



Vývoj aplikací v prostředí .NET

© Katedra řídicí techniky,
ČVUT-FEL Praha

Přednáška – 10. týden

Omluva

*Vzhledem k tomu,
že se předmět přednáší
i v angličtině a není
v mých silách vyrobit
dvojjazyčné prezentace,
nejsou všechny snímky
v češtině.*

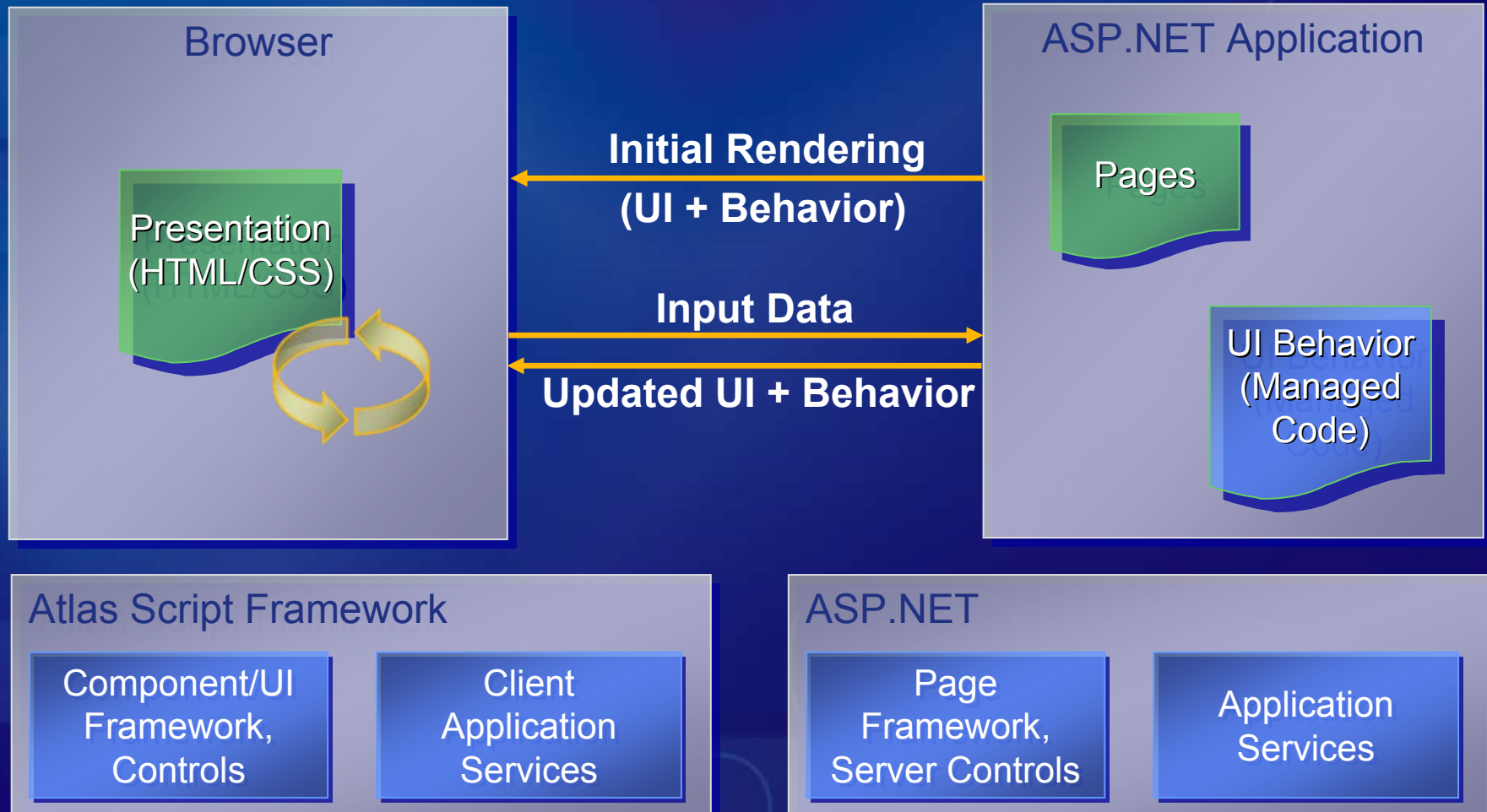
Děkuji za pochopení.



NETwork of ASP.NET II.

