Advanced SW Engineering: Scripting Languages

Uwe Zdun
Software Architecture Research Group
Faculty of Computer Science
University of Vienna
http://cs.univie.ac.at/swa

Definition: Script

A computer script is a list of commands that are executed by a certain program or scripting engine

Scripts may be used to automate processes on a local computer or to generate Web pages on the Web

From http://www.techterms.com/definition/script

Definition: Scripting Language

A scripting language or script language is a programming language that supports the writing of scripts

From
http://en.wikipedia.org/wiki/Scripting_language

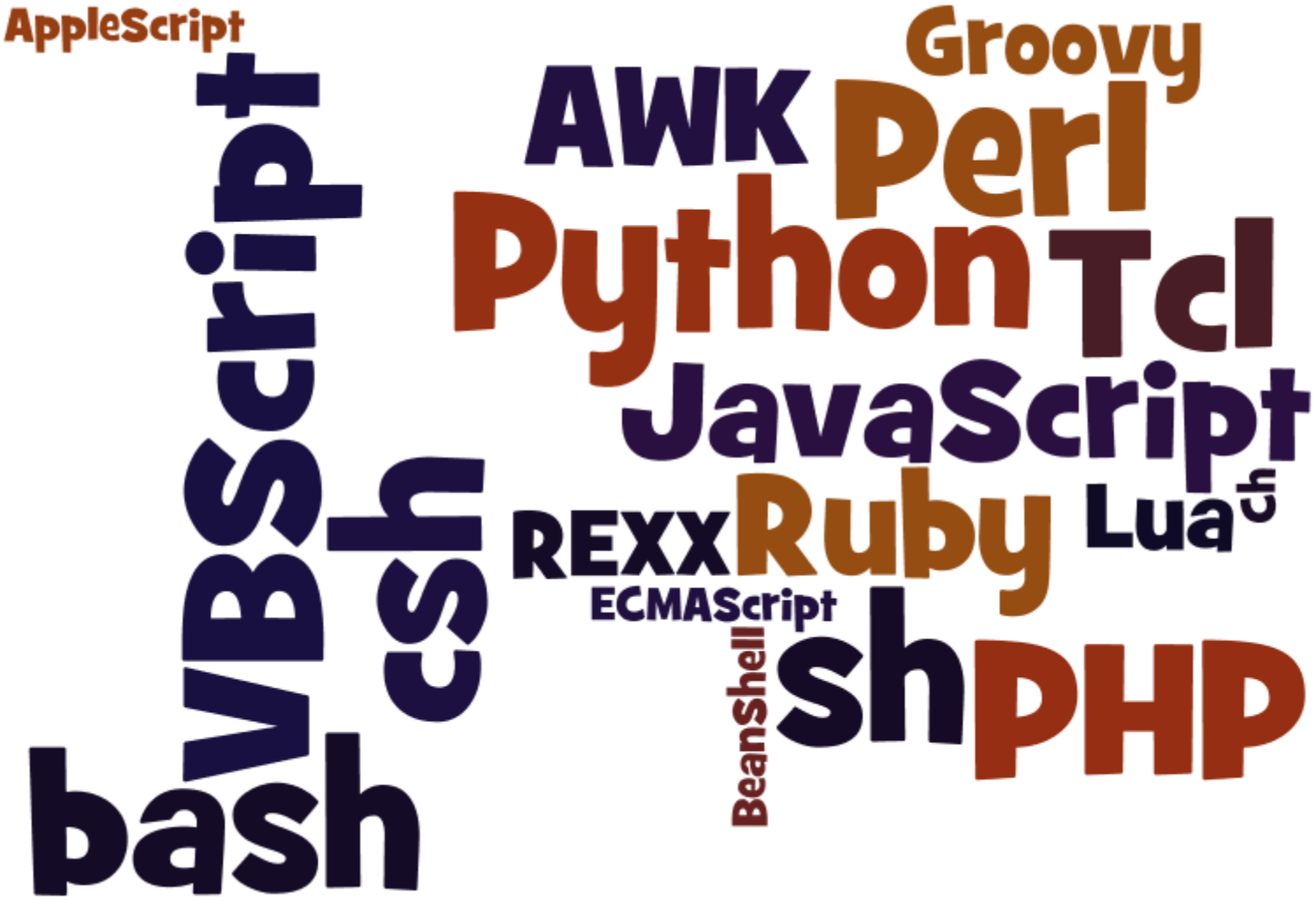
Another Definition: Scripting Language

A scripting language is a programming language designed for integrating and communicating with other programming languages

