

## Perl



- Practical Extraction and Report Language
- Pathologically Eclectic Rubbish Lister

1

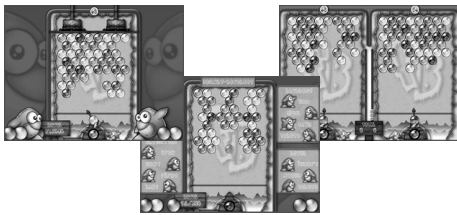
## Mi a Perl?

- általános célú programozási nyelv
  - Web fejlesztés
  - Rendszeradminisztráció
  - GUI programozás
  - Hálózati programozás
  - ...
- Gyakorlatias nyelv:
  - Hatékony, jól használható, teljes
  - A szépség nem volt cél (minimalitás, elegancia)
- Használhatjuk struktúrált és objektum-orientált programozáshoz is
- Rengeteg kiegészítő modul

2

## Frozen Bubble

<http://www.frozen-bubble.org>



- A játék Perl-ben készült
- SDL-t használ a grafika megjelenítéséhez

3

## Perl történetéről

- Larry Wall alkotta a nyelvet saját céljaira az awk és a sed korlátai miatt
- A világ hackereinek megtetszett...
- Perl 1: a Perl és a világ találkozása
  - 1987 december 17.
  - \(\...\)\ helyett (...|...)
- Perl 2: új reguláris kifejezések Henry Spencer regex csomagja révén
- Perl 3: bináris adatok kezelése
- Perl 4: egy kisebb lépés

4

## Perl5

- Perl 5: 1994 októberében 5.000
- Perl5 nagy lépés
  - ez tartalmaz mindent, ami a Perlben lényeges és jó
  - Modulok, tie, OO
  - Threads
  - i18n, l10n
  - POSIX
  - debugger, interaktív környezet
  - regexp
  - Lexikális hatókör (lexical scope)
  - Beágyazható és kiterjeszthető
- Legfrissebb: perl 5.8.5

5

## A Perl jövője

- Perl6:
  - Körülbelül 2000 nyarán jelentették be, hogy el fogják készíteni
  - A közösség fejleszti
  - Fontos, hogy Perl maradjon
  - Fontos, hogy 16 év munkájával kompatibilis legyen
  - Parrot: a fordító és az értelmező szeparálása
    - Python kódot is lehessen fordítani Parrot bytekódra
    - Java bytecode to Parrot bytecode konverter
    - hasonló: Common Language Runtime

6

## Perl bevezetés

7

hello.pl

## Hello world!

- `print "Hello world!\n"`
- Futtatása:
  - `perl -e 'print "Hello world!\n"'`
  - Beírójuk a `hello.pl` nevű fileba, majd `perl hello.pl`
  - UNIX: `chmod +x hello.pl`, valamint egy shebang sor hozzáadása a filehoz:  
`#!/usr/bin/perl -w`  
`print "Hello world!\n"`
  - Majd egyszerűen `hello.pl`

8

## Perl bevezetés

- Megjegyzések a `#` jellel tehetők
- A Perl utasításokat ; választja el egymástól
- Nem kell egy sorba írni az utasításokat

```
print 21 +  
    21,  
    "\n"  
;
```
- Zárójel elhagyható függvényhívásoknál

```
keys %hash;  
keys(%hash);
```
- a nyelv kis és nagybetűket megkülönböztet

9

## Perl bevezetés: típusok

- Perl típusai:
  - Skalárok (scalars)
    - Sztringek
    - Számok
    - Referenciák
  - Listák-tömbök (lists, arrays)
  - Asszociatív tömbök (associative arrays, hashes)

10

scalars.pl

## Perl bevezetés: skalárok

- Szövegek és számok tárolása
- Egy skalár típusú változó nevének első karaktere mindig `$`
- Egy skalár változó tartalmazhat szöveget is, számot is (akár felváltva is)
- A számokat szöveggént is tárolja

11

lists.pl

## Perl bevezetés: listák

- Tömb és lista szinonímák Perlben
- Mérete dinamikusan változik
- Nincs méretkorlát
- Nem kell előre megadni a lista maximális méretét sem
- Lista változók nevének első karaktere mindig `@`
- Listák elemei skalárok lehetnek
- Egy lista elemet már skalárként érünk el

12

## Perl bevezetés: hash-ek

- Kulcs-érték párok tárolására alkalmasak
- Hash típusú változók első karaktere mindig %
- Az értékek skalárok lehetnek
- Mérete dinamikusan és korlátlanul változik
- Elkérhető a kulcsok illetve az értékek listája
- A tárolás a gyorsabb elérés érdekében a Perl magánügye

## Perl bevezetés: referenciák

- Szimbolikus referenciák
  - Egy változó nevét tárolhatjuk benne
  - Nincs jelentős szerepe
  - A strict modul megtiltja a használatukat
- Valódi referenciák (hard references)
  - A továbbiakban 'referenciák'
  - Lényegében mutatók, de null-referencia nincs
  - Speciális skalár típus

## Perl bevezetés: referenciák

- Mutathat más skalárra, referenciára, listára, hash-re, kódra, GLOB-ra és balértékre (például egy substr visszatérési értéke)
- Haszna:
  - Paraméterátadásnál nem kell nagy adatmásolást végezni
  - Komplex adatszerkezetek
    - Többdimenziós tömbök
    - Hash/tömb a hash-ben
    - Hash-ek tömbje
    - Stb.

## Perl bevezetés: referenciák

- Egy változóra a \ operátorral állíthatunk referenciát:
  - \@array, \%hash, \%scalar
- []: új tömbreferencia konstruálása
  - [ 2, 3, 5, 7, 11 ]
  - Tömb: ( 2, 3, 5, 7, 11 )
- {}: új hash-referencia konstruálása
  - { 'I' => 1, 'V' => '5', 'X' => 10, }
  - Hash:
    - ( 'I' => 1, 'V' => '5', 'X' => 10, )

## Perl bevezetés: referencia-dereferálás

- Teljes tömb elérése:
  - @\$\_array\_ref
- Tömbreferencia i. Eleme:
  - \$\_array\_ref->[i]
  - \$\$array\_ref[i]
  - Eml: \$array[i]
- Egy kifejezés értékének dereferálása, ha a kifejezés eredménye tömbreferencia:
  - @{...}

