Introduction

- History

Bertrand Meyer

Eiffel Power™
from ISE
Case Study

defered class VEHICLE
feature entities
  velocity: INTEGER is a function
  do
    Result := speed
  end
wheels: INTEGER is
defered
end
speed: INTEGER
stop is a procedure
defered
end
end-VEHICLE

class CAR
  inherit VEHICLE
feature entities
  color: COLOR is a field
wheels: INTEGER is 10
speed: INTEGER
stop is a procedure
  speed := 0
  end
end-CAR
Special Properties

- Multiple inheritance and Renaming

```plaintext
class TEACHING.Assistant inherit TEACHER
    rename
    account as faculty_account
    select
    faculty_account
end

STUDENT
    rename
    account as student_account
end

ta: TEACHING.Assistant
up: UNIVERSITY.Person
up := ta

ta.faculty_account

ok

ok

ta.student_account

ta.account

x

up.account --ta.faculty_account
```
Special Properties

- Typing (Static)
  - Genericity
    - Unconstrained
      - STACK[G]
    - constrained
      - BINARY_TREE[G->COMPARABLE]
  - Assignment attempt
    - X ?= Y
  - Anchored declarations
    - X: like Y
  - No type cast
Special Properties

- Exception Handling
  
  ```
  write_next_character(f:FILE) is
    —Write the available in last_character in to the file
    —retry 5 times
  require
    writeable:file.writeable
  local
    num_attempts:INTEGER.
  do
    low_level_write_function(f,last_character)
  rescue
    num_attempts:=num_attempts+1
    if num_attempts<5 then
      retry
    end
  end
  ```

- No Main and Globals
  - Root class
  - Root procedure
Design Principles

- Design by Contract

<table>
<thead>
<tr>
<th>provide_service</th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>(Satisfy precondition:) Pay bill</td>
<td>(From postcondition:) Get telephone service</td>
</tr>
<tr>
<td>Supplier</td>
<td>(Satisfy postcondition:) Provide telephone service</td>
<td>(From precondition:) No need to provide anything if bill not paid</td>
</tr>
</tbody>
</table>

- Command Query Separation
- Uniform Access Principle
  - a.balance (attribute? function?)
class ACCOUNT
feature - Access
  balance : INTEGER - Current balance
  deposit_count : INTEGER is - Number of deposits
    do
      if all_deposits /= VOID then Result:= all_deposit.count end
    end

feature - Element change
  deposit(sum:INTEGER) is - Add sum to account
    require non_negative: sum >= 0
    do
      if all_deposits = VOID then create all_deposits end
      all_deposits.extend(sum)
      balance:=balance+sum
    ensure
      one_more_deposit: deposit_count = old deposit_count+1
      updated: balance= old balance+sum
    end

feature{NONE} - Implementation
  all_deposits: DEPOSIT_LIST - List of deposits
  invariant
    consistent_balance : (all_deposits /= Void) implies (balance=all_deposits.total)
    zero_if_no_deposits : (all_deposits = Void) implies (balance=0)
end-class ACCOUNT
Internals

- Compilation process
  - Eiffel Compiler
  - Translator
  - C Compiler

- Garbage Collection
  - Mark and Sweep
  - Generation Scavenging
# OO languages Comparison

<table>
<thead>
<tr>
<th>Factor</th>
<th>Eiffel</th>
<th>C++</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>How object-oriented</td>
<td>Purely OO</td>
<td>Hybrid</td>
<td>Purely OO</td>
</tr>
<tr>
<td>Design by Contract and assertion</td>
<td>Design by Contract Language support</td>
<td>Nothing comparable Only assert instr</td>
<td>Nothing comparable Only assert instr</td>
</tr>
<tr>
<td>Static typing</td>
<td>Statically typed</td>
<td>Statically typed but C style cast allowed</td>
<td>Typed mostly statically but dynamic for containers</td>
</tr>
<tr>
<td>Compiler Technology</td>
<td>Combination of interpretation and compilation</td>
<td>Usually compiled</td>
<td>Mix of interpretation and on the fly compilation</td>
</tr>
<tr>
<td>Automatic Documentation</td>
<td>Documentation extracted automatically without extra programmer effort</td>
<td>No standard mechanism</td>
<td>JavaDoc: add special comments</td>
</tr>
<tr>
<td>Multiple Inheritance</td>
<td>Multiple inheritance</td>
<td>Multiple inheritance but with problems</td>
<td>Single inheritance but multiple interface</td>
</tr>
<tr>
<td>Automatic memory management</td>
<td>Garbage collection automatic memory management</td>
<td>No Garbage collection</td>
<td>Garbage collection</td>
</tr>
</tbody>
</table>
Summary

... All that you need,
To program with speed:
Objects and classes,
Compiled in four passes...
Try inheritance today,
In the Eiffel way,
And you are certain to find,
That when it is combined,

With the bindings dynamic,
And the typings static,
And the classes generic,
And ISE magic,
Things fall in place,
By mysterious grace,
Makes programming a pleasure,
By any sane measure...

