

```
Hello, $name!\n
```

- Backquotes (` `) run an external program and grab the output. You will rarely, if ever, use these.

Backslash interpolation

- Perl has a number of special characters that have a particular meaning within double quotes an in regular expressions. Here are two important ones:

```
\n    means "newline"
      a.k.a. "carriage return"
\t    means "tab"
```

- So, here's what we can type and what we'll get:

```
print "Name:\tBecky\nEyes:\thazel\n";
Name:   Becky
Eyes:   hazel
```

IV. Variables

Data types in Perl

- Most data is basically either a **string** or a **number**, or a bunch of strings or numbers.
- A **string** is character data: "froggy", "11-30-04".
  - Always write strings within quote marks, as above.
- A **number** is exactly what you think it is.
- Note, however, that a given piece of data that looks like, say, 300, can be treated like a number (e.g., 300 + 1) or like a string (e.g., through concatenation: 300a).
- Unlike other programming languages, Perl takes a very relaxed attitude toward this distinction. It basically looks at what you're trying to do and treats the data as the appropriate type.

Variables

- The concept of the **variable** is fundamental in all programming languages. A variable is essentially a placeholder for a value.
- We know variables from basic algebra:
  - In **x + 1 = 5**, **x** is a variable.
- Use of variables in programming differs a bit from algebra: We spend a lot of time creating variables, assigning values to them, changing the values, and reading the values.
- Virtually all data is stored in variables. We manage this information through operators (like +) and functions (like print).

Scalars

- In Perl, the three most important types of variables are the **scalar**, the **array** and the **hash**.
- A **scalar** is a variable that holds a single value. Scalar names begin with **$**:

VARIABLE **VALUE**

```
$name = "Aisha";
$age  = "20";
```

Arrays and hashes

- There are two kinds of variables that hold multiple pieces of data: **arrays** and **hashes**.
- An **array** is a variable that holds multiple values in series. Array names begin with **@**:

```
@names = ("Howard", "Leslie", "Bob");
```

- A **hash** is a variable that holds pairs of data. Hash names begin with **%**:

```
%traits = ("name" => "Howard",
 "age" => "30", "eyes" => "brown");
```

Variable names

- The name of a variable (whether scalar, array, or hash) begins with the special type-indicating character (**$**, **@**, or **%**, respectively), and then can contain any combination of letters, numbers, and underscores.
  - However, the first character after the **$**, etc. must be a letter or underscore, not a number.
  - Also, case IS distinctive.
  - Examples of valid names:

    `$time_of_arrival, $Time_of_Arrival`

    `$timeofdeparture, $TOD`

    `$Time4U2Go` (if you're childish)

Good naming practice

- When you choose variable names, choose illustrative ones, not obscure ones:

```
BAD                      GOOD
 ↓                        ↓
$x = "John";      $first_name = "John";
$y = "Bates";     $last_name = "Bates";
```

- I make a point of using singular names for scalars and plural names for arrays and hashes:

```
$person = "Christine";
@people = ("Christine", "Jean");
```

Two kinds of operators

- The important one to watch out for is the "smooth operator". No, just kidding.
- A useful distinction to keep in mind is that between **assignment operators** and **comparison operators**. The former **give** a value to a variable, and the latter **test** the value of a variable.
  - To set the value of a variable:

```
$limit = 100;
```

If this were a speech act, it would be a **performative**.

  - To test the value of a varable:

```
if ($number == 100) {print "Limit!"}
```

If this were a speech act, it would be a **question**.

Working with scalars

- We assign a value to a scalar using the assignment operator **=**, whether we are dealing with strings or numbers.

```
$name = "Eliani";
$grade = 100;
```

- We can perform math on a scalar if it makes sense to do so:

```
$grade = ($grade * 30)/100;
```

- We can perform **concatenation** on strings with the `.` operator:

```
$name = $first_name . " " . $last_name;
# now $name is, e.g., "Mary Hughes"
```

Working with arrays

- We can assign values to arrays in various ways. The most straightforward is to use parentheses, which create a "list context":

```
@grades = ("98", "84", "73", "89");
```

- The values in an array are numbered (or **indexed**), **starting with 0**, rather than 1. We can access any value in an array by referring to its numerical index in square brackets:

```
print "$grades[0] and $grades[2]";
98 and 73
```

- Note that we use **$** rather than **@**, because here we're referring not to the whole array, but to one **element** in it, which is a scalar.