## Perl bevezetés: referencia-dereferálás

- Teljes hash elérése
  - %\$\_hash\_ref
- Hash-referencia 'key' kulcsú leme:
  - \$\_hash\_ref->{'key'}
  - \$\$hash\_ref{'key'}
  - Eml: \$hash{'key'}
- Egy kifejezés értékének dereferálása, ha a kifejezés eredménye hash-referencia:
  - %{...}
- Skalár referencia:
  - \$\$scalar\_ref

## Always 'use strict;'

- A strict pragma használatával a fenti problémák kiküszöbölhetők
- Minden változót deklarálni kell local vagy my kulcsszóval, ellenkező esetben fordítási hibát kapunk
- Szimbolikus referenciák nem használhatók
- Stb.
- Minden Perl tutorial és könyv tartalmazza az "always use strict" mondatot, és igazuk is van

19

## Perl bevezetés: deklarációk

- Nincs szükség deklarációra
- Alapértelmezetten minden változó globális (pontosabban csomagszintű)
- Ebből sok probléma eredhet
  - Deklarálatlan változó használata egy typo miatt
  - Ha egy változó hatóköre túl széles, akkor lehet, hogy ugyanazt használjuk több helyen is:

```
for $i (1..10) {  
    @array = somefunc($i);  
    $AoA[$i] = \@array;  
}
```

20

## Perl bevezetés: deklarációk

- Egy lehetséges workaround:
- ```
for $i (1..10) {  
    @array = somefunc($i);  
    $AoA[$i] = [ @array ];  
}
```
- Igazi megoldást a lexikális hatókörű változók nyújtanak

21

## Mine :)

- A **my** kulcsszóval deklarált változó...
  - abban a lexikális egységben látható, ahol deklarálták
  - Ha a vezérlés elhagyja a deklarálás blokkját, akkor a változó azonnal felszabadul
  - minden blokkba belépéskor újra létrejön (ez az elvárt viselkedés!)
- Ez az, amit általában lokális (néha automatikus) változónak nevezünk

22

## For ciklus lokális változóval

- A ciklus lokális változóval
- ```
for $i (1..10) {  
    my @array = somefunc($i);  
    $AoA[$i] = \@array;  
}
```

23

## Perl bevezetés: elágazások

- Tetszőleges kifejezés szerepelhet feltételben
- Csak a 0, '' (üres sztring), undef hamis
- if (feltétel) {  
 ...  
}
- elsif (feltétel) {  
 ...  
}
- else {  
 ...  
}

24

## Perl bevezetés: elágazások

- unless (feltétel) {  
    ...  
}
- Hátracsapott (postconditional) feltételek:
  - utasítás if feltétel;
  - utasítás unless feltétel;
- print "21 records read." if \$verbose;
- die "Can't open file: \$!\n"  
    unless defined \$filehandle;

25

## Perl bevezetés: ciklusok

- while (feltétel) {  
    ...  
}
- until (feltétel) {  
    ...  
}
- sleep 1 until \$ready;
- print "Hello World!\n" while 1;

26

## Perl bevezetés: ciklusok

- C-style (Perl-kód):  
for (my \$i=0; \$i <= \$#array; \$i++) {  
    print "\$i: \$array[\$i]\n";  
}
- Perl-style:  
foreach my \$element ( @array ) {  
    print "\$element\n";  
}

27

## Perl bevezetés: operátorok

- Aritmetikai: +, -, \*, /, %, \*\*
- Numerikus összehasonlító:
  - ==, !=, <, <=, >, >=, <=>
- Sztring összehasonlító:
  - eq, ne, lt, le, gt, ge, cmp
- Logikai műveletek
  - &&, ||, !
  - and, or, not, xor
- Bitenkénti: &, |, ^, ~
- Inkrementálás, dekrementálás: ++, --
- Sztring összefűzés: .
- Értékadás: =
- Elágazás: ?:
- stb.

28

## Perl bevezetés: I/O

- File megnyitása:  
open(HANDLE, "data.txt")  
or die "Can't open data.txt: \$!\n";
- Olvasás egy file handle-ből:
  - \$line = <HANDLE>;
  - \$line nem definiált, ha EOF
  - while ( my \$line = <HANDLE> ) {  
    print \$line;  
}
- close(HANDLE) or die "Can't close: \$!";

29

## Perl bevezetés: regexpek

- Reguláris kifejezést általában / jelek közé tesszük, de tehetjük más közé is, csak akkor kell 'az *m*' (ld. később)
- Mintaillesztés a =~ operátorral illetve a !~ operátorral történik
- Használható feltételben is
- if ( \$line =~ /\w+/ ) {  
    ...  
}
- A '()' karakterekkel körbezárt csoportok a \$1, \$2, ... változókban érthetőek el
  - print "\$1" if \$line =~ /\w+/;

30

## Perl bevezetés: regexpek

- Az illesztés visszaadja egy listában a csoportokat
- `my ($dir, $filename) = $absolute_path =~ m#(.*)/([^\/]*)#;`
- `my ($dir, $filename) = $absolute_path =~ m#(.*)/(.*)#;`
- Csere is könnyen megy:  
`my $s = "Peti fizika jegye: 1";`  
`$s =~ s/\d$/5/;`
- Minden előfordulás cseréje:  
`my $s = "Itt a meggy. Kell a meggy?";`  
`$s =~ s/meggy/dinnye/g;`

## Perl bővebben

## perldoc

- `perldoc -f` parancs
- `perldoc perl`
- <http://www.perldoc.com>
- nagyon jó dokumentációk

## Perl adattípusok

- skalár
  - szám
  - szöveg
  - referencia
- lista vagy tömb
- asszociatív tömb, hash (map, dictionary)
  - asszociatív, mint összekapcsoló, nem mint zárójelezhető

## Perl változónevek

- első karakter
  - `$`: skalár
  - `@`: lista
  - `%`: hash
  - `&`: eljárások nevei
  - `*`: type glob
- második karakter `'_'` vagy alfa
- többi: alnum és `'_'`
- röviden:
  - `/[\$@%&]*?[:alpha:]_[:alnum:]*`
- speciális a filehandle
- a Perl belső változóinak nevei nem felelnek meg a fenti szabálynak
- a nyelv a kis és nagybetűket megkülönbözteti

## Változó névterek

- Minden változótípusnak külön névtere van
- `@foo`, `%foo`, `$foo` külön változók
- `$foo[0]` -nak nincs köze `$foo`-hoz!
  - `$foo`: a `foo` nevű skalár
  - `@foo`: a `foo` nevű tömb
  - `$foo[0]`: a `foo` nevű tömb első eleme

