Introduction to Perl

- Today's lecture: overview of Perl
  - goals/philosophy/history
  - basic program structure
  - control structures
  - data structures
  - I/O
- Next lecture:
  - pattern matching
  - Perl scripts

Perl References

- Some O'Reilly books:
  - Schwartz, et al., Learning Perl (1997)
  - Srinivasan, Advanced Perl Programming (1997)
  - Many others on specific applications, e.g. databases, CGI, ...
- Websites (documents, libraries)
  - www.perl.org (Perl Institute)
  - www.perl.com (commercial)

History

- Perl = Practical Extraction and Report Language
- History: tool for generating reports
  - processing large amounts of data, extracting and reformatting information
- Roots: Unix tools (awk, sed) and shell scripts
- Developed late 1980s by (and for) Unix systems administrators
- Now public domain software with a life of its own...

Scripts vs Programs

- The term "script" implies "controlling other programs"
  - also known as "component programming"
  - scripting languages are also known as "glue languages"
- Scripts can
  - launch other programs
  - process and reformat output from those programs
  - act as an interface to a remote database
  - generate web pages (when launched by a web server)

  "Perl is the duct tape of the internet"
  - Hassan Schroeder, Sun webmaster

Other Scripting Languages

- Some other scripting languages to look into
  - Python
  - Tcl (especially for GUI construction)
  - Ruby
- See links and articles on class Web page

Stand-alone Programs

- Perl can be used for stand-alone applications, not just scripts
  - Rich control structures
  - Data structures (lists, arrays, pointers)
  - Classes and object-oriented programming
- But:
  - no type checking (serious drawback for large programs)
  - inefficient
Program Structure

- Programs are collections of statements
  - assignment
  - control (for, if, ...)
  - I/O
- There are no variable declarations
  - variables are introduced when you assign them values
- Statements can be collected into subroutines
  - externally defined subroutines can be loaded dynamically, e.g. from a library

Hello, World in Perl

- The complete text of the "hello" program:
  ```perl
  #!/usr/bin/perl
  # Hello world in Perl
  print "Hello, world!
";
  ```

Notes:
- The O/S sees this program is to be executed by Perl
- The Perl system compiles the file into an intermediate form
- Execution begins with the first statement in the program

Perl Is Not Interactive

- Some scripting languages can be used interactively, e.g.
  ```python
  $ python
  >>> print "Hello, world"
  Hello, world
  >>> "0"
  ```
- The Perl compiler reads the entire program before it begins execution
  - syntax checking
  - translation to internal form

Variables Do Not Have Types

- Scalar variables have names beginning with $
- Variables are untyped
  - a scalar can hold a string, integer, floating point value, etc
  - a variable can hold different types at different times
- More accurately:
  - all variables are strings
- When used as a number, the system converts the string into the corresponding value

Variables (cont'd)

- Example (where . is the string concatenation operator):
  ```perl
  $n = "3";  # $n is a string
  print "n = 
";  # use $n as int
  $m = $n * 2;
  print "m = 
";
  $n .= 'little pigs
';  # it's still a string
  print "n = 
";
  ```
- Note that $n always holds a string, but the string is interpreted as a number in the arithmetic expression

Undef

- Perl has a special value called undef
  - similar to NULL in C and Java
- New variables have the value undef until they are assigned a value
- Depending on the context, undef may look like (or be interpreted as) 0, "", or false
```perl
$x = 3;
print 'x = "x" y = "y"
produces:
  x = "x" y = 
```
Program Structure Revisited

- As mentioned previously, programs are sequences of statements.
- Variables are not defined until you assign them a value.
- Subroutines are defined when:
  - the compiler sees a subroutine definition, or
  - the program loads a subroutine from a library at runtime.
- Consider the order of events in the following version of the “hello” program.

Hello Again

```perl
#!/local/bin/perl
print "Hello, world!
"; # Not a syntax error; runtime error printed after "Hello, world!"
bogus("world");
```

Consider the order of events in the following version of the “hello” program.