-- Ross D'Souza, 1992
Objective-C

Introduction

[Karthik Ravichandra]
[Bhavin Mankad]
History

1984
- Designed by Brad Cox and Tom Love at their company StepStone

1988
- Steve Jobs’ NeXT licensed Objective C from StepStone and released their own version of Objective-C compiler, libraries (NeXTstep)

1992
- NeXT partnered with Sun Microsystems to develop OpenStep based on NeXTstep
- GNUstep - Glatting, Stallman

1996
- Apple acquired NeXT and used OpenStep in Mac OS-X

* Timeline not to scale
Goals of Design

- Object Oriented Design and Development
- Strict Super-set of C
- Simple Syntax Extensions to C, influenced by SmallTalk
- Convenient Mixing of Structured and Object Oriented Programming
- Dynamic Behavior
Sample Code

- **Square.h - Interface**

  ```
  #import "Shape.h"

  @interface Square: Shape {
    float side;
  }

  +(void) countRectangles;
  -(id ) initWithSide: (float) side;
  @end;
  ```

- **Square.m - Implementation**

  ```
  #import "Square.h"

  @Implementation Square

  -(void) initWithSide: (float) side
  {
    //method body
  }
  @end
  ```
Objects, Classes and Inheritance

- Basic Building Blocks of OO programming
- Only Single Inheritance allowed
- NSObject is the root for all classes
- Dynamic or Static Typing for Objects

```objective-c
id square = [Square alloc] init];
or
Square *square = [[Square alloc] init];
```
Messaging

- Syntax:

  - Message without arguments
    
    ```
    [ rectangle init];
    ```

  - Message with two arguments
    
    ```
    [ rectangle setLength:100 andWidth:60];
    ```

- Method calls via message passing
- Preprocessor translates a message into `objc_msgSend(..)` function
- Class template lookup for method calls on objects at runtime (Similar to Luca!)
Dynamic Behavior

- Dynamic Typing
  - Deciding the class of an object at runtime.

- Dynamic Binding
  - Deciding which method to invoke at runtime.

```objc
id Obj = getReceiverObj(bool flag); // True – Rectangle, False – Square
SEL msg = getMessage(bool flag); // True – SetWidth, False - SetSize
[ Obj performSelector: msg];
```
Categories

- Extends the functionality of a class.
- A good alternative to sub-classing.
- Sub-classes inherit the new methods.

Merits

- Can split implementation of a huge class.
- Simplifies code management.
- Base class code is not recompiled.
Properties.

- Allows for easier ways to access variables.
- Can associate attributes.
  - `readwrite`, `readonly` etc.
- Can synthesize getter and setter methods.

```objective-c
@interface Rectangle{
    ...
    @property (readwrite) int size;
    ...
}

@implementation Rectangle{
    ...
    @synthesize size;
    ...
}

Rectangle *rect = [[Rectangle alloc] init];
rect.size = 10; // [Rectangle setSize: 10]
```
Memory Management

- Manual Memory Management in 1.0
- Uses reference counting technique.
- Keywords - retain, release, autorelease.
- Example:

  ```
  Rectangle *rect = [[Rectangle alloc] init]; // retainCount is 1.
  Rectangle *tmprect = rect;                // Another reference to rect.

  [tmprect retain];                         // So we increment the retainCount.
  ....
  [tmprect release];                       // Decrement retainCount.
  ....
  [rect release];
  ```
Summary

- One of the early Object Oriented Languages.
- Popularized by Apple.
- Based on C, influenced by Smalltalk.
- Significant Dynamic Behavior.
- Some cool features like Categories, Properties etc.
- Generation based Garbage Collection.
MODULA-3

Swaminathan Sankararaman  Bojan Durickovic
MODULA-3 Example

MODULE Main;
BEGIN
n := 3;
END Main.