## Kiértékelési környezetek

- Context
- Egy kifejezés értéke függ a kiértékelés helyétől
- Környezetek
  - Skálár:
    - üres (void context)
    - logikai (boolean context)
    - operátor (operator context)
  - Lista
- Hash környezet nincs

37

## Lista skálár környezetben

- `@foo = (1..10);`
- Lista környezetben a lista önmagára értékelődik ki
- Lista skálár környezetben az elemszámát adja
  - `$bar = @foo;`
  - `$bar == 10;`
  - Eml:
    - `$#foo: 9` (utolsó index)
    - `@foo` skálár környezetben 10 (elemszám)
- Ez a lista lista környezetben értékelődik ki:
  - `my ($bar) = @foo;`
  - `$bar` a lista első elemét tartalmazza
  - `my ($bar, @rest) = @foo;`
- `scalar`: kiértékelés skálár környezetben
  - `print scalar(@foo);`

38

## Skálár lista környezetben

- A skálár lesz a lista egyetlen eleme
  - `my @foo = $bar`
  - `@foo: ($bar)`
- `my @foo = ($bar);`

39

## Hash

contexts.pl

- `$hash{$_} = chr($_) for (48..57);`
- Lista környezetben a kulcs-érték párok felsorolásaként értékelődik ki
- Skálár környezetben a hash belső tárolásáról szolgált információt
- Kulcsok számának meghatározása:
  - `scalar(keys( %hash));`

40

## Függvények viselkedése

- Egy függvény viselkedése függhet a hívás helyétől
- `wantarray` függvény
- `if (wantarray) =>` lista környezet
- `if (!wantarray) =>` skálár környezet
- `if (!defined wantarray) =>` void context (skálár)
- Pl:
  - ```
if (defined wantarray) {  
    # bonyolult számítás elvégzése  
}
```
- `return wantarray ? @results : $results[0];`

41

## Skálár változók

undef.pl

- Skálár:
  - szám
  - szöveg
  - referencia
- Nincs se lehetőség, se szükség a skálár típuson belüli különbségtételre
- Számok és szövegek között automatikus konverzió
- Speciális skálár érték:
  - `undef`
  - logikailag hamis
  - `defined()` operátorral vizsgálható egy skálár definiált volta
  - `undef()` operátor a nemdefiniált (`undef`) értéket adja

42

## Számok

- Egy skalár kifejezésnek mindig van numerikus értéke
- Ha a skalár egy sztringet tartalmaz, akkor az értéke 0 (de ez warning!)
  - `1 + 'alma' # 1 és egy warning`
  - `1 + '3' # 4`
- `undef: 0` (de ez warning!)
  - `1 + undef # 1 és egy warning`
- Perl általában valósággént kezeli a számokat

43

## Numerikus literálok

- 231847
- 12381.123
- .42E+2
- 1\_000\_000
- 0xffff
- 0xdead\_beef
- 0b10001001
- 0644
- 0b110\_100\_100
- Ha ezek egy sztringben vannak, akkor automatikusan csak decimális értékek konvertálódnak
  - `hex('0xff')`
  - `oct('0755')`

44

## Sztringliterálok

- ' és " között is megadható
- bash shellhez képest van különbség
  - aposztróf escape-elhető aposztrófok között
- " között változóhelyettesítés
  - variable interpolation
  - double-quoted string
  - `"$var"`
  - `"${var}"`
  - `"${var}::"`
  - `"$var\::"`
  - `"Price is \$100"`
- skalár és tömb változókra (ill. hash slice-okra)
- ' között nincs változóhelyettesítés
  - `nincs \n \t stb sem de van \'`

45

string-literals.

## Sztringliterálok

- `q{}:` nem helyettesítő
- `qq{}:` helyettesítő
- HERE dokumentum:
  - `<<'END_HERE':` nem helyettesítő
  - `<<"END_HERE":` helyettesítő
- `__DATA__` illetve `__END__` után minden további sort figyelmen kívül hagy a fordító
  - a DATA (már megnyitott) filehandle segítségével kiolvashatjuk az adatokat onnan is
  - utána `close(DATA)`
- `v52.50: '42'`

46

## Műveletek sztringekkel

- `chomp, chop`
- `$line = "example.pl\n";`
- `chop($line): "example.pl"`
  - levágja az utolsó karaktert
  - UNIX-okon rendben van (LF)
  - MAC OS-n rendben van (CR)
  - DOS/Windows: nincs rendben (CRLF)
- `chomp($line)`
  - csak a sorvége jelet vágja le, de azt megfelelően
  - minden OS-n megfelelő

47

## Műveletek sztringekkel

- `uc, ucfirst:`
  - `uc('alma'): 'ALMA'`
  - `ucfirst('alma'): 'Alma'`
- `lc, lcfirst:`
  - `lc('ALMA'): 'alma'`
  - `lcfirst('ALMA'): 'aLMA'`
- `length('almafa'): 6`
- `$greeting = 'Szia Peti! Isten hozott!';`
  - `substr($greeting, 5, 4): 'Peti'`
  - balérték!
    - `substr($greeting, 5, 4) = 'Zsuzsi';`

48



## Listák

- Elemei csak skalárok lehetnek
- Első karakter mindig @
- Például a @ARGV tartalmazza a program paramétereit
- Egy lista megfelelő elemének hivatkozása
- Nem kell előre megadni a tömb elemszámát
- A listát kezelhetjük tömbként is, azaz tetszőleges indexű elemét konstans idő alatt elérhetjük
- \$#list megadja az utolsó elem indexét
- scalar(@list) megadja a lista hosszát
- A skalárok "altípusaira" nincs megkötés

49

## Lista konstruktorok

- a zárójel opcionális, bár a precedencia szabályok miatt mindig érdemes használni
- a listaelemeket, választja el egymástól
- (): üres lista (null list)
- @foo = (1, 'abc', -12, 23.11);
- Vigyázat:
  - \$foo = (1, 'abc', -12, 23.11); # \$foo == 23.11
  - @foo = (1, 'abc', -12, 23.11);  
\$foo = @foo; # 4
- @languages = (  
'Hungarian',  
'English',  
'Italian',  
);

