# Scripting Languages
# Perl Basics

Course: 67557

Hebrew University

Lecturer: Elliot Jaffe – אליוט יפה

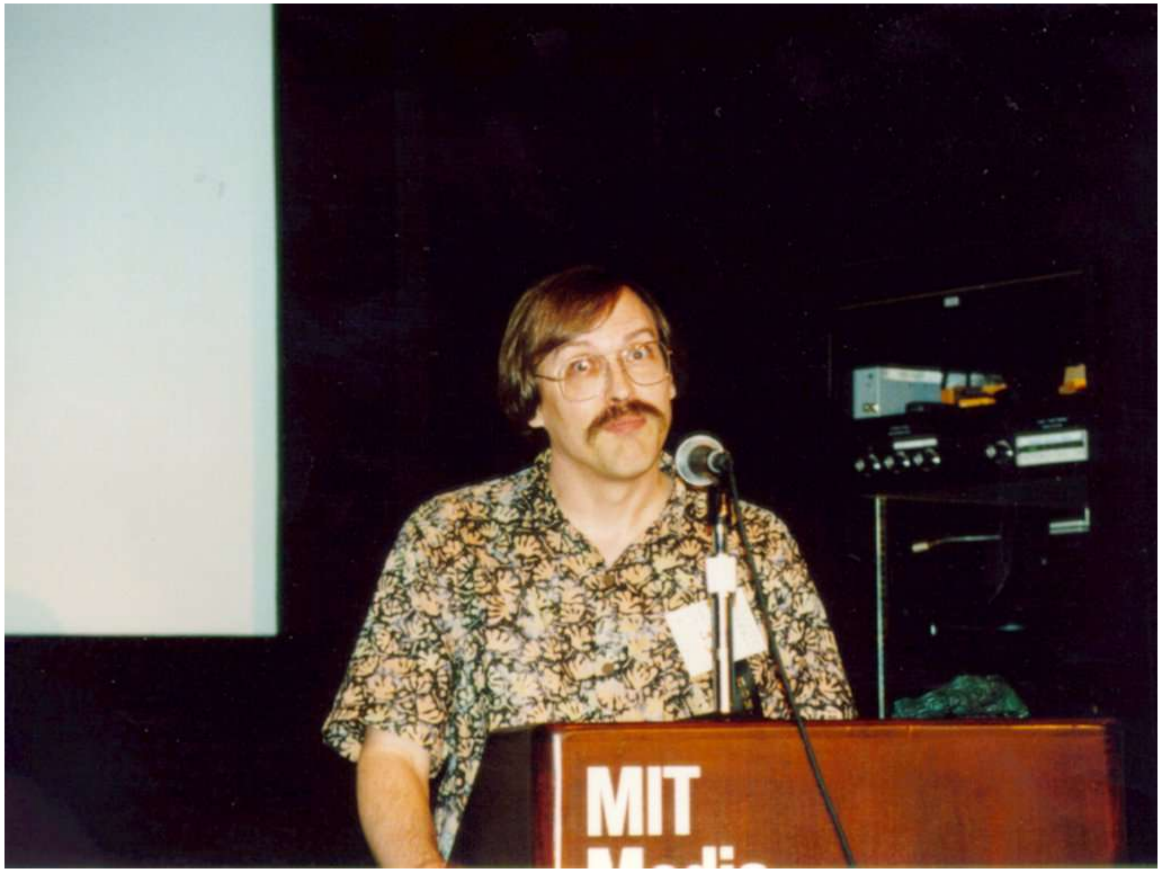
## FMTEYEWTK

- Far More Than Everything You've Ever Wanted to Know
- Perl
  - Pathologically Eclectic Rubbish Lister
  - Practical Extraction and Report Language
- The Perl motto is TMTOWTDI
  - ``There's more than one way to do it."