Imports

Exports

LinkedList.i3:

INTERFACE LinkedList;
PROCEDURE Add(VAR I : INTEGER);
END LinkedList.

MODULE LinkedList;
PROCEDURE Add(VAR I : INTEGER) = 

BEGIN
n := 3;
LinkedList.Add(n);
END Main.
## History

<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1986</td>
<td>Proposal</td>
</tr>
<tr>
<td>1988-1989</td>
<td>Language Definition</td>
</tr>
<tr>
<td>1990s</td>
<td>DEC SRC-M3, CM3 and PM3 released</td>
</tr>
<tr>
<td>1990s</td>
<td>Popular as a teaching language</td>
</tr>
<tr>
<td>1994</td>
<td>SPIN OS developed</td>
</tr>
<tr>
<td>1996-1997</td>
<td>CVSup developed</td>
</tr>
<tr>
<td>2000</td>
<td>Critical Mass Inc. ceases operations</td>
</tr>
<tr>
<td>2000-2002</td>
<td>Elegosoft takes over CM3 and PM3</td>
</tr>
</tbody>
</table>
Goals and Target Audience

- Structuring of Large Programs
- Safety and Robustness
- Machine-level programming
- Simplicity

TARGET AUDIENCE

Intended

Programmers of large, sophisticated systems and applications

Actual

Programming Language Instructors
Researchers
Objects and Generics

OBJECT

- Record paired with Method Suite

```plaintext
TYPE T = OBJECT
  a: INTEGER;
  METHODS
  a() := A
END;

PROCEDURE A(self: T) = ... ;
```

GENERICS

- Parametric Polymorphism

```plaintext
GENERIC INTERFACE LinkedList(Elem);
  INTERFACE Integer; TYPE T = INTEGER END Integer.
END m1.

METHODS set(v: Elem.T) := P; END;

PROCEDURE P(self:T; v:Elem.T) = ... ;

GENERIC INTERFACE LinkedList(Elem);
REVEAL T = BRANDED OBJECT val: Elem.T ;
METHODS set(v: Elem.T) := P; END;
PROCEDURE P(self:T; v:Elem.T) = ... ;
END m1.
```

- Entire Modules and Interfaces are Generic

```plaintext
INTERFACE Integer; TYPE T = INTEGER END Integer.
INTERFACE IntList = LinkedList(Integer) END IntList.
MODULE IntList = LinkedList(Integer) END IntList.
```
Type System

NO Ambiguous Types or Target-Typing

```plaintext
VAR x : REAL; y : INTEGER; z : REAL;
x := y*z;
(* Type of y*z depends on y and z and not x *)
```

NO Auto-Conversions

```plaintext
VAR x : REAL; y : INTEGER;
y := x;
(* No Automatic floor(). Static Error *)
```

Type Compatibility

```plaintext
S <: T \iff S \subseteq T
```

Structural Equivalence

```plaintext
T \equiv U \implies
Expanded Definitions of T and U are the same
```

Procedures

Global\rightarrow First-Class, Local\rightarrow Second-Class
Information Hiding - Revelations

- At an implementation level, reveal only features of the type relevant to that level
  \[ \text{REVEAL } T <: U \]

- Revelations are linearly ordered with final revelation defining the concrete type of the object

<table>
<thead>
<tr>
<th>Level</th>
<th>Interface</th>
<th>Revelation</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>INTERFACE LinkedList; REVEAL T &lt;: LL</td>
<td>Implementor knows only details of Linked List</td>
<td></td>
</tr>
</tbody>
</table>
| 2nd   | INTERFACE LinkedListChar; REVEAL T <: LLChar | Implementor knows only details of Char Linked List  
Also LLChar <: LL |
| 3rd   | INTERFACE String; REVEAL T = STR | Implementor knows details of String |
Safety and Garbage Collection - I

▲ Traced/Heap and Untraced References/Heap

Roots

Traced References

REF1

REF2

Untraced References

Heap

Traced Heap

Untraced Heap
Garbage Collection Algorithms in SRC Modula-3 Compiler

- Mostly Copying Collection
- Incremental, Generational

SAFE AND UNSAFE MODULES

SAFE MODULES
No way to produce *unchecked runtime error*

VAR P : REF INTEGER
INC ( P );
( * WRONG!! * )

UNSAFE MODULES LOOPHOLE ( expr , Type )

VAR x , y : REF INTEGER;
y := ADR ( x ) + 1;
( * Allowed * )
Summary

- Large-Scale Systems and Applications
- Systems-Level programming when required and isolated when not
- Designers’ goals realized through committee voting
- Not widespread in industry
  - Decline of DEC
  - Compiler Inefficiency
- Personal Experiences - Complex for small code but easy to modify and change implementation. Focus on design.
OBERON

Waj and Manish
Sample Program

MODULE Hello ;

IMPORT InOut ;

PROCEDURE Disp();
BEGIN
  InOut.WriteString("HelloWorld");
  InOut.WriteLn;
END Disp ;

BEGIN
  Disp();
END Hello.
Introduction.