50

## Lista konstruktorok

- ekvivalensek:
  - @range = (1,2,3,4,5);
  - @range = (1..5);
- ekvivalensek
  - @chars = ('a', 'b', 'c', 'd', 'e', 'f');
  - @chars = ('a'..'f');
- sőt: @a\_pairs = ('aa'..'az'); # aa, ab, ac, ad, ...
- ekvivalensek:
  - @numerals = ('one', 'two', 'three', 'four');
  - @numerals = qw(one two three four);

51

## Lista konstruktorok: qw

- Ezzel az operátorral szavak listáját hozhatjuk könnyen létre
- a következők ekvivalensek
  - my @months = qw(Jan Feb Mar Apr);
  - my @months = ('Jan', 'Feb', 'Mar', 'Apr');
- fehér szóközöknél vág

52

## Listák kiegyenesítése

- Abból, hogy csak skalárokat tartalmazhat, következik ez a viselkedés
- @num\_list = (1,2,3,4);
- @char\_list = ('a', 'b', 'c', 'd');
- @lists = (@num\_list, @char\_list);
- az új lista nem két listát, hanem 8 skalárt tartalmaz

53

## Szimultán értékadás

- értékadás bal oldalán is állhat lista
- teljes lista helyettesítése:
  - @foo = ('a', 'b');
- @foo = (1, 2, 3, '#');
- (\$one, \$two) = @foo;
  - \$one, \$two = @foo;
  - eredmény: \$one undef, \$two = 4;
  - ui:
    - \$one, (\$two = @foo);
- (\$one, \$two, @rest) = @foo;
- csere szimultán értékadással:
  - (\$a, \$b) = (\$b, \$a);

54

## localtime

- skalár környezetben
  - `print scalar localtime;`
  - Sat Jan 17 22:34:49 2004
- lista környezetben 9 elemű listát ad vissza
  - `perlsec -f localtime`
  - sec, min, hour, mday, mon, year, ...
  - ha csak az óra, a nap és a hónap érdekel:
  - `(undef, undef, $h, $day, $mon) = localtime();`

55

## Lista elemeinek elérése

- `@hex = ('a'..'f', 'A'..'F');`
- `$hex[0];` # 'a'
- `$hex[-1];` # 'F'
- `$hex[$#hex];` # 'F'
- szeletek (slices): a tömb egy részének kiválasztása
- ez a rész is tömb
- `@hex[0..5]` # kisbetűk
- `@hex[0,6]` # 'a' és 'A'
- Ennek megfelelően:
  - `($year, $month, $day) = (localtime())[5,4,3];`

56

## Műveletek listákkal

- `pop`: a lista végétől kivesz egy elemet
- `push`: a lista végére helyez egy elemet
- `shift`: a lista elejétől kivesz egy elemet
- `unshift`: a lista elejére helyez egy elemet
- lista veremként kezelése:
  - `pop` és `push`
- lista sorként kezelése:
  - `shift` és `push`
  - `pop` és `unshift`
- lista közepe is módosítható a `splice` függvénnyel

57

## Műveletek listákkal

- `push` és `unshift` egy listát vár
  - `push(@nums, 1..10);`
  - `push(@nums, 50..55);`
  - `push(@nums, -1, -2, -10, -57);`
- `@nums = (1, 2);`
- `push(@nums, 3..5);`
- `@nums; # (1, 2, 3, 4, 5);`
- `unshift(@nums, -2..0);`
- `@nums; # (-2, -1, 0, 1, 2, 3, 4, 5);`
- `splice`
  - `my @list = qw(a b c d);`
  - `splice(@list, 1, 0, 'á');`

58

## Műveletek listákkal

- `my $top = pop(@nums);`
  - `$top; # 5`
  - `@nums; # (-2, -1, 0, 1, 2, 3, 4);`
- `my $first = shift(@nums);`
  - `$first; # -2`
  - `@nums; # (-1, 0, 1, 2, 3, 4);`
- `top` függvény nincs
  - `$nums[$#nums]; # 4`

59

## join és split

- Egy lista elemei összerakhatók egy sztringbe
  - `@names = ('Peti', 'Andi', 'Kata');`
  - `join(' ', @names);` # Peti, Andi, Kata
- Egy sztring szétszedhető listába:
  - `$codes = '34:52:a9:9e:5f';`
  - `my @codes = split(':', $codes);`
  - `@codes; # ('34', '52', 'a9', '9e', '5f');`

60

## Hash-ek

- Kulcs érték párok
- Kulcs szám vagy szöveg lehet
- Érték csak skalár lehet (szám, szöveg, referencia)
- `%hash = (`  
    `'date' => '2004-01-10',`  
    `'wheather' => 'rainy',`  
    `);`
- Adott kulcsú elem elérése:  
    `$hash{'date'}`
- A kulcsot nem kell mindig aposztróf közé tenni
- A tárolás nem tartalmaz információt az elemek sorrendjére vonatkozóan

61

## Kulcsok és értékek

- Egy hash kulcsai elérhetők a `keys` operátor segítségével
  - `keys(%hash)`
  - ez egy lista, a kulcsok sorrendje nem definiált
- Egy hash értékei elérhetők a `values` operátor segítségével
  - `values(%hash)`
  - ez is lista, az értékek sorrendje nem definiált
  - a sorrend megegyezik a kulcsok sorrendjével

62

## Értékek vizsgálata

- `defined $hash{'key'}`
  - igaz, amennyiben az érték definiált
- `exists $hash{'key'}`
  - igaz, ha létezik ilyen kulcs a hash-ben
  - különbség, ha az érték `undef`
- `%hash = (`  
    `'key' => undef,`  
    `);`
- `defined $hash{'key'}`
  - hamis
- `exists $hash{'key'}`
  - igaz

63

## Hash és listák

- Egy hash lista környezetben a kulcsok és értékek listájává értékelődik ki
  - `%hex_codes = (`  
    `'a' => 10, 'b' => 11, 'c' => 12,`  
    `'d' => 13, 'e' => 14, 'f' => 15,`  
    `);`
  - utolsó vessző nem typo
  - `@list = %hex_codes;`
    - az eredmény valami ilyesmi:
      - ('e', 14, 'c', 12, 'a', 10, 'b', 11, 'd', 13, 'f', 15)