Hello Again

```perl
#!/local/bin/perl
print "Hello, world!
";
greeting("world");
exit;
sub greeting { # How local variables are declared and parameters are passed (more later)
  my $w = shift;
  print "Hello, $w
";
}
```

Consider the order of events in the following version of the “hello” program.

Hello Again

```perl
#!/local/bin/perl
print "Hello, world!
";
#
bogus("world");
greeting("world");
exit;
$name = "John";
sub greeting { # How local variables are declared and parameters are passed (more later)
  my $w = shift;
  print "Hello, $w, my name is $name!
";
}
```

Consider the order of events in the following version of the “hello” program.

Hello Again

```perl
#!/local/bin/perl
print "Hello, world!
";
#
bogus("world");
greeting("world");
exit;
$name = "John";
sub greeting { # How local variables are declared and parameters are passed (more later)
  my $w = shift;
  print "Hello, $w, my name is $name!
";
}
```

Consider the order of events in the following version of the “hello” program.

Hello Again

```perl
#!/local/bin/perl
print "Hello, world!
";
#
bogus("world");
greeting("world");
exit;
$name = "John";
sub greeting { # How local variables are declared and parameters are passed (more later)
  my $w = shift;
  print "Hello, $w, my name is $name!
";
}
```

Conditional Execution

- if statements are similar to C/C++/Java.
- To test several conditions, use elseif.
  ```perl
  if ($x == 10) {
    print "x = 10\n";
  }
  elsif ($x == 11) {
    print "x has increased\n";
  }
  else {
    print "error\n";
  }
  ```
Alternate Syntax

- Perl has other ways of writing conditional statements
- You can put a single statement before the if:
  ```
  print "hello\n" if $expr;
  ```
  Note no braces, parens are optional
- You can use unless as the opposite of if:
  ```
  unless ($expr) {
    print "a string\n";
  }
  ```

Boolean Expressions

- Recall that all values in Perl are strings
- In an expression – "0" is false
- the empty string " " is false
- Every other string is true
- "0" is produced by arithmetic operations, e.g.
  ```
  print "always" if ($x - $x);
  ```
- "0" is also produced by comparison operators, e.g.
  ```
  print "greater" if ($x > $y);
  ```

Loops

- for and while statements are as in C/C++/Java
  ```
  for ($i = 0; $i < 10; $i++) {
    $sq = $i * $i;
    print "$i squared = $sq\n";
  }
  ```
  ```
  while ($sum < 100) {
    $sum += $i * $i;
    $i += 1;
  }
  ```

Alternate Syntax

- until ($expr) { # iterate while $expr 0
  -
  }
  ```
  do {
    # test condition after
    -
  } while $expr;
  ```
  ```
  do {
    -
  } until $expr;
  ```

Subroutine Declarations

- Introduce procedures or functions with the sub keyword
  ```
  sub foo {
    # body of function
    return 1; # optional return statement
    -
  }
  ```
- Notes:
  - no parameter list in subroutine header
  - subroutines can be called as in C/C++/Java, in contexts that use values or as simple statements:
  ```
  foo(3);
  $x = foo(3);
  ```

