• Facts:
  ■ Creator: Guido van Rossum
  ■ Based on:
    ▪ ABC
    ▪ Modula-3
  ■ Name:
    ▪ From a TV show
Python

- High-level scripting language
- Multi-paradigm
- Easy to learn
- Stack-based interpreter
- Garbage collection using reference count algorithm
Goals:
- An interpreted language useful for writing and testing administration scripts.
- A programming language intended for non-technical users.

Principal uses:
- Web programming.
- Software development.
- Building prototypes.
def sum(a, b):
    if type(a) == int:
        if type(b) == int:
            return a + b
    if type(a) == str and type(b) == str:
        return a.upper() + b.upper()
Python has a set of built-in types:

*Lists and Tuples*: a collection of objects which can be referenced by their position number within each object.

```python
mylist= [1,44,[3,'cat'], 'pet']
mytupler= (1, 2, 'salad')
```
Dictionary: a collection of associated *key:data* pairs

```python
students={123: 'Michael', 134: 'Joanna'}
```

(Similar to Perl hashes)

File: disk files as in:

```python
file1 = open('data.01','r')
data = file1.read()
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Parallelppiped:
    pass

>>> cube=Parallelppiped()
>>> cube.length = 10
>>> cube.center = Point(20, 20)
Iterators

for element in (1, 2, 3):
    print element

>>> l = 'abc'
>>> it = iter(l)
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'

→ Returns an iterator object
Generators are functions that create and return iterators. The instruction `yield` specifies the point of return of an intermediate value. (Much like Ruby)

```python
def fib():
    a, b = 0, 1
    while 1:
        yield b
        a, b = b, a+b

def func():
    c=0
    b=fib()
    while c<10:
        print b.next()
        c=c+1
```
Lists

List of objects

```python
list = [1, 2, 3]
list.append(42.5)
```

stack

```python
list.pop()
```

queue

```python
list.pop(0)
```

Functional prog.

```python
def square(x):
    return x**x
```

```python
map(square, list)
```

List comprehension

```python
[x**x for x in list]
```
def directories(path):
    a = []; i = []
    for root, dirs, files in os.walk(path):
        a.append([root, dirs, files])
    for list in a:
        for folders in list:
            for files in folders:
                i.append(files.split('.')
    return i

def types(filelist):
    typelist = []
    for file in filelist:
        if len(file) == 2:
            typelist.append(file[1])
    return typelist

filetypes = Set(types(directories(path)))
Sorting and seeking for a value within a set

```python
sports = [ 'Football', 'Baseball', 'Cricket']
sports.sort()
print sports
[ 'Baseball', 'Cricket', 'Football']

>> 'Cricket' in sports
True
```
# Productivity

Our project in 220 lines!!

## Python News Parser

Results for the search of the word 'olympic' in the RSS news feeds.

<table>
<thead>
<tr>
<th>Category</th>
<th>Articles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Economy</td>
<td></td>
</tr>
<tr>
<td>Politics</td>
<td></td>
</tr>
<tr>
<td>Religion</td>
<td></td>
</tr>
<tr>
<td>Sports</td>
<td></td>
</tr>
<tr>
<td>Olympic diary</td>
<td>(BBC News) Following the Olympic torch on its Everest journey</td>
</tr>
<tr>
<td>Major attractions</td>
<td>Olympic torch relay route in Ho Chi Minh city (<a href="http://www.xinhuanet.com">www.xinhuanet.com</a>)</td>
</tr>
<tr>
<td>Vietnamese</td>
<td>eager to watch Beijing Olympic torch relay (<a href="http://www.xinhuanet.com">www.xinhuanet.com</a>)</td>
</tr>
<tr>
<td>Olympic Flame</td>
<td>arrives in Vietnam (<a href="http://www.xinhuanet.com">www.xinhuanet.com</a>)</td>
</tr>
<tr>
<td>Syrian</td>
<td>political, sports elite express support for Beijing Olympics (<a href="http://www.xinhuanet.com">www.xinhuanet.com</a>)</td>
</tr>
<tr>
<td>Olympic Flame</td>
<td>reaches Mt. Qomolangma (<a href="http://www.xinhuanet.com">www.xinhuanet.com</a>)</td>
</tr>
<tr>
<td>Olympic torch</td>
<td>arrives in Vietnam (Indiatimes - Times of India)</td>
</tr>
<tr>
<td>Tibetan</td>
<td>shadow on Olympics (Indiatimes - Times of India)</td>
</tr>
<tr>
<td>World</td>
<td></td>
</tr>
<tr>
<td>Tibetan</td>
<td>shadow on Olympics (Indiatimes - Times of India)</td>
</tr>
</tbody>
</table>

The Beijing Olympic flame arrived in Vietnam's Ho Chi Minh City from North Korea, on one of the final legs of its troubled worldwide journey.

The Olympic torch was greeted by protesters in Japan on Friday.
In short...

- Python is a highly productive scripting language.
- Easy to learn, easy to read.
- It has “batteries included”.
- Useful for writing working prototypes.
Perl

in a Clamshell

Jordan Marshall & Matthew Ghigliotti
An Example...