• Developed by Niklaus Wirth at ETH Zurich in 1986.
• Was originally designed for the Oberon Operating System.
• Simplified, reduced Modula-2.
• Is both Procedural and Object Oriented.
• Language specified in a page of EBNF.
• Oberon report is about 16 pages.
Procedures and Parameters

• Both pass by value and pass by reference supported.

• Open array parameter allowed.
  – PROCEDURE P(s: ARRAY OF CHAR)

• Procedure Forward Declarations
  – PROCEDURE ^Sum(x, y: REAL): REAL
Types

• Static type checking, Strongly typed
• Procedure type
  – Function = PROCEDURE():BOOLEAN ;
    search: PROCEDURE( f:Function; arr:ARRAY OF INTEGER) ;
    search(LinearSearch, in);
    search(BinarySearch, in);
Type Extension

• Ability to derive a new type from an existing one.
• Eg:
  Base = RECORD x, y: INTEGER END;
  Extn = RECORD (Base) z: INTEGER END;
• Extn is compatible with Base.
• This leads to classes and objects
**OO Features**

- **Type Bound Procedures** – methods and installation.
- **Type Guard/Type casting.**
- **Polymorphism**

```pascal
TYPE Fptr = POINTER TO Figure;
Figure = RECORD (* base type *)
  print : PROCEDURE (f: Fptr);
END;

Circle = RECORD(Figure)
  radius:Integer END;
Var c:Circle;
NEW(c);
c.print := PrintCircle;
```
Modules

• Basic building blocks.
• Definitions and implementation in the same file.
• Modules can be imported and exported.
• Facilitates data abstraction.

```
MODULE Hello;
IMPORT InOut;
PROCEDURE Disp();
BEGIN
  InOut.WriteString("HelloWorld");
  InOut.WriteLn;
END Disp;

BEGIN
  Disp();
END Hello.
```
MODULES Contd…

• Unit of execution is a procedure. Procedure in a module can be invoked as a command (eg: M.P where P is a procedure in M).

• Modules are dynamically loaded. Global data structures retain values and can be reused upon successive activations of the same procedure.
Dynamic Loading

- Modules are separate compilation units.
- No pre-linked files on the disk. Every module is compiled to a separate object file.
- Modules are loaded as and when commands are executed.
- Modules share the same address space and are loaded only once.
Other Features

- Garbage collection
- Fast Compilation.
- Generates native and portable code.
- Support for System Programming.
- Assertions.
Summary

• Simple Language
• 3 main features
  – Type extension
  – Abstraction
  – Dynamic Loading.
• No support for concurrency
C# Language
By Anand and Deepti
using System;
using System.Collections;

namespace MySpace
{
    class Hello
    {
        static void Main()
        {
            Console.WriteLine("Hello world");
        }
    }
}
History

- Conceived by Microsoft in 1997
- Anders Hejlsberg leads development of C#
- Based on
  - C/C++
  - Java
- Emphasis on simplification
Target Audience

- Projects that require high productivity
  - Short learning curve
  - Massive base class library, like Java
- Windows Platform Developers
- Embedded Software
- Programming Models
  - Procedural Object Oriented
**Type System**

- **Value types**
  - Directly contain data
  - Cannot be null
- **Reference types**
  - May be null

```csharp
float a = 4.2;
string s = "PPL";
```

```
<table>
<thead>
<tr>
<th>a</th>
<th>4.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>&quot;PPL&quot;</td>
</tr>
</tbody>
</table>
```
Unified Type System

- Everything is an object
  - Primitive types are just an alias to the object type
- Type-safety
  - All variables are initialized
Unified Type System

- **Boxing**
  - Creates an object

- **Unboxing**
  - Copies the value out of the object

//Signature for hash table Add method

```csharp
Add(object key, object val)
```

//In C# code, you can write

```csharp
Hashtable h = new Hashtable();
h.Add(5, "five");
```
Properties