Working with arrays

- When we print an array, it makes a difference whether we put it in double quotes or not; if we do, we get spaces between the elements.

```
@VOTs = ("400", "378", "352");
print @VOTs;
400378352
print "@VOTs";
400 378 352
print '@VOTs';   # to review
@VOTs
```

Input

- Perl needs data to interact with. This data typically comes from one of three sources: keyboard input, files, or the system (the server).
  - We're only going to cover the first of these today.
- Input from the keyboard is so standard that we call it "standard input", abbreviated **STDIN**.
- Here's how we get some input from the keyboard:

```
$input = <STDIN>;
```

- This assigns to **$input** whatever the user typed. Unfortunately, this also includes the newline from when they hit "Enter", so we take that off like so:

```
chomp ($input); # the () are optional
```

V. Conditionals and loops

Conditional statements: if

- Of course, the power of programs comes from their input to react differently to different data. The **conditional statement** is the key to this.
- The if statement has two parts: a **condition**, which is evaluated, and a **block**, which is executed if the condition evaluates to be true.

```
if ($price > 100) {
    print "Too expensive!\n";
}
```

Conditional statements: elsif

- An **if** statement can optionally be followed by an **elsif** statement. This is evaluated *after* the if statement **only if the previous condition evaluated (the if statement's) was false**. The two blocks will never both be exectuted.

```
if ($price > 100) {print "expensive"}
elsif ($price < 10) {print "bargain!"}
elsif ($price < 40) {print "cheap"}
```

- Again: If the **if** statement evaluates to true, the **elsif** statements are just skipped. Similarly, if the first **elsif** evaluates to true, the second is skipped.

Conditional statements: else

- An **else** statement is a special kind of **elsif** that covers all remaining conditions. In linguistics, this is known as "the elsewhere condition".

```
if ($grade < 60) {
    print "Failing grade!: $grade\n";
}
else {print "Grade is $grade.\n"}
```

- Note that there is no condition to be evaluated in an **else** statement, because it's the "garbabe can" category and is always true.

Loops

- A major reason we use computers is that they can do the same thing many times without getting tired or needing coffee. We get them to repeat actions using **iterative** or **looping constucts**.
- There are three main types of loop statement, which are all related but specialized for different uses: **for**, **foreach**, and **while statements**.

'while' loops

- **While statements** are the least complex of the three types, so we'll start there. They are similar to conditional statements, because they have a condition and a block. If the condition evaluates to true, the block is executed. Then, the condition is evaluated again, and **if it's still true**, the block is executed again, and so on.

```
while ($money < 1000) {
    $pay = Get_Paycheck();
    $money = $money + $pay;
}
print "Ok, we can pay the rent!\n";
```

This represents some user-defined subroutine, which presumably checks for more money and returns the value as **$pay**.

'until' loops

- **Until statements** are a special type of **while** statement in which the block is executed **until the condition evaluates as true**. It's equivalent to "while NOT (condition)...".

```
until ($money < 1000) {
    $money = Eat_Out ($money);
}
print "Time to eat noodles!\n";
```

'for' loops

- **For statements** are very useful, because they allow you to do something a preset number of times, or until a condition becomes true.
- Their structure is more complex than a while statement's:

```
for ($i = 1; $i <= 100; $i ++) {
   print "We've reached $i\n";
}
We've reached 1
We've reached 2
...
We've reached 100
```

'for' loops

This sets the initial value of the variable: its value on the first pass through the loop.

This sets the test that determines when to exit the loop: as long as this is true, the loop will be rerun.

This sets how the variable will be incremented on each pass through the loop.

```
for ($i = 1; $i <= 100; $i ++) {
   print "We've reached $i\n";
}
```

```
We've reached 1
We've reached 2
...
We've reached 100
```

'foreach' loops

- **Foreach statements** allow us to perform the same operations on every element of an array or hash in series. They temporarily assign the value of each element to a scalar that we can work with.

```
foreach $grade (@grades) {
   if ($grade < 60) {
      print "Failing grade!: $grade\n";
      $fail_tally ++;
   }
}
print "We have $fail_tally failing
 students!\n";
```

Exiting the loop

- Sometimes you want to exit the loop via a method other than the standard one. We do this using the statements **next**, **redo**, and **last.**
- **next** means "skip over everything else in the block, increment the counter, and evaluate the conditional again."
- **redo** means "skip over everything else in the block and evaluate the conditional again, without incrementing the counter."
- **last** means "exit the block and never come back."

Exiting the loop

- Here's an example that uses both **next** and **last**:

```
foreach $student (@students) {
    if ($student eq "END_REGISTERED") {
        last;
    }
    elsif ($student eq "Silber"){
        next;
    }
    else {
        $grade = Check_Grade ($student);
    }
    print "$student: $grade\n";
}
```

VI. Exercises

Exercise 1

- Here is a **command line** script that print "Hello, world!" (Windows version—others may want to use single quotes):

```
perl -e "print qq/Hello, World!\n/;"
```

  - Note that inside the " ", we use qq/ / instead; this is because we can't nest double-quotes. These are equivalent:

```
"hello" = qq/hello/
```

- Write a script on the command line that greets you instead of the world.
- Then add some code so that it prints two messages on two lines.