```perl
open(FILE, shift @ARGV) ||
    die "I couldn’t open the file.\n";

LOOP:
while ($line = <FILE>) {
    $line = lc $line;
    foreach $word (@ARGV) {
        $word = lc $word;
        if ($line =~ /\W|\A$word(\W|\Z)/) {
            print $line;
            next LOOP;
        }
    }
}
```
History of Perl

- Created by Larry Wall; first release in 1987
- Originally for text manipulation
- Created in response to
  - ↑ costs of programmers
  - ↓ costs of hardware
  - Superior compiler software
- Perl 5 released in 1994 – still the major version number
  - Introduced modules; inspired Comprehensive Perl Archive Network (CPAN) in 1995
Implementation

- Core interpreter written in C
- De facto standard
- Garbage collection scheme
- Interpreting and compiling operations interleaved
- Specialized tools to parse Perl
- Ported to most OSes
- Difficult to maintain Perl’s source code
Language Features

- Objects
- Types
- Garbage collection
- Dynamic type checking
- Most of the advanced structures discussed
- Extensible through modules
- Ties closely to the UNIX shell and C
The Perl Philosophy

“More than one way to do something in Perl.”

- Imperative

$/ = ";";

```perl
while (<>)
    /
        /\A\s*(.*?)/\s*\z/;
    print $1, "\n" if $1;
```

sub quicksort {
    @_ ? () : (quicksort(grep {$_ < $_[0]} @_[1..$#_]),
             $_[0],
             quicksort(grep {$_ >= $_[0]} @_[1..$#_]));
}
Paradigms (Logic)

use AI::Prolog;

my $input = <<'PROLOG';
    color(tree, green).
    color(sun, yellow).
    plant(X) :- color(X, green).
PROLOG

my $pro_inst = AI::Prolog->new($input);
$pro_inst->query("plant(sun). ");

while (my $results = $pro_inst->results) {
    print $results->[2], "\n";
Type Conversion

- Operations are evaluated according to context
  - Depends on the operation and arguments

```perl
$string = "jordan";
$array = ("great","awesome","charming");
$string = $string.@array;
$array = $string;

# $string has the value 'jordan3'
# and @array has one element, $string.
```
package Thingy;
sub new {  
    my $class = shift;
    my $self   = {};  
    $self->{NAME} = undef;
    bless ($self, $class);
    return $self;
}
sub name {  
    my $self = shift;
    if (@_) {$self->{NAME} = shift}
    return $self->{NAME};
}
Objects and Inheritance

- Objects are equivalent to anonymous hashes

```perl
package Doodad;
use Thingy;
@ISA = ("Thingy");
1;

package Widget;
use Thingy;
use Doodad;
@ISA = ("Thingy", "Doodad");
1;
```
Text Processing

- Regular expressions built in
- Implicit global variables