# TMTOWTDI

There's more than one way to do it

# Larry Wall



# Data Types

Values of any type may be stored in a variable

```
$myVar = 'c';            # Character
$myVar = "Hello World!"; # String
$myVar = 42;             # Integer
$myVar = 3.14159;        # Float
```

# Strings

- Double Quotes strings are interpolated
- Single Quoted strings are static

```
$myA = 'a';
$myB = "this is $myA string";
$myC =
    'this is another $myA string';
```

# Automatic Type Conversion

Conversion happens automatically

| From | To | Conversion |
|---|---|---|
| “42” | 42 | String to Integer |
| 42 | “42” | Integer to String |
| “3.14159” | 3.14159 | String to Float |
| 3.14159 | “3.14159” | Float to String |
| “c” | ‘c’ | String to Char |
| ‘c’ | “c” | Char to String |

## Perl Data Structures

- Scalar
- Arrays of Scalars
- Associative Arrays of Scalars – Hashes

- Variables are identified by sigil
  - a preceding dereferencing symbol which tells Perl what kind of variable it is

## Scalars

```
$myVar = 3.14159;
```

- Sigil: $
- Holds a single scalar value of any type
- Undefined variables have the value `undef`

```
 defined(undef) == FALSE
```

## Notes

- Notice that we did NOT have to
  - declare the variable before using it
  - define the variable's data type
  - allocate memory for new data values

- Is this a good thing?

# Arrays of Scalars

```
@myVar = (3, “foo”, ’c’);
```

- Sigil: @
- A list of any type of scalar values

```
$myVar[0] is 3
$myVar[2] is ’c’
```

- Access to array elements is by integer index (zero based)

# More on Arrays

- Creating and setting an element

```
$foo[3] = "dog";
```

- Assigning multiple element values

```
$foo[1,3] = ( "bear", "dear" );
```

- Adding new elements

```
@foo = ( @foo, "elk" ); # Append
@foo = ( "ace", @foo ); # Prepend
```

## Sizes of Lists

- Two approaches yield two different results

```
@foo = ( "apple", "bat", "cat" );
```

- Get the number of elements contained in the list

```
$size = scalar( @foo ); # Yields 3
```

- Get the index for the last element contained in the list

```
$size = $#foo;    # Yields 2
```

# Lists as LHS values

- You can use lists on the left-hand side of an assignment "=" operator

```
($first, $last) = ("John", "Moreland");
```

- Perl uses "greedy" assignment for L-Values. Here, $d is left untouched

```
($a,$b,@c,$d) = ("a","b","c","d","e");
```

- But, here, "e" is simply not assigned

```
($a,$b,$c,$d ) = ("a","b","c","d","e");
```

# Range Operators

- Perl defines a special list range operator ".." to simplify the specification of such a range
- The ".." operator is used as an infix operator placed between any two scalar values
- Perl will interpolate the (quantized "in between") values automatically

```
( 1..5 ) # Yields ( 1, 2, 3, 4, 5 )
( 1.3..6.1 ) # Yields ( 1.3, 2.3, 3.3, 4.3, 5.3 )
( 2..6, 10, 12 ) # Yields ( 2, 3, 4, 5, 6, 10, 12 )
( "a".."z" ) # Yields ( "a", "b", "c", ..., "z" ) Nice.
( "a1".."e9" ) # Yields ( "a1", "a2", ..., "e9" ) Wow!
```

## Example

- Put “bat” between (“ape”, “cat”)

```
@foo = ( "ape", "cat" );

$foo[2] = "cat";
$foo[1] = "bat";
```

or

```
$a = shift( @foo );
unshift( @foo, $a, "bat" );
```

# Builtin List functions

| | | |
|---|---|---|
| `pop` | Remove last item | `$a = pop(@list);` |
| `push` | Insert item at end | `push(@list, $a);` |
| `shift` | Remote first item | `$a = shift(@list);` |
| `unshift` | Insert item at front | `unshift(@list, $a);` |
| `splice` | Remove | `@olditems = splice(@list, $pos);` |
| `splice` | Remove n items | `@olditems = splice(@list, $pos, $n);` |
| `splice` | Remove and insert | `@olditems = splice(@list, $pos, $n, @newitems);` |

# List processing

| | | |
|---|---|---|
| `grep` | search | `@code = grep !/^#/, @lines;` |
| `join` | Insert item at end | `$str = join ':', @words;` |
| `split` | Split string | `@list =`<br>`  split /[-,]/, "1-10,20";`<br>`  # (1, 10, 20)` |
| `reverse` | Reverse list | `@new = reverse @old;` |