Local Variables

- Variables created in a subroutine are global, by default
  ```
  foo();
  print "x = $x\n";
  exit;
  ```
  ```
  sub foo {
    $x = 10;
  }
  ```
- Restrict the scope of a variable using the keyword my
  ```
  sub foo {
    my $x = 100;
  }
Parameters

- Parameters are passed by value, as in C/C++
- The system collects the parameter values in a list and passes the list to the subroutine
- The subroutine accesses the parameter values using list operations
- More on lists later; for now:
  - $[0]$ is the first parameter
  - $[1]$ is the second parameter
  - etc
  (cont'd on next slide)

Parameters (cont’d)

- Idiom:
  - declare a local variable for each parameter
  - use the shift operator to "pop" elements from the left side of the
    parameter list
  - with no arguments to shift, the compiler assumes you want to
    shift values from the parameter list
  
  sub product {
    my $x = shift;
    my $y = shift;
    return $x * $y;
  }
  print "2 * 6 = ", product(2,6), "\n";

Arrays

- So far we’ve seen only scalar (single value) variables
  - Scalar names start with $
  - Array names start with @

- Array variables hold lists, which are collections of values

- Examples:
  - @A = (1, 2, 3);  # an array with 3 items
  - @B = ($here, $there);

Arrays (cont’d)

- Array values can be inserted into strings just like scalar values:

Array Elements

- Use the [ ] operator to access contents of an array
  - Since arrays hold scalars, use $ when referring to array
elements

  @teams =
  ("ducks","beavers","huskies","cougars");

  $n = @teams;
  for ($i = 0; $i < $n; $i++) {
    print "teams[$i] = @teams[$i]\n";
  }

  There are $n teams: @teams\n";

Array Size

- Arrays grow dynamically, as needed
  - RA = ("alpha", "beta", "gamma");
  - RA[1] = "beta";
  - RA[2] = "gamma";
  - RA[3] = "omega";
  - RA[4] = "omega";
  - RA[5] = "omega";
  - A new 84 elements; RA[34] are undef

- To determine the number of elements in an array:

  for ($i = 0; $i < @A; $i++) {
    print "A has $n elements\n";
  }

Array Strings

- Array values can be inserted into strings just like scalar
type:

  @teams =
  ("ducks","beavers","huskies","cougars");
  $n = @teams;
  $n = number of teams

  print "There are $n teams: @teams\n";
  There are 4 teams:
ducks beavers huskies cougars
List Operations

- There are several built-in operations on lists:
  - `push(@A,$n);` append a value on the right
  - `$n = pop(@A);` remove a value from the right
  - `unshift(@A,$n);` append a value on the left
  - `$n = reverse(@A);` copy of @A in reverse order
  - `@B = @A;` copy of @A
  - `@B = sort(@A);` copy of @A with items sorted

Iterating Over Lists

- There is a special control construct for lists:
  ```perl
  foreach $i (0 1 2 3) {
    $dna .= $code[$i];
  }
  ```
  ```perl
  print "Pac-10 Standings\n";
  foreach $t (@teams) {
    print "$t\n";
  }
  ```

Associative Arrays

- Perl has an associative array type – also known as a "hash" (from "hash table")
- Similar to C++ STL associative map
- Like regular array, except index terms can be any scalar values (i.e., strings)
  ```perl
  $greek("a") = "alpha";
  $greek("b") = "beta";
  ```
- Note the use of braces {} as the index operator
- Read about hashes in the reference texts...

I/O

- To send output to a file, use a filehandle
  ```perl
  open(IN, "ecoli.gbs");
  ```
  ```perl
  open(OUT, "> ecoli.ptt");
  ```
  ```perl
  open(OUT, ">> ecoli.ptt");
  ```

Opening a Filehandle

- To create a filehandle variable and attach it to a file, use the open function
  ```perl
  open(IN, "ecoli.gbs");
  ```
  ```perl
  open(OUT, "> ecoli.ptt");
  ```
  ```perl
  open(OUT, ">> ecoli.ptt");
  ```

Output

- Use the print statement to send characters on an output stream via a filehandle
  ```perl
  print OUT "$gene $gi $start..$end\n";
  ```
- Note that print with no filehandle is just the equivalent of print STDOUT...
Input

- An input filehandle name enclosed in angle brackets is an expression that means "read the next line from the file"
- The expression returns undef (i.e. false) when there is no more input
- The line will include the newline character at the end

Input (cont’d)

- A convenient way to remove a newline when you don’t want it:
  - chop($line);
- A better way: make sure the last character is a newline, otherwise don’t remove it:
  - chomp($line);

Template for File Processing

- A common idiom for reading all the lines in a file

```perl
$filename = shift; # get argv[0]
open(IN,$filename) || die "Can’t open file $filename\n";
while ($line = <IN>) {
  # remove newline
  chomp($line);
  print "line = \"$line\n\";
}
```

Perl Scripts

- So far we haven’t seen much that shows how to use Perl as a glue language
- Coming up in future lectures:
  - interacting with the file system (file tests, renaming, removing, etc)
  - launching another program and sending it data
  - reading data from other programs
  - examples: downloading genome data from NCBI