```perl
while (<>) {
    s/<.+?>//g;
    print;
}

$/ = ";";
while (<>) {
    /\A\s*([^\s]+)\s*\z/;
    print $1, "\n" if $1;
}
```
Summary

- Built for ease of use
- Scripting and processing in conjunction with the UNIX shell
- Very powerful and capable
- Multiple programming paradigms
- Not a large learning curve from similar languages
Introduction

• Product of University of Arizona, Ralph Griswold et. al, influenced by SNOBOL4 and SL5
  - High level general-purpose programming language
  - Paradigm : Procedural
  - Compiled / Interpreted

• Language extensions
  • Graphics features, Unicon(+OOP), Jcon(Java based), MT Icon- (multitasking)

• Applications :
  • Scripting/String processing – compilers(LUCA), databases, word processors.
  • Research – AI, Natural language processing,
  • Expert systems, Rapid Prototyping
  • Graphic displays
procedure main()
    local a
    a := [3.14, 42, 1.2, 1]
    s_sort(a)
    write("Ans: ", a[*a])
end

procedure s_sort(a)
    local i; local j
    every i := 1 to *a-1 do
        every j := i+1 to *a do
            if a[j] < a[i] then a[i] :=: a[j]
        end
    end
end

• local variable passed-by-reference
• every creates generators
• a is a list
• *a gives the size of the structure
• :=: swap operator!
Procedures

procedure main()
  every write(Invert(-3, 3))
  every write(Invert(-5))
end

procedure Invert(i, j)
  /j := i
  while i <= j do {
    suspend -i
    i +=: 1
  }
  fail
end

3 2 1 0 -1 -2 -3 5

• No nesting
• Returns with
  - success giving a valid result
  - fail returning null value or
  - suspend saving the dynamic state which can be resumed

• Parameter Passing
  - Variables passed-by-value
  - If object mutable(eg. List, Table) then passed-by-reference
  - initial, &null, default parameters
Typing & Scope

• Data Types
  – Objects have types, variables do not
  – Weak dynamic typing
  – Built-in types
    • numerics (integers, reals), csets, strings, sets, lists, tables, records and procedures
  – Type conversion
    • implicit - operation dependent
    • explicit - using (conditional) conversion routines

• Variables & Scope
  – Lexical scoping
  – local, static, global scope declarations
  – record, link, procedure
record imdb(title, rating)

list1 := [1, "Two", [3, 4]]
r1 := imdb("Matrix", 8)
r2 := imdb(21, 6.6)

tbl := table(0)
tbl[1] := list1
tbl["2"] := r1

get, push, pop

put, pull

List based Queue OR Stack

Lists - queue/stack
- ===, |||, ~===, |||:=

Tables (Dictionaries)
- key, value pairs

Sets - unique elements
- union ++, intersection **, difference --

Records
- declared global
- fields can hold any type
Generators

str := "Python is like Icon"
# produces 5 and 18
every write(find("on", str))

list1 := ["i","c","o","n"]
writes(every(!list1))
#prints icon

i := 0; j := 1; k := 2
# generates 1 and 2
every write(i < j | i < k)

# generates 0 to 4
every write(seq(i) \ 5)

• Iterate through outcomes of expressions which produce multiple values
• every is used to create expression context for multiple values
• procedures can also act as generators using suspend
Expressions

4 > 3 #success, result = 3
1 = 0 #failure

if find("on", str) > 10 then
write("Bingo!")

max := max < x
write("y =", (x | 5) > y)

line := read() & write(line);