Some of the most widely used scripting languages are JavaScript, VBScript, PHP, Perl, Python, Ruby, ASP and Tcl. Since a scripting language is normally used in conjunction with another programming language, you often find it alongside HTML or Java, C++.

From
http://en.wikipedia.org/wiki/Scripting_language

Widely Used Scripting Languages



Characteristics

- Text as a basic (or the only) data type

```
set a 1
set b 2
set c [expr "$a$b" + 3]
```

Characteristics

- Minimal use of types, declarations, and other baggage

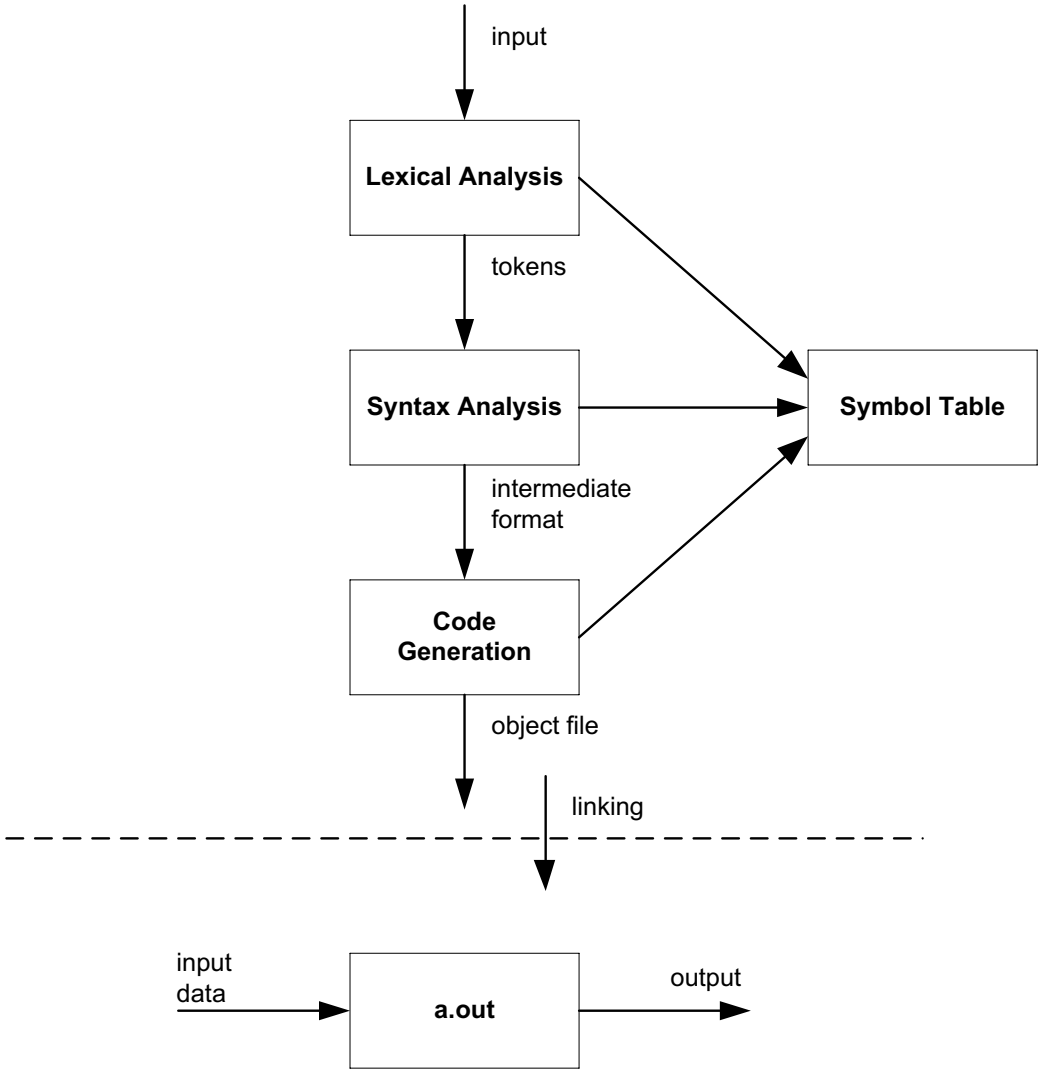
```
set a 1
```

```
proc add {a b} {
    return [expr $a + $b]
}
add 3 4
```

