Scripting languages

• typically used for short programs to manage other programs
• interpreted, dynamically typed, permissive semantics
• garbage collection
• usually minimal declarations
• usually rich set of string operations (the ultimate untyped data)
• easy interface to OS, file and directory manipulation
Prototyping

**Prototyping**: Quickly putting together a program that does a task in order to be able to experiment with different designs.

Some desirable characteristics of a prototyping language:

- interactive (like LISP, ML, etc)
- garbage-collected, no pointers (LISP, ML, etc.)
- fast compilation: minimal translation to bytecode (like JAVA)
- can be slow
- dynamic typing (like LISP, SETL, APL)
- higher-order functions (LISP)
- built-in indexable structures (like SETL)
- built-in associative structures (like SETL)
**PERL overview**

**PERL** stands for *Practical Extraction and Report Language*

A very successful scripting language, originally for systems administration

- excellent string manipulation facilities
  - regular expressions for string matching and substitution
- combines and surpasses sh/sed/awk
- compact, often cryptic syntax
- dynamically typed (with some minor static typing features)
- scoping: static *and* dynamic (both kinds!)
- built-in arrays/lists and maps
- a vast array of libraries available
- strongly imperative, but
- has first-class functions
- and some higher-order functions: map, grep, sort
**PERL history**

- (1987) PERL 0-3: test versions
- (1991) PERL 4: first public version
- (1994) PERL 5: various new features:
  - references
  - function prototypes
  - statically scoped variables
  - first-class functions, **map**
  - OOP

**The Future:**

- PERL 6 is currently being designed and implemented
  - see [http://dev.perl.org/perl6/](http://dev.perl.org/perl6/)
- a prototype is being written in Haskell (!)
  - see [http://www.pugscodes.org/](http://www.pugscodes.org/)
Some PERL code

```perl
my $greeting = "Hello";
my $num = 2;
print $greeting, $num, "\n";  # 'Hello2' + newline
print "$greeting $num\n";       # 'Hello 2' + newline
print '$greeting $num
';       # '$greeting $num\n'
my @words = ($greeting, $num, "\n");
print @words;                   # 'Hello2' + newline
```

- scalar variable names prefixed by `$`
  (but this is not the whole story)

- list variable names prefixed by `@`

- variable interpolation within double-quoted strings,

- but not within single-quoted strings

- can declare variables using `my`

- `print` takes a list of arguments
A simple type model

- Atomic types: numbers and strings. Not much distinction between them.

- Standard operations use value semantics.

```bash
$count = 15;
$count++;       # now $count == 16
$fullname = $first . $last;   # concatenation
$fullname = "$first$last";   # string interpolation
$count = "count";            # dynamic typing.
$count = "15";
$count++;              # now $count == "16"
```
Scoping

• to declare a *statically* scoped variable:

  my $a = 5;

• to “declare” a new *dynamically* scoped variable, just assign to it:

  $b = 6;

• to save the old value of a dynamically scoped variable and restore it at the end of the current scope:

  local $b = "hi";  # save old value of global $b,
  # assign "hi" to it,
  # and restore old value upon
  # leaving scope
Scoping example

<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub f {</td>
<td></td>
</tr>
<tr>
<td>my $a = &quot;in f&quot;;</td>
<td></td>
</tr>
<tr>
<td>print &quot;f1: a = $a; b = $b\n&quot;;</td>
<td></td>
</tr>
<tr>
<td>g ();</td>
<td></td>
</tr>
<tr>
<td>print &quot;f2: a = $a; b = $b\n&quot;;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>sub g {</td>
<td></td>
</tr>
<tr>
<td>print &quot;g1: a = $a; b = $b\n&quot;;</td>
<td></td>
</tr>
<tr>
<td>my $a = &quot;in g&quot;;</td>
<td></td>
</tr>
<tr>
<td>$b = &quot;in g&quot;;</td>
<td></td>
</tr>
<tr>
<td>print &quot;g2: a = $a; b = $b\n&quot;;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>f();</td>
<td></td>
</tr>
</tbody>
</table>

f1:   a = in f;  b =
g1:   a = ;  b =
g2:   a = in g;  b = in g
f2:   a = in f;  b = in g
Arrays/Lists