#implicit error handling

coopr_name := create expr
@coopr_name

• Success-Failure semantics
• Goal directed evaluation
  • failure drives control
  • automatic searching among alternatives

• Control Backtracking
  • limited to expr
  • every e1 do e2 `drives' e1
• Co-expressions
  – capture state of an expr
  – suspend/resume an expr
  – transfer control, co-routines
String Operations

1 2 3 4
i c o n
-4 -3 -2 -1 0

s ? expr
#s-subject, expr-exprn/operator

line ? while tab(upto(&letters))
do write(tab(many(&letters)))
#prints all letters in line

s:=map("fg/ij/cd","cdefghij",&date)

• Strings - first class objects
  - cursor, subscripting
• String scanning
  - tab(),upto(),many()
• Editing & conversion
  - image(s) , left(s,i), map () , trim() , replace(s1,s2,s3)
• Character sets
  - &lcase,&letters,&digits
• Regex
More features...

• Polymorphism
  – polymorphic operations
  – operation depends upon the operand types
  – 1. \( s := \text{“ad”} \)
    \[
    s[3:3] := \text{“a”} \quad \# \text{concatenation followed by assignment, } s \text{ is “ada”}
    \]
    \[
    \text{write}(s[1:3]) \quad \# \text{substring operation}
    \]
  
  2. \( \text{write}(*\text{str}) \) \quad \# \text{gives the size of the object, be it a string, list or table}

• Preprocessing
  • \$\text{define PI 3.14} \ (\text{macro expansion - done before syntactic and semantic analysis})
  • \$\text{include “mathlib.icn”}
Storage Management

- Data allocated in separate regions
- Static, Non-static blocks & strings
- Co-expressions in static blocks
- Garbage collection: marking & compaction
- Predictive Need
  + assures adequate amount of space before allocation
  + garbage collector can be invoked at a safer time
  - maximum amount of storage must be determined
  - predictive need requests prior to allocation
Summary

• Missing features
  – Modules with distinct namespaces
  – Object oriented programming
    • Unicon adds OO features
  – Exception Handling
  – Concurrency
    • MT Icon

• Comparison with other scripting languages
  • Python (heavily influenced), Perl, Ruby, gawk

• Overall
  • Good for scripting, advanced features
  • Too many operators, readability issues
GAWK

Amit Wadhwa
Nithya Chandrasekaran
Introduction

History

- Aho, Weinberger, Kernighan 1977
- GNU AWK – 1986
  - Paul Rubin & Jay Fenlason
  - Arnold Robbins
- NAWK
  - Kernighan

Text Processing and Pattern Matching

Formatted reporting

One to Thousand line programs
# Walk, Talk & GAWK

## General Users
- As a power tool on command line
- Excellent for work on databases and tabular data

## Bigger things
- Networks
- AI
- XML
- Bio Informatics
Getting Started

Records and Fields
- File = records + fields
- Each line is a record
- Fields are delimited by a special character
  - Whitespace
  - Change with “–F” (command line) or FS (special variable)

Generic GAWK Command
- /pattern/{ action}
- BEGIN and END Blocks
Example 1

• Command: `ps aux | gawk '$11 == "ssh" {print $2}'`

• Output:

Example 2

• Command: `awk 'NF > 0' file`

• Output:

List all pids running ssh ...

Getting Started ...
Two Ways to run Gawk

From the Command line

• `cat file | gawk '(pattern){action}'`
• `cat file | gawk -f program.awk`

From a script