64

## Hash és listák

- Egy listával is feltölthetünk egy hash-t
- `%hex_codes = @list;`
- `%hex_codes = (`  
    `'a', 10, 'b', 11, 'c', 12, 'd', 13, 'e', 14, 'f', 15);`
- Akár egy kifejezés eredménye is lehet:
  - `%hex_codes = map { $_, ord($_)-87 } 'a'..'f';`
  - `%hex_codes: (`  
    `'a' => 10,`  
    `'b' => 11,`  
    `'c' => 12,`  
    `'d' => 13,`  
    `'e' => 14,`  
    `'f' => 15,`  
    `)`

65

## Hash szeletek

- Egy hash egy szelete az értékeinek egy listája (sorrend van!)
- `@hex_codes{'c', 'b', 'a'}`
  - (12, 11, 10);
- Ez balérték is:
  - `@hex_codes{'c', 'b', 'a'} = (12,11,10);`
  - `@hex_codes{'c', 'b', 'a'} = (12,11);`
    - `$hex_codes{'a'}` undef lesz
- A következő kifejezés például ugyanazt eredményezi, mint a `values` operátor:
  - `@hex_codes{keys %hex_codes};`

66

## Feltételek

- "0" # hamis
- "0.0" # igaz, mert nem a '0' sztring
- 0.0 # hamis, mert konvertálódik, és "0" lesz belőle, ami hamis
- 00 # hamis
- 0.01 # igaz
- "00" # igaz
- {} # igaz, mert ez egy referencia
- [] # igaz, mert ez egy referencia
- undef # hamis

67

## Feltételek

- Listák is kiértékelődhetnek logikai környezetben
- Egy listaváltozó akkor értékelődik ki igazra, ha a lista nem üres
- Egy ,-s kifejezés ilyenkor az utolsó értéket adja
- @list = ();  
print "TRUE" if @list; # hamis
- @list = (0);  
print "TRUE" if @list; # igaz
- De:  
print "TRUE" if (0); # hamis
- print "TRUE" if (); # hamis
- print "TRUE" if (1); # igaz
- print "TRUE" if (1, 0); # hamis

68

## Elágazások

- Tetszőleges kifejezés szerepelhet feltételben
- Csak a 0, '' (üres sztring), undef hamis
- if (feltétel) {  
...  
}  
elseif (feltétel) {  
...  
}  
else {  
...  
}  
• unless (feltétel) {  
...  
}

69

## Feltételek hátul

- A feltétel hátul is lehet, mind if, mind pedig unless
- Ilyenkor a feltétel zárójelek közé zárása nem kötelező
- print 'Connecting..' if \$verbose;
- die 'Cannot connect' unless defined \$connection;

70

## Hibakezelés

- Feladat
  - próbál kapcsolódni
  - ha sikerül, folytassa a munkát
  - ha nem sikerül, lépjen ki hibával
- Így íránk meg:  
my \$conn = get\_connection();  
unless (defined \$conn) {  
die "Cannot connect: \$!\n";  
}  
# do the work

71

## Hibakezelés

- Írhatnánk ezt is:
- my \$conn;  
unless (defined \$conn = get\_connection()) {  
die "Cannot connect: \$!\n";  
}  
# do the work
- De van más mód is
- Lusta kiértékelés
  - and, or operátorok használhatóak hiszen az értékadás is kifejezés
  - &&, || is használhatóak, de ezeknek magasabb a precedenciájuk

72

## Hibakezelés

- `my $conn = get_connection()`  
or `die "Cannot connect: $!\n";`
- Előnyei:
  - tömör
  - olvasható
  - látszik a lényeg, nevezetesen, hogy kapcsolódni szeretnénk
  - de közben korrekt hibakezelés is van
  - Perl-es :)

73

## and példa

- `fork and exit;`
  - probléma: nem kezeli, ha sikertelen a `fork`
- `my $pid = fork and exit;`  
die `"Cannot fork: $!\n"` unless defined `$pid`;
  - mivel az `and` precedenciája kicsi, ezért tökéletesen működik
  - `(my $pid = fork) and exit;`
- `my $pid = fork && exit;`
  - az `&&` operátor precedenciája nagyobb, ezért ezt jelenti:
    - `my $pid = (fork && exit);`
  - helyette:
    - `(my $pid = fork) && exit;`

74

## ?: operátor

- ismerős C-ből
- `feltétel ? igaz_ág : hamis_ág;`
- `defined $param ? print "OK" : print "NOT OK";`
- Ez kifejezés
- `print(defined $param ? "OK" : "NOT OK");`
  - a zárójel elhagyható lenne a precedencia-szabályok miatt
- Ami érdekes, hogy az eredmény balérték is lehet.
- `(is_left() ? $left : $right) += 1;`
  - a zárójel elhagyható lenne a precedencia-szabályok miatt

75

## Ciklusok

`list_manip.pl`  
`list_manip_better.pl`

- `while (feltétel) {`  
...  
`}`
- `until (feltétel) {`  
...  
`}`
- `for (init kif; feltétel kif; növ kif) {`  
...  
`}`
- `foreach $element (@list) {`  
...  
`}`

76

## Ciklusok

- Egy ciklus iterációjának közepén is terelhetjük másfele a vezérlést
  - last: kilépés a ciklusból  
`while (my $line = <STDIN>) {`  
  `last if $line =~ /^\.$/;`  
  ...  
`}`
  - next: következő iterációra lépés  
`while (my $line = <STDIN>) {`  
  `next if $line =~ /^#/;`  
  ...  
`}`

77

## Hátultesztelő "ciklusok"

- `do BLOCK`
- végrehajtja a megadott blokkot és az utolsó kifejezés értékét visszaadja
- ez utasítás, ezért használhatunk utasításmódosítókat hozzá
- `do {`  
  `$lines++;`  
  `print "[ $lines] $line\n";`  
`} if $print_the_lines;`
- `do {`  
  `$lines++;`  
  `print "[ $lines] $line\n";`  
`} while $lines < N;`

78

## Hátultesztelő “ciklusok”

- a do nem ciklus
- nem használhatók a ciklusvezérlő utasítások: next, redo, last
- alternatív lehetőségek
- do {{  
    next if \$x == \$y;  
    ...  
}} until \$x++ > \$z;
- LOOP: {  
    do {  
        last if \$x == \$y\*\*2;  
    } while \$x++ < \$z;  
}

79

## Hátultesztelő “ciklusok”

- a fapados megoldás:
- while (1) {  
    ....  
    last if \$x == \$y;  
}  
\$ \ = “\n”;  
my \$n = 0;  
for (;;) {  
    print \$x++;  
    last if \$x > 10;  
}