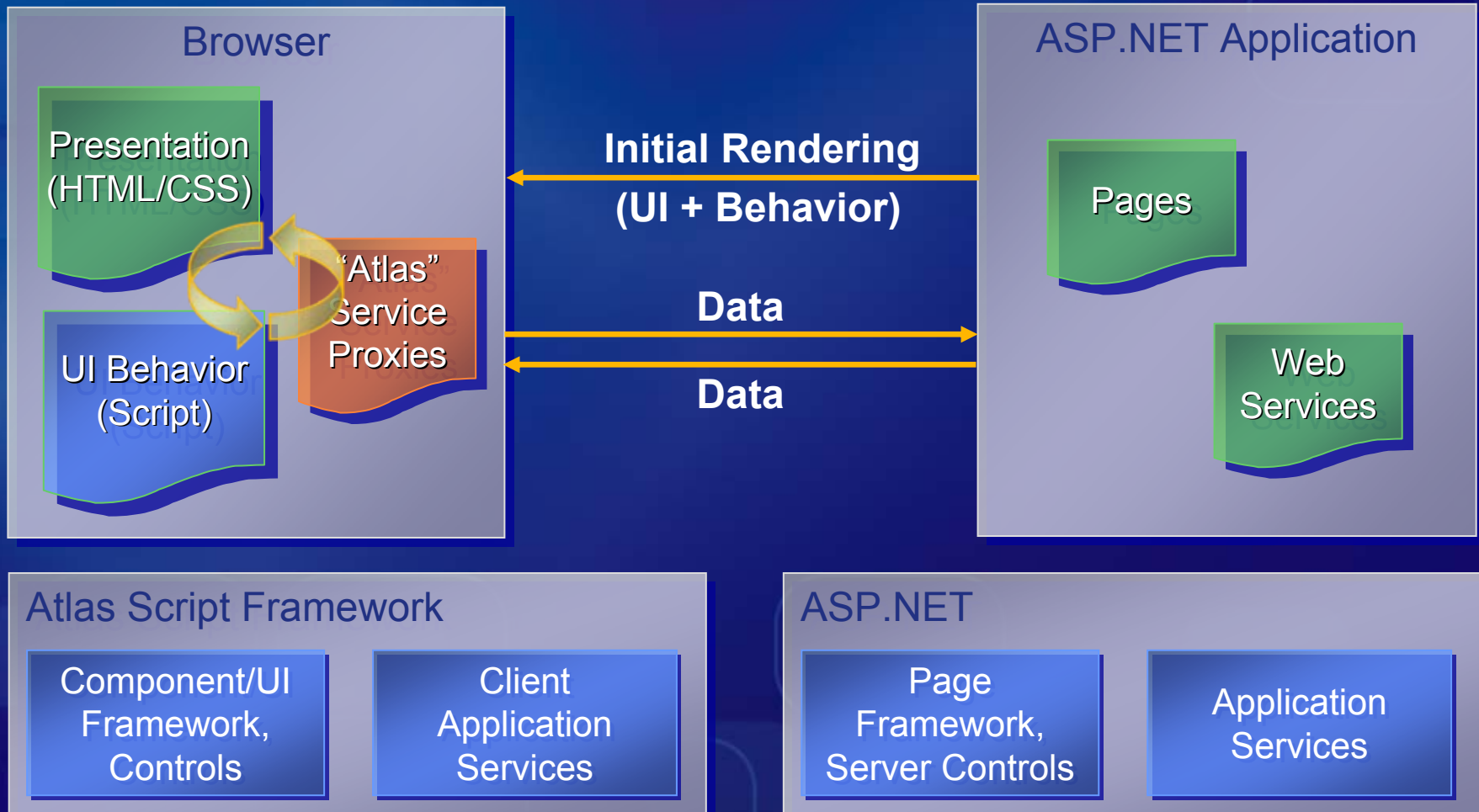


Server-Centric Programming Model



PostBack

Client-Centric Programming Model



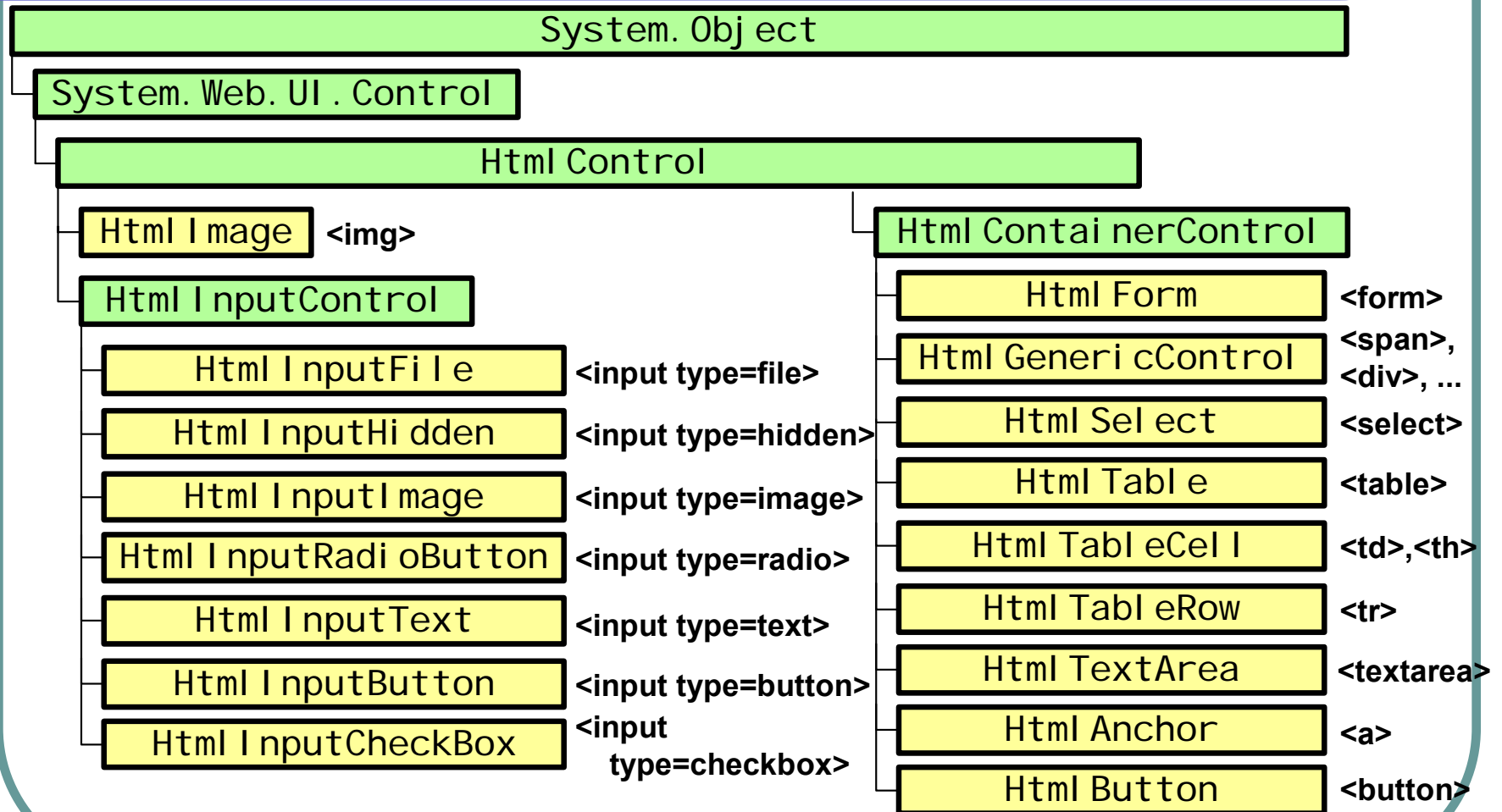
AJAX

Controls

ASP.NET – vytváří dva typy komponent

- *HtmlControls – od základních prvků*
- *WebControls – vyšší abstrakce*

HtmlControls

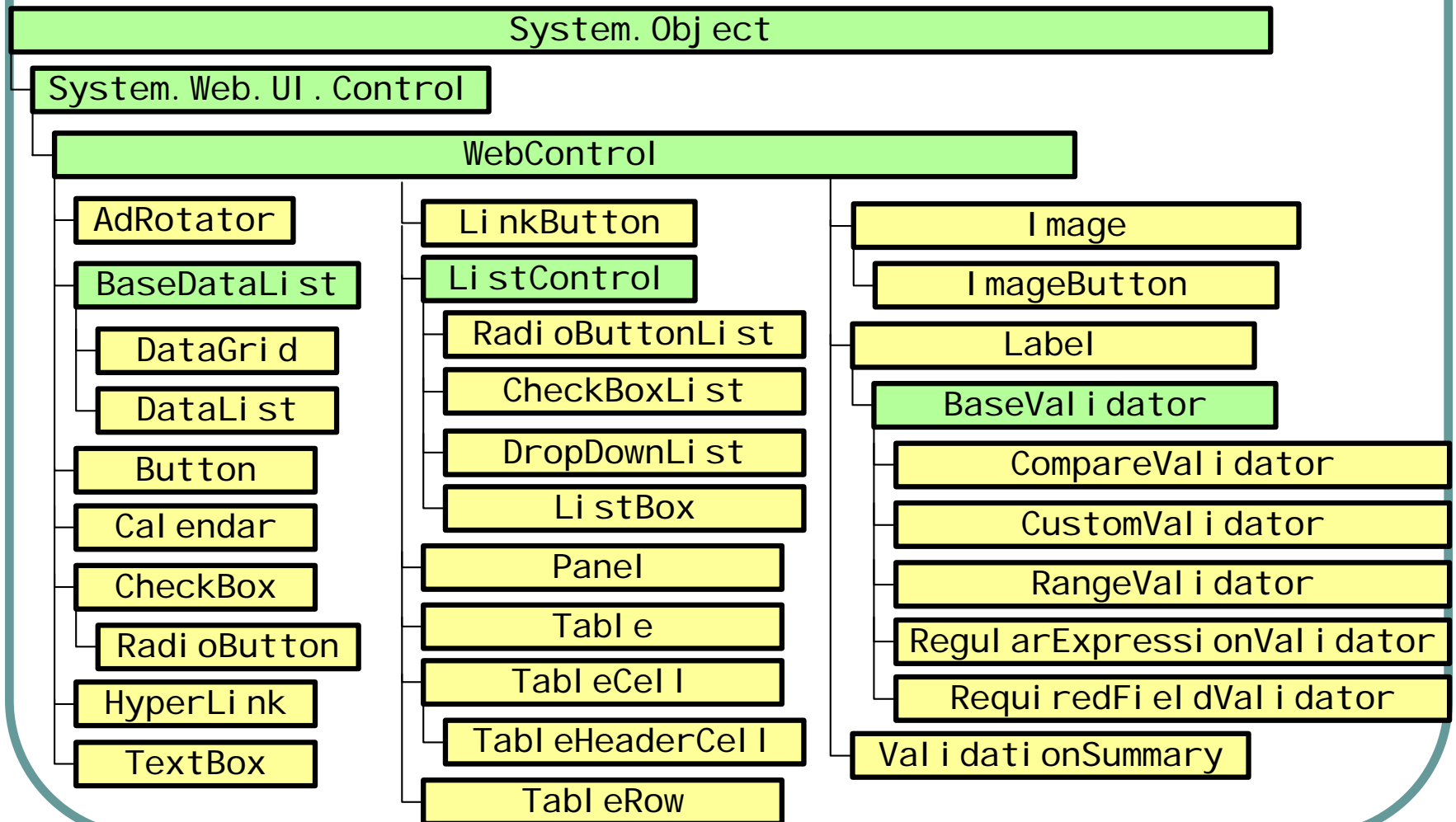


Source: Developmentor ASP.NET Architecture

WebControls

- WebControls provide a more consistent object model and a higher level of abstraction than HtmlControls
 - Most HTML elements can also be represented as **WebControls** on the server
 - **WebControl** versions typically have a more consistent interface (background color is always **BackColor** property whereas in HTML it may be a style attribute (span) or a property (table))
 - **WebControls** also provide higher-level controls with more functionality than primitive HTML elements (like the Calendar control)