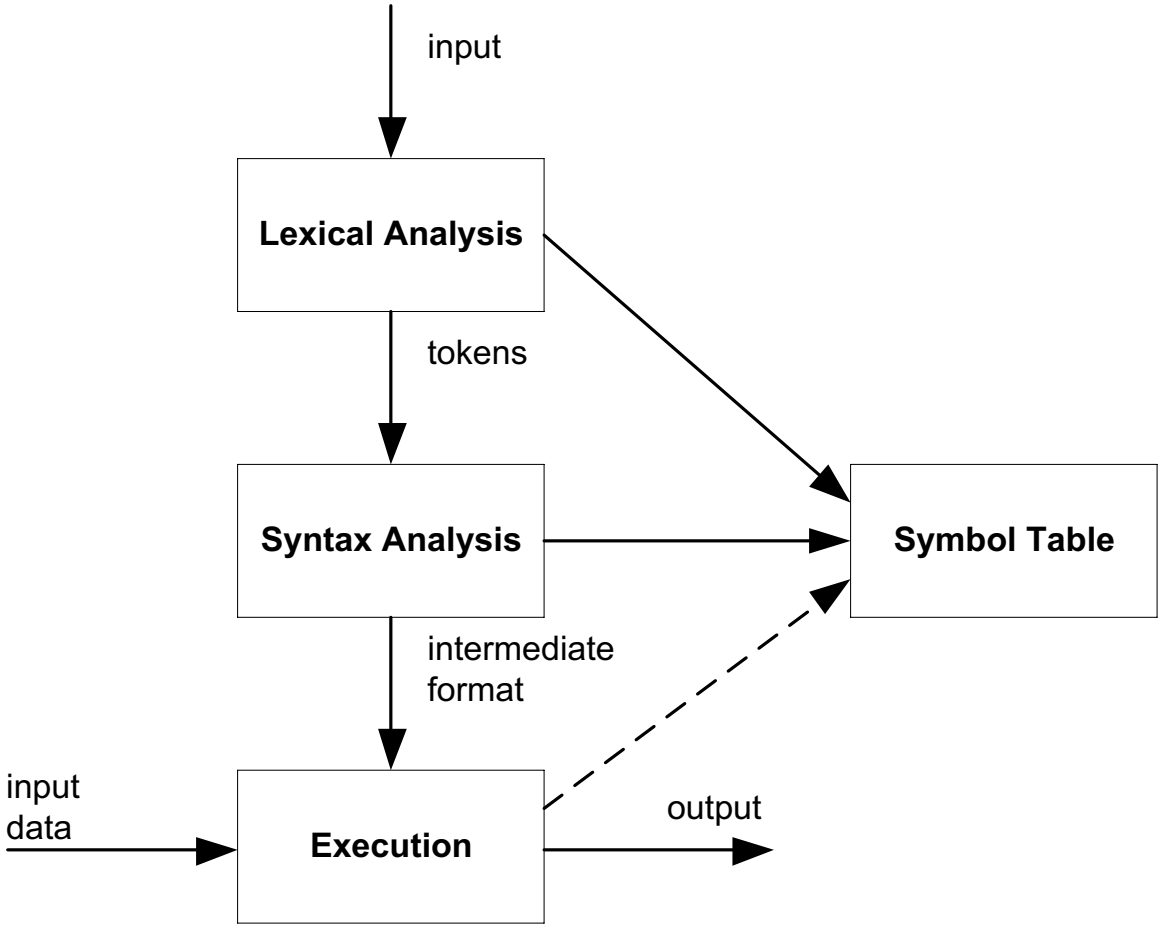
Characteristics

- Usually interpreted instead of compiled

(Simplified) Anatomy of a Compiler



(Simplified) Anatomy of an Interpreter



Often (on the fly) byte code compilation happens in the background

Python function:

```
def myfunc(alist):
      return len(alist)
```

Bytecode:

```
2    0 LOAD_GLOBAL      0 (len)
     3 LOAD_FAST        0 (alist)
     6 CALL_FUNCTION    1
     9 RETURN_VALUE
```

Characteristics

- Regular expression support, perhaps even in syntax

```
set env(DISPLAY) sage:0.1
regexp {([^:]*):} $env(DISPLAY) match host
```

Characteristics

- Associative arrays as a basic aggregate type

```
set commits(Mo) 0
set commits(Tu) 2
set commits(We) 1
puts $commits(Mo)
```

```
set color(0,0) blue
set color(0,1) red
set color(1,0) brown
set color(1,1) blue
puts $color(0,1)
```

Characteristics

- Homoiconic Languages: Evaluate data provided in the language as code

```
set var {$greeting}
set command {puts}
set greeting "Hello World"
eval "$command $var"
```

Examples: Shell

```
#! /bin/bash