Exercise 2

- Here is a program (to be written in a text file) that prints again on the screen whatever you type. Modify it so that it asks for your name and greets you by name.
  - Name: Call it 'greet.pl'.

```
#!/usr/bin/perl -w
print "Type something: ";
$something = <STDIN>;
chomp $something;
print "$something\n";
```

This is a terrible variable name. Change it to something more appropriate!

Exercise 3

- Write a program that, given the base form of a verb (through STDIN), outputs the correct 3rd-person singular form.
  - Name: Call it 'conjugate.pl'.
  - NOTE: Don't forget about irregular forms!
  - NOTE: You can leave out all negative forms.
  - NOTE: You can also ignore forms like "watch**es**". (We'll do those ones once we get to regular expressions)
- HINT: You will probably want to use the following things:

```
if    elsif    else    eq    .
```

VII. More stuff

Opening files

- Perl makes it fairly easy to load data from files. To do this we use the **open** and **close** functions, and the "angle-operator" **< >**.
- Here's a simple example:

```
$input_file = "bigoldfile.dat";
open (INPUT, "$input_file");
while ($line = <INPUT>) {
      print "$line";
}
close (INPUT);
```

Note that this prints to STDOUT, or the screen.

- INPUT is what's known as a **filehandle**. It's a temporary name for a file. STDIN is a filehandle too.

Creating files

- We can use the same commands to output to a file that the program creates.

```
$input_file = "bigoldfile.dat";
$output_file = "output.txt";
open (INPUT, "$input_file");
open (OUTPUT, ">$output_file");
while ($line = <INPUT>) {
   print OUTPUT "$line";
}
close (INPUT);
close (OUTPUT);
```

This **>** makes all the difference. It means that this file is being written to, not read from.

The 'split' function

- The function **split** enables us to easily transform a scalar to an array, by splitting the scalar up on spaces or another character.

```
$sentence = "Sue and I split up.";
@words = split(/ /, $sentence);
print "$words[4]\n";
up.
```

That's a space.

- We can also split on commas, etc.

```
$list = "Eenie, meenie, miney, moe";
@words = split(/,/, $list);
print "$words[3]\n";
 moe
```

Counting elements in an array

- We often want to know how many elements are in an array. There are three ways to get this info.
  - The function **scalar**:

```
@people = ("Moe", "Larry", "Curly");
print scalar(@people) . "\n";
3
```

  - Use of "scalar context":

```
$count = @people;
print "$count\n";
3
```

  - Use of **$#**, which gives the **last index** of an array:

```
print "$#people";
2
```

Finding the length of a string

- The function **length** can be used to find to length in characters of a scalar.

```
$string = "abcdefghij";
$howbig = length($string);
print "$howbig\n";
10
```

The 'sort' function

- You can sort the elements of an array or the keys of a hash with the function **sort**. BUT: By default, it sorts both strings and numbers alphabetically!

```
@array = ("Betty", "Cathy", "Abby");
@array = sort(@array);
print "@array\n";
Abby Betty Cathy
```

  - But watch out:

```
@array = ("3", "40", "24", "100");
@array = sort(@array);
print "@array\n";
100 24 3 40
```

Sorting array keys

- A very common type of loop makes use of the functions **sort** and **keys**. The latter yields all the keys (not the values) in an array.

```
%signs = ("Frank" => "Capricorn",
  "Amanda" => "Scorpio");
foreach $person (sort keys %signs) {
  print "$person: $signs{$person}\n";
}
```

```
Amanda: Scorpio
Frank: Capricorn
```

Adding elements to arrays

- There are two ways of adding new elements to existing arrays.
- If we know the index we want the element to have, we can do this:

```
@numbers = ("210", "450", "333");
$numbers[3] = "990";
```

- If we simply want to add an element to the end of an array, we can use **push**:

```
push(@numbers, "990");
```

- In either case, @numbers is now:

```
210 450 333 990
```

Taking elements out of arrays

- The reverse of **push** is **pop**, which takes removes the last element from the array:

```
@numbers = ("210", "450", "333");
$last = pop(@numbers);
print "$last\n";
333
```

- Note that this is different from saying

```
$last = $numbers[2];
```

  because this doesn't remove the element from the array. After **pop**, the array will have only 2 elements!

That's all!