# Associative Arrays - Hashes

- Associative arrays are hash tables
- Sigil: %

- Stored as unordered lists of (key, value) pairs
- Any scalar value can be used as a key

## Hash examples

- You can initialize an associative array much like a list

```
%days = ( 'M', "Monday", 'T', "Tuesday");
```

- The scalar **$** plus {} references one element (note: key is any scalar value)

```
$days{'W'} = "Wednesday";
```

- Any scalar data type can be used for the key or the value

```
$myConst{"PI"} = 3.14159;
$hg{42} = "life, the universe, and me";
```

# Builtin Hash Functions

```
%days = ( 'M', "Monday", 'T', "Tuesday" );
```

- The **keys()** function returns a list of keys

```
@letters = keys( %days );
            # Yields ( 'M', 'T' )
```

- The **values()** function returns a list of values

```
@names = values( %days );
            # Yields ( "Monday", "Tuesday" )
```

- The **delete()** function removes a Key-Value pair

```
delete( $days{'M'} );
            # Yields ( 'T', "Tuesday" )
```

- The exists() function checks if a key exists in this hash

```
exists( $days{`W'} );   # Yields False (0)
```

# Example

- Sort and count unique lines

```
while ( chop( $line = <STDIN> ) ) {
  $unique{$line} += 1;
}

foreach $key ( sort keys %unique ) {
  print "$key = $unique{$key}\n";
}
```

# Subroutines

- **Defining**

```
sub MyFunction {
  # your code goes here
  return $value;  # optional
}
```

- **Calling**

```
&MyFunction; # if not yet seen
MyFunction;  # if seen
```

## Subroutine Parameters

- **Calling a function**

```
&MyFunction;
&MyFunction();
&MyFunction($arg1, $arg2);
&MyFunction($arg1, $arg2, @list1);
```

- **The & is optional and deprecated**

```
MyFunction($arg1, $arg2);
```

# Subroutine Parameters

- This is probably the ugliest thing in Perl!
- Parameters are stored in the variable `@_`;

```
sub MyFunction {
  ($arg1, $arg2, @list) = @_;
  $arg1 = $_[0];
}
```

  - Parameters are passed by value unless otherwise specified

## Subroutines

- What happens here?

```
$a = 1;
$b = 2;
@c = (3,4);
sub MyFunction {
  ($arg1, @list, $arg2) = @_;
}
MyFunction($a, @c, $b);
```

## Scoping

- By default, all variables are GLOBAL
- Perl support lexical and dynamically scoped variables
  - Lexical: variable is defined within the textual block
  - Dynamic: variable is defined to all functions called within this block

# Global Scoping

```
$a = “foo”;
sub global {
  ($arg1) = @_;
   print “in global arg1 = $arg1 \n”;
   nested;
}
sub nested {
   print “in nested arg1 = $arg1 \n”;
}
global($a);
print “outside arg1 = $arg1 \n”;
```

# Lexical Scoping

```
$a = "foo";
sub lexical {
  my($arg1) = @_;
  print "in lexical $arg1 \n";
  nested;
}
sub nested {
  print "in nested arg1 = $arg1 \n";
}
lexical($a);
print "ouside arg1 = $arg1 \n";
```

## Command line

- Two variables provide access to command line arguments
- Slightly different from the C conventions

| Variable | Contents |
|---|---|
| `$0` | Script name |
| `$ARGV[0]` | First arg |
| `$ARGV[1]` | Second arg |

## Flow Control

- No main function
- Statements are executed as they are encountered in the file
- Subroutines are defined but not executed
- `exit()` leaves the program

## Flow Control

- Standard if-elsif-else blocks

```
if ( $colour eq "red" ) {
  print "hot\n";
} elsif ( $colour eq "blue" ) {
  print "cold\n";
} else {
  print "warm\n";
}
```

## Flow Control

- C style :? shortcuts

EXPR ? EXPR_IS_TRUE : EXPR_IS_FALSE

```
$happy = 1;
print $happy ? “good" : “bad";
```