echo Backup Started `date` >> ~/backuplog
mkdir /mnt/usbdrive/backups/`date +%Y%m%d`
tar -czf /mnt/usbdrive/backups/`date +%Y%m%d`/data.tar.gz /data
echo Backup Completed `date` >> ~/backuplog
```

Examples: Awk

AWK Script:

```
#!/bin/awk -f
BEGIN { print "File\tOwner" }
{ print $8, "\t", $3}
END { print " - DONE -" }
```

Usage:

```
ls -l | FileOwner
```

Possible Output:

```
File Owner

a.file barnett
another.file barnett
- DONE -
```

Examples: Perl

```
#!/usr/bin/perl
$line = 1;
while (<>) {
  print $line, " ", $_;
  $line = $line + 1; }
```

Examples: UnrealScript

```
PlayerPos.X = (C.Pawn.Location.Y - GameMinimap.MapCenter.Y) / ActualMapRange;
PlayerPos.Y = (GameMinimap.MapCenter.X - C.Pawn.Location.X) / ActualMapRange;

DisplayPlayerPos.X = VSize(PlayerPos) *
        Cos( ATan(PlayerPos.Y, PlayerPos.X) - MapRotation);
DisplayPlayerPos.Y = VSize(PlayerPos) *
        Sin( ATan(PlayerPos.Y, PlayerPos.X) - MapRotation);
```

Excerpt of script code to place player on the Mini Map

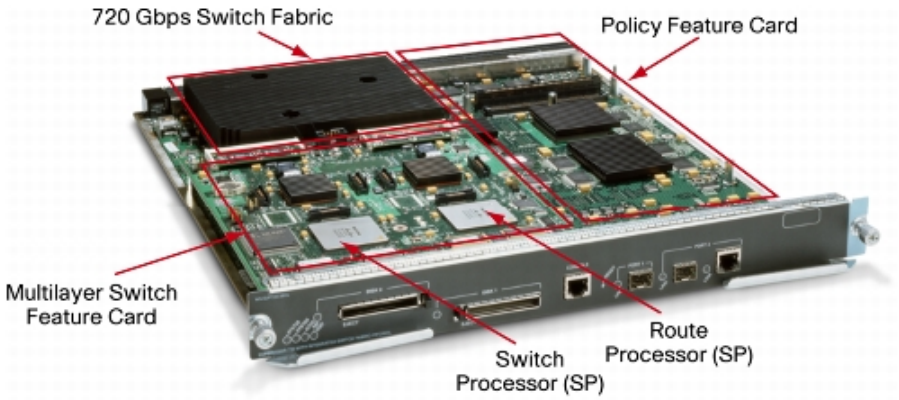
Examples: Lua Game Scripting in World of Warcraft