- Used to ‘set’ and ‘get’ class members

```csharp
public class MyClass
{
    private string myString;

    public string MyString
    {
        get
        {
            return myString;
        }
        set
        {
            myString = value;
        }
    }
}

MyClass object = new MyClass();
object.MyString = "PPL";
String str = object.MyString;
```
Delegates

- Object oriented function pointers

```csharp
public delegate void LogHandler(string message); // Define the Delegate

public void ConsoleLog(string s){ Console.WriteLine(s); } // Subscriber 1

public void FileLog(string s){ File.WriteAllText(s); } // Subscriber 2

LogHandler myLogger = null; // Declare the Delegate

myLogger += new LogHandler(ConsoleLog); // Add to Subscription List
myLogger += new LogHandler(FileLog);

myLogger(message); // Call Subscribers
```
Events

- Declare a Delegate
- Declare an Event based on Delegate
- Fire the Event

Publisher

- Subscribe Event
- Fires the event handler
- Event Handler

Subscriber 1

Subscriber 2

- Subscribe Event
- Fires the event handler
- Event Handler
public delegate void LogHandler(string message);  // Declare a delegate

public event LogHandler Log;  // Declare an event based on the delegate

protected void OnLog(string message)  // Fire the event
{
    if (Log != null)
        Log(message);
}

static void ConsoleLog(string message) {  // Event Handler
    Console.WriteLine(message);
}

Log += new LogHandler(ConsoleLog);  // Subscribe to the Event Handler
Unsafe Code

- ‘unsafe’ keyword
  - Enables pointers of primitive type
  - Casts and pointer arithmetic
- ‘fixed’ keyword to escape the GC
- ‘stackalloc’ keyword to allocate from stack
  - Does not initialize, hence faster

```c
unsafe
{
    int * ptr;
    ptr = &(new Int32(5));
    *ptr = 10;

    fixed (long *ptr_a = &(myClass.a))
}
```
Iterators

- Allows to iterate through members of your collection

```csharp
public class MyList : IEnumerable<string>
{
    public IEnumerator<string> GetEnumerator()
    {
        foreach (string s in strings)
            yield return s;
    }
}

// Main program
foreach (string s in mylist_object)
{
    Console.Write(s);
}
```
Garbage Collector

- Implicit GC
- Explicit GC
  - `System.GC.Collect();`
  - Run each unused object's finalizer on separate thread
  - `System.GC.WaitForPendingFinalizers();`
Summary

- Procedural Object Oriented
- Large Base Class Library
- Unified Type System
- Properties, Delegates and Events
- Unsafe code
- Iterators
- Garbage Collector
  - Implicit and Explicit
Forth
A Stack Programming Language

Harley Witt
Russell Lewis
Basic Syntax

1 2 +
1+2

5 DUP *
5*5

DUP 0 < IF -1 * THEN  (i < 0) ? -i : i
History of FORTH

• Charles (Chuck) H. Moore (1970)
  – Shorten Edit/Compile/Run Time
  – Text Interpreter To Act On Words

• National Radio Astronomy Observatory (Arizona)
  – Data Acquisition and Analysis
  – Radio Telescope Control
Our Example Program

- Luca VM Interpreter
- Example word
  : addi
    1 IGNORE_TOKEN
    +
  ;
- Luca VM code (filtered)
  addi 0
Structure Of Interpreter

- Interpreter
- Luca VM Code
Executing Luca

pusha 0 15 0
pushi 0 37
storei 0

Forth Stack Before: (Empty)

: pusha
1 IGNORE
NUMBER
GETVARADDR
1 IGNORE
;

Forth Stack After: (0)
Executing Luca

pusha 0 15 0
pushi 0 37
storei 0

: pushi
1 IGNORE
NUMBER
;

Forth Stack Before: (0)
Forth Stack After: (0 37)
Executing Luca

pusha 0 15 0 : storei
pushi 0 37 1 IGNORE
storei 0 SWAP

Forth Stack Before: LUCATOFORTH
(0 37) !