**Perl** has a data structure that subsumes arrays and lists.

- how to recognize a list variable: `@a`
- how to write a list literal: `("a", 1, 2, "hello")`
- how to index a list: `$a[4]` (index starts at 0)
- index of the last element: ` $#a`
- conversion to scalar:
  ```perl
  $numElems = @a;  # $numElems gets the number
  # of elements in @a
  ```
- lists are automatically flattened:
  ```perl
  @xs = (1, 2, 3);
  @ys = (10, 20, 30);
  @zs = ("a", @xs, "b", @ys);
  # @zs is ("a", 1, 2, 3, "b", 10, 20, 30)
  ```
Hashes (maps)

**P**erl has an associative data structure called a *hash*.

- how to recognize a hash variable: `%w`
- how to write a hash literal:
  
  `(cat => 5, dog => 22, parrot => 1)`
- how to look up a key in a hash: `$w{dog}`
- how to add/modify a hash element: `$w{dog}++`
- conversion to list:

  `%w = (cat => 5, dog => 22, parrot => 1);`
  
  `@a = %w;  # @a gets ("cat", 5, "parrot", 1, "dog", 22)`

  `# key, val pairs in some order`

- conversion from a list: ("=>" is just an alias for comma)
- hashes are flat – no hashes or lists as values in the hash
Hash iteration

%weight = (cat => 5, dog => 22, parrot => 1);

foreach my $pet (keys %weight) {
    # iterate over a permutation of ("dog", "cat", "parrot")
    ...
}

foreach my $weight (values %weight) {
    # iterate over a permutation of (22, 5, 1)
    ...
}

while (my ($pet, $weight) = each %weight) {
    ...
}
Reference creation

**PERL 5** introduced a new kind of scalar: the reference.

- reference to an existing variable:
  - \$var (reference to a scalar)
  - @var (reference to an array)
  - %var (reference to a hash)

  a bit like \& operator in C/C++

- creating a reference to a new value:

  reference to a list: [1, 2, 3]

  reference to a hash: { dog => 22, cat => 1, mouse => 0 }
Reference dereferencing

- if $v$ is a reference to a scalar, $$v$$ is the value of that scalar
- if $v$ is a reference to an array, @$v$ is the value of that array
- if $v$ is a reference to a hash, %$v$ is the value of that hash

Accessing part of the aggregate:

- if $v$ is a reference to an array, $v->[3]$ is the third element
- if $v$ is a reference to a hash, $v->{dog}$ is the value associated with "dog"
Reference testing

$\text{ref$ } \text{\$var} \text{ is a string that describes the type of$ \text{\$var}.}

\begin{verbatim}
my $i = 1; # number (scalar)
my $rs = \$i; # ref to a scalar
my $rr = \\$i; # ref to a ref to a scalar
my $ra = [ 1, 2 ]; # ref to an array
my $rh = { }; # ref to a hash

print ref $rs, "\n"; # SCALAR
print ref $rr, "\n"; # REF
print ref $ra, "\n"; # ARRAY
print ref $rh, "\n"; # HASH
\end{verbatim}
Nested data structures

Perl 4 only supported “flat” data structures:

- lists of scalars
- hashes of scalars

References allow us to have nested data structures, e.g. lists of lists of hashes.

```perl
my $t = { vowels => [ "a", "e", "i", "o", "u" ],
          consonants => ... };

my @xs = (1..4);
my @ys = ("a", [ @xs ], "b");
# @ys = ("a", [ 1, 2, 3, 4 ], "b")
```
File reading