```bash
#!/usr/bin/gawk –f
# This is a comment
(pattern) {action}
```
Interesting Aspects

- Asks the machine about the current time

BEGIN
"/inet/tcp/0/localhost/daytime"
|& getline
print $0
close("/inet/tcp/0/localhost/daytime")
}

Data Driven Paradigm

*nix to Windows

POSIX compliant

Can access standard UNIX stream

Includes TCP/IP networking functions

No objects, Not functional; No built-in logic programming!
### Aspects...

- Interpreted
- No predefined memory limits
- Associative Arrays
- Lint Checking
- Automatic initialization
- Implicit Coercion
- No pointers 😊
Program

Prepare

Sort

Format

Human readable

Array of numbers
grep output

Name: 1.0 | 1.2
A: 1.0 | 1.2
B: 4.0 | 5.0

A: 1.0 | 1.2
B: 2.5 | 6.0
Vars and Arrays

Variables
- Predefined variables (NF,NR,SUBSEP...)
- No need for declaration
- Implicitly set to 0 AND Empty String
- Implicit coercion

Arrays
- Dynamic
- Associative
- String indexing
- Built in Array Iterator “in”

One type
- X=x+1
- X=“ppl”

Array Assignment
- B[“foo”] = “bar”;

Iterator
- For ( x in myarray ) {
### More about Arrays

The arrays in gawk can be used to implement almost any data structure.

<table>
<thead>
<tr>
<th>Type</th>
<th>Example Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set:</td>
<td>• myset[&quot;1&quot;] = 1; myset[&quot;a&quot;] = 1;</td>
</tr>
<tr>
<td></td>
<td>• If ( &quot;x&quot; in myset )</td>
</tr>
<tr>
<td>Multi-dimensional</td>
<td>• myarray[1,3] = &quot;a&quot;; myarray[1,&quot;ppl&quot;] = 5;</td>
</tr>
<tr>
<td>array:</td>
<td></td>
</tr>
<tr>
<td>List:</td>
<td>• mylist[1,&quot;value&quot;] = 2; mylist[1,&quot;next&quot;] = 3;</td>
</tr>
</tbody>
</table>
Concepts

Regular Expressions

Case Sensitive Matching

- \( x = "aB" \)
  
  ```
  if (x ~ /ab/) ...  \# this test will fail
  ```

This one will ignore case

- \( \text{IGNORECASE} = 1 \)
  
  ```
  if (x ~ /ab/) ...  \# now it will succeed
  ```

Conditional flow

Dynamic Expressions

- \( \text{BEGIN} \)
  
  ```
  \{$0 \sim \text{regexp} \{ \text{print} \}$
  ```
Summary

Text Processing
- Regular Expressions

Reporting
- Prepare, Sort, Format

TCP/IP and String functions available

Simple and smart variables/arrays
- No types, Associative, Iterator

Niche
- No objects, logic programming, functional concepts
References

- GAWK manual
  http://www.cs.utah.edu/dept/old/texinfo/gawk/gawk_toc.html
- Introduction to GAWK
  http://www.linuxjournal.com/article/1156
- Getting started with GAWK
- Sams Red Hat 6 unleashed
“The Worlds most misunderstood Programming Language” - Douglas Crockford

Presented By:
Pallavi Chilappagari
Natasha Gaitonde
Road Map

- Features and Applications
- Functions
- Objects
- Prototype Based Inheritance
- Closures
- Garbage Collection
- Conclusion
Features
- Light weight, interpreted scripting
- Weakly typed
- Dynamically Typed
- First class Functions
- Object Oriented
- Javascript Engines

Applications
- Validating user input
- Interactive, dynamic content
- Client environment customization
- Cookie interaction
HelloWorld.html
<script type="text/javascript" src="Validate.js"/>
<body onload="welcome()">

Validate.js
function welcome()
{
  var txt="Hello World!";
  document.title=txt;
  setTimeout('startTime()',500);

  var x= new Array(2);
  document.getElementById('listtxt').value=x.sort();
}
Functions

- Functions are Objects
- Higher Order Functions
  Function myMap(f, arr) {
    for (x in arr) {
      arr[x] = f(arr[x]);
    }
  }
- Custom class Constructors and Methods
- Event handling using Callbacks
function Person(name) {
    this.name = name;
    this.sayHi = function() {
        alert( this.name + " says hi");
    };
} }  
x = new Person("Jerry");
x.sayHi();

Person.prototype.changeName = function(name) { this.name = name; }

y = new Person("Ben");
y.changeName("Tom");
y.sayHi();
Prototype Based Inheritance

- Classical Inheritance
- Runtime alteration of Prototypes

**Circle.prototype**

<table>
<thead>
<tr>
<th>Area = CArea()</th>
</tr>
</thead>
<tbody>
<tr>
<td>circum = CCircum()</td>
</tr>
<tr>
<td>pi = 3.1428</td>
</tr>
</tbody>
</table>

**Circle: c**

<table>
<thead>
<tr>
<th>radius = 5.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 2.0</td>
</tr>
<tr>
<td>y = 6.2</td>
</tr>
</tbody>
</table>

**Circle: d**

<table>
<thead>
<tr>
<th>radius = 5.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 2.0</td>
</tr>
<tr>
<td>y = 6.2</td>
</tr>
<tr>
<td>pi = 3.14</td>
</tr>
</tbody>
</table>
Closures

- A closure is a special kind of object that combines two things: a function, and the environment in which that function was created.

- Can be used to emulate private methods in JavaScript.

- Used in ‘trimbreakpoint’, a debugging utility for JavaScript, to inspect variables during execution.
Emulating private methods with closures