Forth Stack After: ;
(Empty) ;
Strengths

• Bootloaders & prototype hardware
• “Portable assembly language”
• Extreme self-reflection and modification
• Integrated interpreter/compiler
Variables

VARIABLE foo

VARIABLE bar 10 CELLS ALLOT

foo
foo @
1 foo !
foo @ bar 3 CELLS + !
bar foo @ CELLS + @ bar foo @
  1- CELLS + !

int foo;

int[10] bar;

&foo

“read” foo

foo = 1;

bar[3] = foo;

bar[foo-1] = bar[foo];
Weaknesses

- Difficult syntax (parens are comments)
- Inconsistent syntax (some words are prefix)
- Terrible memory management
- Inadequate abstraction
- Not even trivial typechecking (think: like assembly language)
PostScript

Ravi Sheshu Nadella
Sushanth K. Reddy
Drawing a Square

/inch {72 mul} def % Convert inches->points (1/72 inch)
newpath % Start a new path
1 inch 1 inch moveto % an inch in from the lower left
1 inch 0 inch rlineto % bottom side of the Square
0 inch 1 inch rlineto % right side of the Square
-1 inch 0 inch rlineto % top side of the Square
closepath % Automatically add left side to close path
stroke % Draw the box on the paper
1.2 inch 1.5 inch moveto (PPL) show % print PPL inside box
showpage % Eject the page
1 Introduction

Your task is to write an interpreter for a small subset of the language LUCA. You will be given a front-end that performs lexing, parsing, semantic analysis, and intermediate code generation on LUCA source files. You will write an interpreter that reads in the code produced by the front-end, and then executes this code.

1. The interpreter should be implemented using indirect threaded code.

2. You should write your interpreter in C or C++ using gcc.

3. The interpreter should be named lucar. It should read the virtual machine code (in an S-expression format) from standard input.

4. You only have to implement control structures (IF, WHILE, etc.), integer and real arithmetic, a WRITE statement, and array indexing.

5. You should test the interpreter on lecture.

6. You should work in a team of two students.
PostScript

- 1976 - Basic concept conceived by John Warnock.
- 1984 - Warnock (founded Adobe) and released PostScript.
- 1985 - Language of choice for graphical output for printing applications.
- Once de facto standard for distribution of documents meant for publication
PostScript..

- Single control language could be used on any brand of printer.
- Implements rasterization - allows arbitrary scaling, rotating and other transformations
- Display postscript
- Device Independent – both printer and Screen
Language

- Interpreted – No intermediate Byte Code.
- Stack based execution.
- All data including Procedures exist as Objects (Simple and Complex)
- Objects – Type, Attributes and Value
- Dictionaries – systemdict, globaldict, userdict
- 300 built-in operators!
Data Types

- Static
- Integer, Real, Boolean, String, Array.
- Dictionary – associative table.
- String, Array and dictionary complex objects.
- Copy and Type Equivalence.

/\exreal 20.5 def
/\exstring (ravi) def
/\exarray 10 array def
<<1 (LUCA) 2 (C#)>>
/\ex2 exarray def
/\ex3 exreal def
[1 2 3] [1 2 3] eq – false
Scope, Memory Management

- Scope enforced by dictionary, dictionary stack
- Values of Complex objects stored in VM.
- Local VM, Global VM
- Garbage collection

\[ \begin{array}{l}
\text{i}
\text{b}{
1 \text{ dict}
begin
\text{d (PPL method) def}
\text{d show}
end
\} \text{def}
\text{d show} \% \text{d out of Scope - GC}
\end{array} \]
Procedures

- Packed Arrays
- Executable Arrays
- Allows recursion
- Arguments passed via Stack
- Early Binding

```
/ factorial
{ dup 0 eq { 1 }
   { dup
     1 sub
     factorial
     mul
   } ifelse
} def
```

```
/ foo {add mul} bind def
```
Graphics

- Device Space - coordinate space understood by the printer hardware.
- User Space - coordinate system used by PostScript program.
- Current Transformation Matrix (CTM)
- Graphics State – current path, font, CTM
- Store and Restore from graphics stack
Graphics cont..

- Current transformation operators translate, rotate, scale
- Clipping Path – limit the region of the page to draw
Basic Graphics

0 10 360 { % Go from 0 to 360 degrees in 10 degree steps
  newpath % Start a new path
  gsave % Keep rotations temporary
  144 144 moveto
  rotate % Rotate by degrees on stack from 'for'
  72 0 rlineto
  stroke
  grestore % Get back the unrotated state
} for % Iterate over angles

showpage
If you want to know how this is done..
Read our report and cool program 😊
Summary

- Device independent
- Easy to code graphics programming languages
- Stack based interpreted language
- Many built in operators
- Flexible to add new operators, fonts.