Example: A function from a Game Mod to remember the sell price for items when not at the merchant.


```
...
function SellValue_MerchantScan(frame)

    for bag=0,NUM_BAG_FRAMES do
        for slot=1,GetContainerNumSlots(bag) do

            local itemName =  InvList_GetShortItemName(bag, slot);
            if itemName ~= "" then
                SellValue_LastItemMoney = 0;
                SellValue_Tooltip:SetBagItem(bag, slot);
                SellValue_SaveFor(bag, slot,
                  itemName, SellValue_LastItemMoney);
            end  -- if item name
        end  -- for slot
    end -- for bag

end
...
```

Examples: Tcl Used to Configure Cisco Routers

```
proc ping_net {x} {
 for {set n 1} {$n<=$x} {incr n 1} {
    exec "ping 172.16.1.$n"
 }
}
```

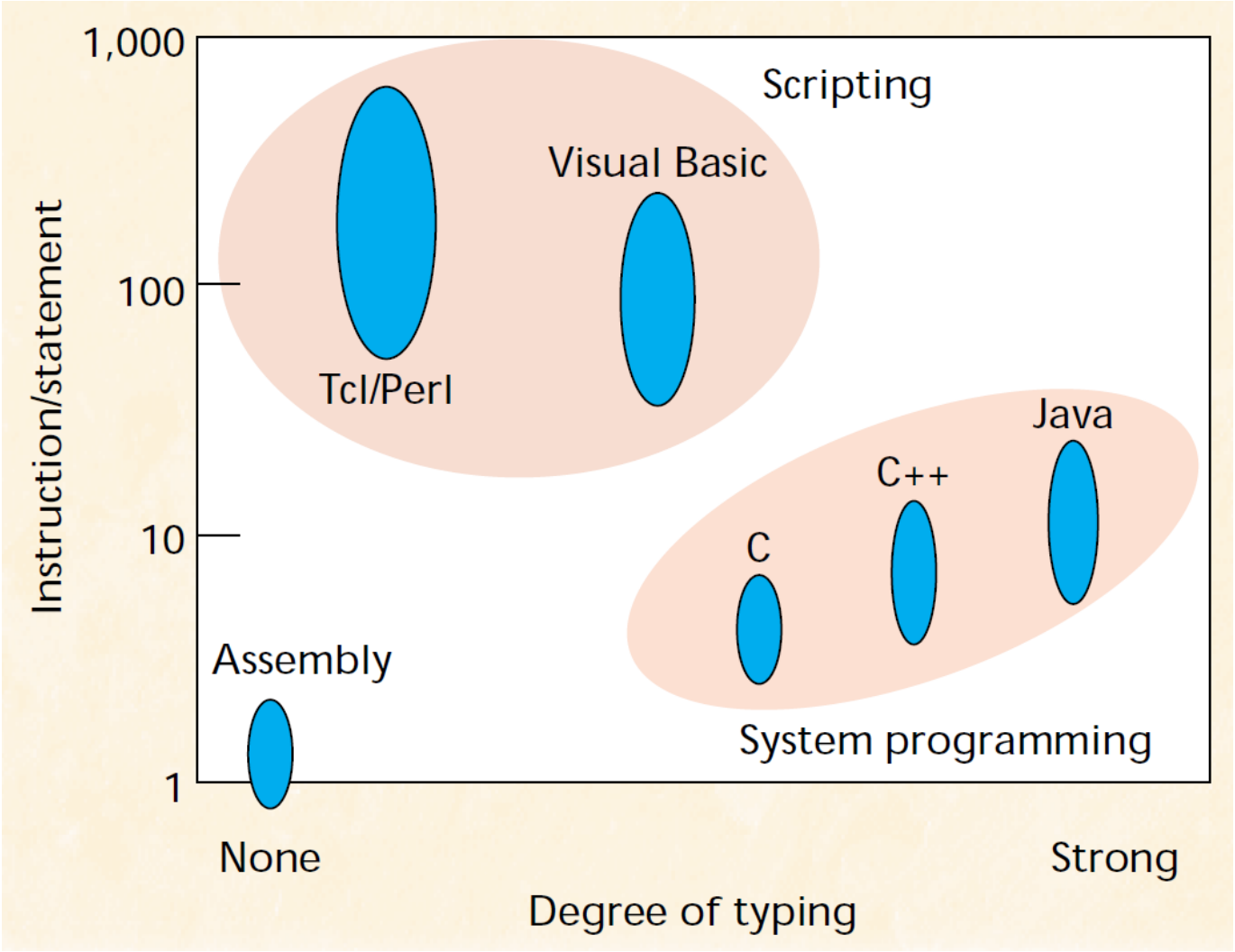



```
Ciscozine(tcl)#ping_net 5

Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 172.16.1.1, timeout is 2 seconds:
!!!!!
Success rate is 100 percent (5/5), round-trip min/avg/max = 1/2/4 ms
Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 172.16.1.2, timeout is 2 seconds:
.....
Success rate is 0 percent (0/5)
Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 172.16.1.3, timeout is 2 seconds:
.....
Success rate is 0 percent (0/5)
Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 172.16.1.4, timeout is 2 seconds:
!!!!!
Success rate is 100 percent (5/5), round-trip min/avg/max = 1/2/4 ms
Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 172.16.1.5, timeout is 2 seconds:
.....
Success rate is 0 percent (0/5)
Ciscozine(tcl)#
```