80

## Ciklusok: redo

- redo: az adott iteráció újrakezdése
- Egy szó visszaszámlálása:
- while ( my (\$line) = (<STDIN> =~ /^(.\*)\n\$/) ) {  
    print "\$line\n";  
    chop(\$line);  
    redo if length(\$line);  
}
- Hibás a \$line scope-ja miatt

81

## Ciklusok: continue

- a continue blokk közvetlenül a következő iteráció előtt fut le
- next hívásakor például ebbe ugrik
- redo, last kihagyja
- while (my \$line = <STDIN>) {  
    ...  
} continue {  
    \$not\_empty = 1;  
}

82

## Listák bejárása

- Egy lista elemeit bejárhatjuk az indexei szerint (tömbként kezelés)
- for (my \$i=0; \$i <= \$#list; \$i++) {  
    # \$list[\$i] feldolgozása  
}
- Jobb megoldás, ha a foreach ciklust használjuk
- foreach my \$element ( @list ) {  
    # \$element feldolgozása  
}
- Az utóbbi módon nem értesülünk az elem sorszámáról, de erre általában nincs is szükség

83

## Hash-ek bejárása

- Egy hash összes kulcsán illetve értékén végigiterálhatunk
- keys(%hash) eredménye egy lista, ezen pedig már végig tudunk haladni
- foreach my \$key ( keys %hash ) {  
    # (\$key, \$hash{\$key}) pár feldolgozása  
}
- Ha csak az értékekre van szükségünk:  
    - values %hash  
    - ennek az eredménye is lista
- foreach my \$value ( values %hash ) {  
    # \$value feldolgozása  
}

84

## Hash: kulcs-érték párok

- Egy hash összes kulcs-érték párján is végigiterálhatunk
- `each`
- `while (my ($key, $value) = each %hash) {  
    # ($key, $value) pár feldolgozása  
}`
- Hogy ez működjön, minden hash-hez kell egy belső mutató, ami a következő elemre mutat
- Ez újra az első elemre fog mutatni, ha egyszer végigért egy iteráció
- a `keys`, `values` operátorok inicializálják ezt a mutatót

85

## Alprogramok

86

## Alprogramok

- Alprogramok a `sub` kulcsszóval deklarálhatók
- Lehetséges előre deklarálni egy eljárást
- Alprogramok hívása olyan, mint a beépített parancsok használata
  - zárójelek opcionálisak is lehetnek
  - általában azért érdemes kitenni
- Formális paraméterek nem adhatóak meg
- Utolsó kiértékelt kifejezés értéke lesz a visszatérési érték
- Egy alprogramhívás történhet lista, skalár vagy üres környezetben (list, scalar, void context)

87

hello\_sub.pl

## Alprogramok hívása

- Egy egyszerű alprogram:
- `sub NAME BLOCK`
- `sub hello {  
    print "Hello world!\n";  
}`
- egy alprogram hívása:
  - `NAME`;
  - zárójel csak akkor hagyható el, ha az alprogram előre definiált, vagy importált
  - `NAME()`;
  - `&NAME`;
  - `&NAME()`;
- `hello()`;

88

## Alprogramok: paraméterek

- Nem definiálhatunk formális paramétereket
- Ilyen forma nem használható:
  - `sub print_message($to, $message) {  
    ....  
}`
- A paraméterek egy egyszerű listában kerülnek átadásra
- Ez a lista az `@_` lista
  - `$_[0]` az első paraméter
  - `$_[1]` a második paraméter
  - stb.

89

## Alprogramok: paraméterek

- `sub print_message {  
    print "Message to $_[0]: $_[1]\n";  
}`
- `print_message('Andi', 'Este randi?');`
- `sub write_capital {  
    foreach my $name (@_) {  
        print ucfirst($name), ' ';  
    }  
    print "\n";  
}`
- `write_capital('andi', 'gabi', 'peti');`
  - Andi, Gabi, Peti,

90

## Alprogramok: paraméterek

- Ez is jó:
- `my @names = ('andi', 'laci', 'zsuzsa');`
- `write_capital(@names);`
  - Andi, Laci, Zsuzsa,
- A Perl kiegyenesíti a paramétereket az @\_ listába
- Ezért ez is teljesen jól megy:
- `@boys = ('peti', 'zoli', 'feri');`  
`@girls = ('juci', 'mari', 'kati');`  
`write_capital(@girls, @boys);`
- Juci, Mari, Kati, Peti, Zoli, Feri,

91

## Alprogramok: paraméterek

- ```
sub pretty_print {  
    foreach my $param (@_) {  
        print ">>$param<<\n";  
    }  
}
```
- A következő hívásnak nincs értelme:
- `pretty_print(@list, %hash, $scalar);`
  - minden paraméter az @\_ listába kerül
  - a Perl kiegyenesíti a listákat
  - tehát nem három paramétere lesz, hanem @list elemszáma plusz %hash kulcsai számának duplája plusz 1

92

## Alprogramok: paraméterek

- Egy hosszabb eljárásban az @\_ tömbön keresztül hivatkozni a paraméterekre kényelmetlen és nem jó
- Ötlet: shift-elhetünk belőle
  - ```
sub print_messages {  
    my $to = shift(@_);  
    print "Message to $to\n";  
    my $first_message = shift(@_);  
    print "\t[1] $first_message\n";  
    my $count = scalar(@_);  
    print "\t$count more message(s)\n";  
}
```

93

## Alprogramok: paraméterek

- @\_ az alapértelmezett tömb, ha nincs megadva semmi, akkor ezt kezeli a shift
- ```
sub print_messages {  
    my $to = shift;  
    print "Message to $to\n";  
    my $first_message = shift;  
    print "\t[1] $first_message\n";  
    my $count = scalar(@_);  
    print "\t$count more message(s)\n";  
}
```
- zavaró, és könnyű elrontani, ha módosítunk a kódon

94

## Alprogramok: paraméterek

- my deklarációk segítségével az alprogramban élő változókat használhatunk a paraméterek nevesítésére
- ```
sub mysub {  
    my ($param_1, $param_2) = @_;  
    ...  
}
```
- @\_ lista:
  - ha üres, akkor mindkettő változó értéke undef lesz
  - ha 1 elemű, akkor csak \$param\_2 lesz undef
  - ha 2 elemű, akkor egyik sem lesz undef
  - ha több elemű, akkor a többi paraméter 'elvész', az alprogram nem használja őket