- To open a file:
  ```perl
  open (HANDEL, "water-music");
  ```

- To close a file handle:
  ```perl
  close HANDEL;
  ```

- To read one or more lines from a file handle:
  ```perl
  $v = <HANDEL>;  # scalar context: read one line
  @vs = <HANDEL>;  # list context: read all lines
  while (<HANDEL>) {
    # next line automatically read into $_
    ...
  }
  ```
Selection

    if (condition1) {
        statement;
        ...
    } elsif (condition2) {
        ...
    } else {
        ...
    }

also has unless:

    unless (condition) { ... }

and there are the post-condition variants:

    statement if condition;
    statement unless condition;
Loops

```perl
my @a = ( 3, 5, 10, 2, 0, 12, 8 );
while ($i < @a) { # length of array
    print "a[$i] = $a[$i]\n";
    $i++;
    last if $a[$i] == 0; # exit loop if element is 0
}

my $sum = 0;
foreach (@a) { # each element gets assigned to $_
    $sum += $_;
}

for ($i = 0; $i < 10; $i++) { # just like C
    ...
}
```
Functions

- arguments are passed by reference
- all arguments are passed as a single flat list of values, placed in global variable `__`
- return value of the function is value of the last expression evaluated, or `return expr` can be used instead

```perl
sub findEq ($@) { # optional 'prototype':
    # one scalar and one list argument
    # need to extract arguments manually:
    my ($a, @b) = @__;
    my @result = ();
    foreach (@b) {
        push (@result, $_) if $a == $_;
    }
    return @result;
}

my @matches = findEq (34, 21, 13, 0, 20..40, 66);
```
Passing functions

Can also pass a reference to a function:

```perl
sub findEq ($@) {  # optional ‘prototype’:  
    # one scalar and one list argument  
    # need to extract arguments manually:  
    my ($f, @b) = @_;  
    my @result = ();  
    foreach (@b) {  
        push (@result, $_) if &$f($_);  
    }  
    return @result;  
}

my @matches = findEq (sub { 34 == $_[0]; },  
                      21, 13, 0, 20..40, 66);

# we could use built-in grep instead:  
my @matches = grep { 34 == $_ } (21, 13, 0, 20..40, 66);
```
String matching

• search for **pattern** in variable $v$:
  
  $$v = /m/pattern/;$$

• we can omit the variable:
  
  $$m/pattern/; \ # \ defaults \ to \ \_$$

• we can also omit the **m**:
  
  $$/pattern/; \ # \ same \ as \ above$$

In all cases, returns true if the match succeeds.

```perl
my $text = "one word after another";

$text =~ m/r /; \ # \ matches \ 'r' \ in \ "after"'
$text =~ m/o.*n/; \ # \ matches \ 'one \ word \ after \ an'
$text =~ m/...r$/; \ # \ matches \ 'ther'
```
Patterns (I)

Single characters:

- “ordinary” characters match themselves
- *metacharacters* can be *escaped* by prefixing them with a backslash to make them ordinary; e.g. `\[` matches the character `[`

  metacharacters: `\ [ ] () {} * + . ^ $ | ?`

- `.` matches any single character
- `[abc0-9]` matches any of the characters `a`, `b`, `c`, or any digit
- `[ˆabc0-9]` matches any single character except `a`, `b`, `c`, or any digit
- `\s` matches any space character, `\S` matches any non-space character

Alternation and sequencing:

- `abc|def|hij` matches any of `abc`, `def`, or `hij`
- `p*` matches zero or more repetitions of `p`
- `p+` matches one or more repetitions of `p`
- `p{i,j}` matches from `i` to `j` repetitions of `p`
Patterns (II)

Anchoring:

- \(^p\) matches \(p\), but only at the beginning of the line
- \(p\$\) matches \(p\), but only at the end of the line

Capturing:

- \((p)\) matches \(p\), but also remembers the match for possible later use
- \(\backslash 3\) matches the 3rd parenthesized pattern (which should precede this backreference)
Substitutions

$v =~ s/pattern/substitution/;

• searches for pattern in variable $v and replaces it with substitution

• substitution can be either a (possibly empty) plain string, or it could contain variables; in particular, $1, $2, ...

These are the 1st, 2nd, ... captured matches.

Examples:

my $t = "The bear ate the cat";
$t =~ s/(\w+) ate the (\w+)/$2 was eaten by the $1/;
print "$t
";  # prints ``The cat was eaten by the bear''

my $x = "abbdddeff";
$x =~ s/(.)/\1/$1/g;  # now $x is "abddef"