## One line conditional

- Often used shortcut for if-then (then-if)

```
$happy = 1;
$good = 1 if $happy;
$bad = 1 if ! $happy;
```

## For loop

```
for ($i = 0; $i < 10; $i++) {
  print $i . “\n”;
}

for (;;) {
  # infinite loop
}
```

## Loops

```
while ( $foo = <FILE> ) {
  # do stuff
}

do {
  # stuff
} until ($end);
```

# foreach

- Loop over a list

```
@list = ("dog", "cat", "fish");

foreach $f (@list) {
  print $f . "\n";
}
```

# Special loop modifiers

- `next`
  - Restart loop with the next value
- `last`
  - Exit loop
- `redo`
  - Restart loop with the current value

## Input/Output

- File handles are pointers to an I/O stream
- By convention they are in UPPERCASE
- No sigil
- Can be a pipe, socket, file
- Standard handles are
  - `STDIO, STDOUT, STDERR`

```
print STDOUT “Hello World”;
```

# open(FILEHANDLE, expression)

- For read:

```
open(INFILE, "<$fname");
```

- For write:

```
open(OUTFILE, ">$fname);
```

- For appending

```
open(OUTFILE, ">>$fname);
```

- For random access:

```
open(FILE, "+>$fname");
```

## close(FILEHANDLE)

- Use to flush and close an open filehandle

```
close(INFILE);
```

## Reading from FILEHANDLEs

- Scalar context reads one line

```
open(INFILE, “<$fname“);
while (<INFILE>) {
  chop ($line = $_);
}
close(INFILE);
```

## Reading from FILEHANDLEs

- List context reads entire file

```
open(INFILE, “<$fname“);
chop (@file = <INFILE>);
close(INFILE);
```

## Numerical and Binary operators

| | | |
|---|---|---|
| `+` | Addition | `$i = 1 + 2;` |
| `-` | Subtraction | `$i= 8 - 5;` |
| `*` | Mult | `$i = 7 * 4;` |
| `/` | Division | `$i = 9 / 3;` |
| `%` | Modulus | `$i = 4 % 3;` |
| `**` | Power | `$i = 2 ** 6;` |
| `++` | Increment | `$i++;` |

| | | |
|---|---|---|
| `--` | Decrement | `$i--;` |
| `<<` | Shift Left | `$i = $i << 2;` |
| `>>` | Shift Right | `$i = $i >> 2;` |
| `&` | AND | `$i = $i & 0xa` |
| `\|` | OR | `$i = $i \| 0xf` |
| `^` | XOR | `$i = $i ^ 2` |
| `~` | NOT | `$i = $i ~ 1` |

## Compound Assignment operators

| | | |
|---|---|---|
| `+` | Addition | `$i += 2;` |
| `-` | Subtraction | `$i -= 5;` |
| `*` | Mult | `$i *= 4;` |
| `/` | Division | `$i /= 3;` |
| `%` | Modulus | `$i %= 3;` |
| `**` | Power | `$i **= 6;` |

- Works also for bitwise operators (<<, >>, \|, &, ^, ~)

## String operators

| | | |
|---|---|---|
| `.` | Concatenate | `$s = "Hello" . " " . "World";` |
| `.=` | Concatenate - Equals | `$s .= "!";` |
| `x` | Replicate | `$s = ":)" x 32;` |
| `x` | Replicate -Equals | `$s x= 32;` |

# Comparison operators

| **Numeric** | **String** |
|---|---|
| == | eq |
| != | ne |
| < | lt |
| > | gt |
| <= | le |
| >= | ge |

Two different operator types are confusing

```
$i = 12;
if ( $foo < 7 )
             # FALSE
if ( $foo lt 7 )
             # TRUE
```

# Compound Logical operators

| | | |
|---|---|---|
| `\|\|` | OR | `$apples \|\| $oranges` |
| `&&` | AND | `$apples && $oranges` |
| `!` | NOT | `! $fruit` |
| `<=>` | “Spaceship” | `-1 if <, 0 if ==, 1 if >` |
| `cmp` | Compare | `-1 if lt, 0 if eq, 1 if gt` |