95

## Alprogramok: paraméterek

- ```
sub mysub {  
    my ($param_1, $param_2, @rest) = @_;  
    ...  
}
```
- @\_ lista:
  - ha üres, akkor mindkettő változó értéke undef lesz, a lista pedig üres
  - ha 1 elemű, akkor csak \$param\_2 lesz undef, valamint a lista üres
  - ha 2 elemű, akkor egyik sem lesz undef, de a lista üres lesz
  - ha több elemű, akkor a többi paraméter a listába kerül

96



## Alprogramok: paraméterek

- ```
sub print_messages {  
    my ($to, $first, @rest) = @_;  
  
    print "Message to $to\n";  
    print "\t[1] $first\n";  
    my $count = scalar(@rest);  
    print "\t$count more message(s)\n";  
}
```

97

## Alprogramok: default paraméterek

- Átadott paramétereknek default értéket nem adhatunk
- Van rá mód, hogy ezt megkerüljük
- ```
sub log_message {  
    my ($message, $level) = @_  
    $level = defined $level ? $level : 'INFO';  
    ...  
}
```
- kicsit jobb, ha:
  - ```
sub log_message {  
    my ($message, $level) = @_  
    $level = 'INFO' unless defined $level;  
    ...  
}
```

98

## Alprogramok: default paraméterek

- De ez is jó:
  - ```
sub log_message {  
    my ($message, $level) = @_  
    $level = $level || 'INFO';  
    ...  
}
```
- Vagy tömörebben és olvashatóbban:
  - ```
sub log_message {  
    my ($message, $level) = @_  
    $level ||= 'INFO';  
    ...  
}
```

99

## Alprogramok: visszatérés

- A visszatérési érték is egy lista lehet
  - Ha két listát adnánk vissza, akkor azok kiegyenesítődnek
- Természetesen a lista lehet egyelemű is
- Az utolsó kiértékelt kifejezés értéke lesz a visszatérési érték
- Explicite megadható a return utasítással
  - Ha nem adunk meg visszatérési értéket, akkor skálár környezetben undef, lista környezetben pedig üres lista lesz a visszatérési érték

100

## Alprogramok: visszatérés

- ```
sub succ {  
    my ($num) = @_  
    $num + 1;  
}
```
- ```
$nat = 0;  
$nat = succ($nat) while 1;
```
- A fenti eljárásba az olvashatóság kedvéért érdemes egy return-t tenni
- ```
sub succ {  
    my ($num) = @_  
    return $num + 1;  
}
```
- vagy teljesen lerövidítve:
  - ```
sub succ { $._ + 1 }
```

101

## Alprogramok: visszatérés

- Természetesen rekurzió is van Perlben
- ```
sub fact {  
    my ($n) = @_  
    return 1 if $n <= 1;  
    return $n * fact($n - 1);  
}
```
- ```
my $fact_of_3 = fact(3); # 6
```

102

## Alprogramok: visszatérés

- Visszaadhatunk listát is
- ```
sub stepped_range {  
  my ($start, $end, $step) = @_;  
  my @result = ();  
  for (my $i = $start; $i <= $end; $i += $step) {  
    push(@result, $i);  
  }  
  return @result;  
}
```
- ```
my @list = stepped_range(10, 21, 3);  
- (10, 13, 16, 19)
```

103

## Alprogramok: visszatérés

- Ha egy alprogramot más környezetben hívunk, akkor más is lehet az eredmény
- ```
my $results = stepped_range(10, 20, 3);
```

  - az eredmény lista négy elemű
  - ennek kiértékelése skalár környezetben a lista elemszámát adja
  - ```
$results; # 4
```

104

## Alprogramok: listák

- ```
sub inc {  
  my (@numbers) = @_;  
  
  my @result = ();  
  foreach my $number ( @numbers ) {  
    push(@result, $number + 1);  
  }  
  
  return @result;  
}
```
- ```
my @inced = inc(@nums);
```
- ```
my (@odds, @evens) = inc(@evens, @odds);
```

  - hibás, az eredmény lista is kiegyenesítődik

105

## Alprogramok: wantarray

- Eldönthető, hogy az eljárást milyen környezetben hívták
- `wantarray` függvény
  - hamis, ha skalár környezetben
  - igaz, ha lista környezetben
  - undef, ha üres környezetben (void context)
- Előző `inc`
- ```
my $result = inc(1); # 1
```
- helyesen:
  - ```
my ($result) = inc(1); # 2
```

    - könnyű elrontani
    - kényelmetlen

106

## Alprogramok: wantarray

- ```
sub inc {  
  my (@numbers) = @_;  
  
  my @result = ();  
  foreach my $number ( @numbers ) {  
    push(@result, $number + 1);  
  }  
  
  return wantarray ? @result : $result[0];  
}
```
- Lista környezetben visszaadja az egész listát
- Skalár környezetben csak az első elemet
- Minden korábbi formával működik

107

## Alprogramok: wantarray

- Kihashználhatjuk még, hogy üres környezetben (void context) a `wantarray` undef-t ad
- ```
sub my_sub {  
  ...  
  return unless defined wantarray;  
  
  # műveletigényes számítások  
  ...  
  return wantarray ? @results : $result[0];  
}
```
- Ha nem használjuk fel az eredményt, akkor el sem végzi a számításokat

108

## Alprogramok: prototípusok

- A paraméterátadás szintaxisa és szemantikája miatt semmiféle ellenőrzés nincs a paraméterek számára és típusára vonatkozóan
- Ezt egy kicsit szigoríthatjuk
- Prototípusok: fordítási idejű ellenőrzés
- Nem teljeskörű, nincs lehetőség elnevezni a paramétereket
- Rövid leírás egy függvény paramétereinek típusára illetve számára vonatkozóan

109

## Alprogramok: prototípusok

- ```
sub add($$) {  
    return $_[0] + $_[1];  
}
```
- két skalárt vár
- `add(12, 13);`
- hibás hívások
  - `add(1);`
  - `add(1,2,3);`
  - `add(@numbers)`
    - ahol `@numbers: (1,2)`
- szintaktikailag helyes:
  - `add(@numbers, $scalar);`
  - lista kiértékelődik skalár környezetben