Typical Application Areas

- Web page scripting
- Web 2.0
- Report generation
- Graphical user interfaces
- System administration
- Configuration
- Game scripting

Ousterhout‘s Dichotomy



Benefits of Scripting Languages According to Ousterhout

- Scripting languages differ from system programming languages in that they are designed for "**gluing**" applications together
- For gluing and system integration, applications can be **developed five to 10 times faster** with a scripting language
- System programming languages require **large amounts of boilerplate and conversion code** to connect the pieces, whereas this can be done directly with a scripting language

Benefits of System Programming Languages According to Ousterhout

- For **complex algorithms and data structures**, the strong typing of a system programming language makes programs easier to manage
- Where **execution speed** is key, a system programming language can often run 10 to 20 times faster than a scripting language because it makes fewer runtime checks

Is Ousterhout‘s dichotomy useful / valid?

- Many believe that the **dichotomy is highly arbitrary**
- Critique 1: Strong-versus-weak typing, data structure complexity, and independent versus stand-alone might be said to be **unrelated features**
- Critique 2: **Focus on distinction of compilation versus interpretation**
  - Neither semantics nor syntax depend significantly on whether code is compiled into machine-language, interpreted, tokenized, or byte-compiled
  - Many languages fall between being interpreted or compiled (e.g. Lisp, Forth, UCSD Pascal, Perl, and Java)

The Casual Programmer Argument

- In recent years, more and more **casual programmers** have joined the programmer community
  - Programming is not their main job function
- Examples of casual programming:
  - Simple database queries
  - Macros for a spreadsheet
- Might be unwilling to spend months learning a programming language
- Scripting languages have a **less steep learning curve**

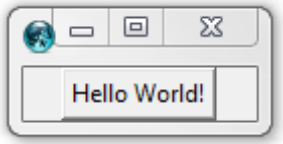
Scripting vs. DSLs

- Both use the casual programmer argument
- Scripting languages are often used to implement interpreted DSLs
- Compared to DSL e.g. in Xtext, DSLs in scripting languages are
  - often internal/embedded DSL
  - easier to realize, especially if language features like for, while, etc. are needed
  - can be tailored (e.g. de-registering commands in Tcl)
  - do not support as sophisticated IDE tooling
  - users might be able to access non-DSL language features