```javascript
var Counter = (function() {
    var privateCounter = 0;

    function changeBy(val) {
        privateCounter += val;
    }

    return {
        increment: function() {
            changeBy(1);
        },
        value: function() {
            return privateCounter;
        }
    };

    alert(Counter.value()); /* Alerts 0 */
    Counter.increment();
```
Garbage Collection

• JavaScript implements a variant of mark-and-sweep algorithm

• Browsers use reference counting to handle the memory allocated for DOM objects

• Common to have cyclic references between the JavaScript variables and DOM objects

• Cyclic references and closures can result in memory leaks
Circular Reference Example

```javascript
<script type="text/javascript">
var ob;
window.onload = function(){
ob=document.getElementById("DivElem");
document.getElementById("DivElem").expandoProperty=ob;
ob.bigString=new Array(1000).join(new Array(2000));};
</script>
```
Circular Reference illustrated
Memory leak due to Closures

<script type="text/javascript">
    function LeakMemory()
    {
        var parentDiv = document.createElement("div");
        parentDiv.onclick=function(){ foo(); };
    }
</script>

<input type="button" value="Memory Leaking Insert" onclick="LeakMemory()"/>
Conclusion

Please refer to our paper for more on

- Scope
- Generators, Iterators and Array Comprehensions
- Exception Handling
- Closures

Thank you
TCL/TK

CS520 Final presentation

• Balaji Ramamurthy
• Xuchen Wang
**Introduction**

- **Tcl** is the abbreviation of "Tool Command Language". It is a scripting language created by John Ousterhout. Originally "born out of frustration"

- **Tk** is one of the most important Tcl extensions, written in C, for rapid prototyping of GUI interfaces for X windows system.
Overview

1) Tcl as general-purpose scripting language
   • special features

2) Tcl `s Embeddability & Extensibility
   • i.e.: Integrated with C

3) Advanced topic
   • FP, OOP, GC in TCL
Tcl as scripting language

- As a **scripting language**, Tcl is similar to other UNIX shell language such as the C shell. Here is what a typical Tcl program looks like:

```tcl
proc add {a b} {
    puts stdout \$x + \$y = [expr \$x + \$y];
}

# same effect as above
puts stdout " \$x + \$y = [expr \$x + \$y] "
```
Regular Expression

• The most powerful way to express patterns is with regular expressions. Here is an example:

```
set x http://www.arizona.edu;
regexp {^[^:]+(?=.*\.edu$)} $x match;

match => http
```
Scope

- **Scope:** Variables defined outside a procedure are not visible to the procedure, unless upvar or `global` scope commands are used.

```plaintext
proc ObjInit { o x y } {
    global obj
    set obj($o,x) $x
    set obj($o,y) $y
    set obj($o,dist) [expr sqrt($x* $x + $y * $y)]
}
proc ObjMove { o dx dy } {
    global obj
    incr obj($o,x) $dx
    incr obj($o,y) $dy
    set obj($o,dist) [expr sqrt(obj($o,x)* $obj($o,x) + \ $obj($o,y) * $obj($o,y))]
}
```
Dynamic Scope

- **upvar && uplevel command**
  The Tcl upvar and uplevel commands allow a procedure to modify the local variables of any procedure on the call stack. i.e:

```
  c
  b
  a
  Global

  level 1 level2 level #0

  d
  b
  a
  Global
```
Embeddability and Extensibility

- Tcl is designed to be easily extensible by writing new command implementations in C.
Functional programming

- **Functional programming in Tcl**

  Tcl is often used imperatively, but at bare-bones it's *fully functional* -- just by the fact that every command returns a result.

```tcl
# used imperatively
proc readfile name {
    set fp [open $name]
    set data [read $fp]
    close $fp
    return $data
}
```

```tcl
# used functionally
proc readfile name {
    {lambda fp {K [read $fp] [close $fp]}} [open $name]
    proc K {a b} {set a}
```
• **OOP in Tcl**
  - There was no build-in OOP support before Tcl 8.5
  - In Tcl 8.5, XOTcl extension is added to the Tcl core.

```tcl
class Stack {    # "tcl++" or "smallTcl"
    variable s {}
    method push args {eval lappend s $args}
    method pop {} {
        if ![llength $s] {error "stack underflow"}
        K [lindex $s end] [set s [lrange $s 0 end-1]]
    }
}
```
• **Garbage collection**
  1) Tcl does not need memory management when it is used in a "Native" way
  2) **GC** is absolutely needed to work with OO.

"An object or two may sometimes be nice, just like a glass of beer. But one shouldn't start drinking at breakfast."

- John Ousterhout
Summary

1) Tcl as general-purpose scripting language
   - string substitution
   - scope
   - regular expression
   - upvar&&uplevel commands

2) Tcl`s Embeddability &Extensibility
   - integration Tcl with C

3) Advanced topic
   - functional programming
   - OOP
   - GC
   - GC