110

prototypes.pl

## Alprogramok: prototípusok

- `$`: skalárt várunk
- `@`: listát várunk
- `%`: hash-t várunk
- `&`: kódot várunk
- `*`: elfogad typeglobot is, skalárt, konstans, bareword-t
- `\` karakterrel az eljárás már referenciát kap
- `;` -vel jelezhetünk opcionális paramétereket
- alternatívákat a `[]` jelöléssel adhatunk meg

111

## Alprogramok: prototípusok

- Referenciák használatával az átadott paramétereket közvetlenül módosíthatjuk
- A Perlben eddig megszokottól eltérően viselkedhet egy függvényhívás
- `sub fill(\@$);`
  - a paraméterül adott tömböt módosíthatja
  - a függvény egy referenciát kap a tömbre

112

## Alprogramok: prototípusok

- `&` paraméterrel akár látszólag új szintaxist is teremthetünk
- az átvett paraméter egy szubrutin referencia
- ```
sub transaction(&) {  
    my ($codes) = @_;  
  
    eval {  
        &$codes;  
    };  
  
    if ($@) { $dbh->rollback(); }  
    else { $dbh->commit(); }  
}
```

113

correctness.pl

## Alprogramok: prototípusok

- a függvény hívása:
- ```
transaction {  
    # DB inserts...  
  
    # DB deletes...  
};
```
- lényeges és kötelező a pontosvessző

114

## Alprogramok: konstansok

- konstansként kezeltek a paraméter nélküli prototípussal definiált alprogramok, ha a szemantikájuk ezt lehetővé teszi
- a következők konstansként viselkednek
  - sub PI () { 3.14159 }
  - sub FLAG\_READ () { 1 << 2 }
  - sub FLAG\_WRITE () { 1 << 1 }
  - sub FLAG\_EXECUTE () { 1 }
  - sub DIR\_PERM () {  
    FLAG\_READ | FLAG\_EXECUTE  
}
- a csupa nagy betűs név konvenció, nem kényszer

115

## Alprogramok: konstansok

- Ilyen célra a constant modult érdemes használni
- use constant PI => 4 \* atan2(1,1);
- use constant DEBUG => 0;
- use constant NAPOK => (  
    'Hétfő', 'Kedd', 'Szerda', 'Csütörtök',  
    'Péntek', 'Szombat', 'Vasárnap',  
);
- use constant NAPOK => qw(  
    Hétfő Kedd Szerda Csütörtök  
    Péntek Szombat Vasárnap  
);

116

## Alprogramok egymásban

- Elvileg lehetséges alprogramokat egymásba ágyazni
- Gyakorlatilag problémákat vet fel
- Egy alprogram definiálása az aktuális környezetben történik
- A belső alprogram nem használja a tartalmazó alprogram változóit
  - warning: ... will not stay shared...

117

## Privát változók

- my operátorral deklarált változók
- lokális a tartalmazó blokkra, eval-ra vagy file-ra nézve
- lexikálisan privát
- ha a strict modul használjuk, akkor mindenképpen szükséges deklarálni minden változót valamilyen módon
- a továbbiakban feltesszük, hogy használjuk a strict modul
- my \$x;
- \$x értéke undef lesz
- my \$x = 10;
- \$x inicializálva a 10 értékre
- **\$x lexikálisan privát**

118

## Privát változók

- Ha egyszerre több változót szeretnénk deklarálni, akkor zárójelek közé kell tenni
- my \$x, \$y;
  - hibás, \$y nem ismert
  - végrehajtásilag ugyanaz, mint:
    - my \$x;
    - my \$y;
- my (\$x, \$y);
- Inicializálás lehetséges listával:
  - my (\$x, \$y) = (10, 12);

119

## Privát változók

- my (\$foo) = <STDIN>;
  - a kifejezés lista környezetben értékelődik ki
- my @foo = <STDIN>;
  - szintén
- my \$foo = <STDIN>;
  - skálár környezetben
- A következő sortól kezdve lesz deklarált a változó
- my \$x = \$x;
  - csak akkor érvényes kifejezés, ha a \$x már ismert volt

120

## Privát változók

- Egy alprogramban a deklarálás helyétől az alprogram végéig látható egy my változó
- sub subroutine {  
    my (\$arg1, \$arg2) = @\_;  
    ...  
    my \$sum = \$arg1 + \$arg2;  
    ...  
}
- Kívülről nem érhető el
- Nem minősíthető az alprogram nevével
  - tehát pl. nincs ilyen: \$subroutine::sum, vagy hasonló

121

## Privát változók elágazásban

- Szintén a deklaráció helyétől az elágazás végéig érvényes
- Ha a feltételben inicializáljuk, akkor az egész elágazáson át látható a változó
- if ((my \$answer = lc(<STDIN>)) eq "yes\n") {  
    user\_agrees();  
} elsif (\$answer eq "no\n") {  
    user\_disagrees();  
} else {  
    chomp \$answer;  
    die "'\$answer' is neither 'yes' nor 'no'";  
}

122

## Privát változók elágazásban

- Hátracsapott feltétel esetén a láthatóság nem definiált
- Bár jelenleg a láthatósága kiterjed az egész blokkra, evalra, file-ra, a későbbi verziók ezt viselkedést megváltoztathatják
- my \$line = <STDIN> if \$read\_next\_line;
  - nem helyes
- my \$line;  
\$line = <STDIN> if \$read\_next\_line;
  - helyes

123

## Privát változók ciklusokban

- while (my \$line = <STDIN> ) {  
    chomp(\$line);  
    ...  
}
- continue blokkban is látható a my változó
- for ciklusban a ciklus végéig látható
  - for (my \$i = 0; \$i < 10; \$i++) {  
    ...  
}
- foreach ciklus esetén szintűgy
  - foreach my \$element ( @list ) {  
    ...  
}

124

## Privát változók élettartama

- my változók lexikális láthatóságát a tartalmazó blokk határozza meg
- élettartamuk lehet szélesebb is ennél
- akkor szabadul fel, ha már nincs rá hivatkozás

125

## local

- Minden változó, mely nem my, globális
- A my ezt a nem túl rugalmas viselkedést szabályozza
- a local elmenti a változó értékét, és egy blokkra vonatkozó lokális értéket ad neki
- {  
    local \$SIG{'INT'} = 'IGNORE';  
    func();  
}

126

## local

- {  
  local \$dbh->{'RaiseError'} = 0;  
  \$dbh->do(q{  
    UPDATE salaries  
    SET salary += salary \* 0.10  
  });  
}
- {  
  local STDERR = \*STDOUT;  
  print STDERR "...";  
}