Glueing

- For gluing applications with a scripting language, language **integration with the system language** is necessary
- Options:
  - hand-written integration code
  - wrapper generator
  - reflective integration code

Example: Integration Code for Button Widget in TK




```
# Hello World in Tk
wm title . "Hello"
button .hello -text "Hello World!" -command exit
pack .hello
```

Realized by a Tcl Command with arguments mapped to a (manually written) C function

Example: Integration Code for Button Widget in TK

```
static int ButtonWidgetObjCmd(clientData, interp, objc, objv)
  ClientData clientData;      /* Information about button widget. */
  Tcl_Interp *interp;         /* Current interpreter. */
  int objc;                   /* Number of arguments. */
  Tcl_Obj *CONST objv[];      /* Argument values. */
{
        TkButton *butPtr = (TkButton *) clientData;
        int index;
        int result;
        Tcl_Obj *objPtr;

        if (objc < 2) {
                Tcl_WrongNumArgs(interp, 1, objv,
                        "option ?arg arg ...?");
                return TCL_ERROR;
        }

        // ... map and evaluate arguments
}
```

Using a Wrapper Generator

Example SWIG: SWIG is a compiler that takes C/C++ declarations and creates the wrappers needed to access those declarations from other languages including Perl, Python, Tcl, Ruby, Guile, and Java.

Example: Using SWIG – Some C Code

```
/* File : example.c */

double  My_variable  = 3.0;

/* Compute factorial of n */
int  fact(int n) {
       if (n <= 1) return 1;
       else return n*fact(n-1);
}

/* Compute n mod m */
int my_mod(int n, int m) {
       return(n % m);
}
```

Example: Using SWIG – SWIG interface file

```
/* File : example.i */
%module example
%{
/* Put headers and other declarations here */
extern double My_variable;
extern int    fact(int);
extern int    my_mod(int n, int m);
%}

extern double My_variable;
extern int    fact(int);
extern int    my_mod(int n, int m);
```

Example: Using SWIG – Generating and Using Wrapper Code with Tcl

```
unix > swig -tcl example.i
unix > gcc -c -fpic example.c example_wrap.c -I/usr/local/include
unix > gcc -shared example.o example_wrap.o -o example.so
unix > tclsh
% load ./example.so
% fact 4
24
% my_mod 23 7
2
% expr $My_variable + 4.5
7.5
%
```

Example: Using SWIG – Generating and Using Wrapper Code with Python

```
unix > swig -python example.i
unix > gcc -c -fpic example.c example_wrap.c -
I/usr/local/include/python2.0
unix > gcc -shared example.o example_wrap.o -o _example.so
unix > python
Python 2.0 (#6, Feb 21 2001, 13:29:45)
[GCC egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)] on linux2
Type "copyright", "credits" or "license" for more information.
>>> import example
>>> example.fact(4)
24
>>> example.my_mod(23,7)
2
>>> example.cvar.My_variable + 4.5
7.5
```

Relevant Literature and Sources

- John Ousterhout. Scripting: Higher level programming for the 21st century. IEEE Computer, March 1998. http://www.stanford.edu/~ouster/cgi-bin/papers/scripting.pdf
- Larry Wall. Programming is hard, let's go scripting. December 2007. http://www.perl.com/pub/2007/12/06/soto-11.html
- Brian Kernighan. Random thoughts on scripting languages. http://www.eecs.harvard.edu/cs152/lectures/CS152-Lecture_14-Kernighan.pdf
- Tutorials on the mentioned scripting languages and SWIG

Many thanks for your attention!

Uwe Zdun

Software Architecture Research Group
Faculty of Computer Science
University of Vienna
http://cs.univie.ac.at/swa