 - **WebControls** may render themselves differently based on client browser capabilities

Hierarchy of WebControls



Source: Developmentor ASP.NET Architecture

ASP.NET Web Controls

- webové objekty žijící na serveru
 - vykreslují svoji podobu do html
 - generují události
 - změna hodnoty
 - stisknutí tlačítka
 - zpracovávají události (event handling)
- Web Controls jsou objekty a vývojář stránek
 - pracuje s vlastnostmi a metodami objektů
 - zpracovává události
 - podobně jako v „okénkovém“ programování
- Kód ošetřující událost
 - je vyvolán při zpracování události nebo při zpracování uživatelské akce
 - může být
 - součástí ASP.NET stránky
 - v odděleném souboru (.cs, .vb)
 - zkompileovaný v dll knihovně

HtmlControls

- HtmlControls are server-side representations of standard HTML elements
 - Any HTML element in an ASPX page marked with the **runat=server** attribute will become an HTML control on the server
 - All derive from **HtmlControl** class
 - HTML elements with no distinguished server-side functionality (like div, span, etc.) are all represented as **HtmlGenericControl** instances

Příklad: counter5

Styl typu PHP versus ASP


```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="counter5.aspx.cs" Inherits="counter5" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server"> <title></title> </head>
<body>
    <div>
        You are visitor number <%
int n=Functions.CounterValue(this);
if (n % 2 == 0)
{<%> <font color='Red'><%=n%></font>
    <%> } else Response.Write(n);<%>
    </div>
</body>
</html>
```

Styl HTML komponenta

```
<body>
  <form id="form1" runat="server">
    <div>
      You are visitor number <%
        int n=Functions.CounterValue(this);
        TextCislo.InnerText=n.ToString();
        if(n%2==0) TextCislo.Attributes.CssStyle.Add(
          HtmlTextWriterStyle.Color,"Red");
      %>
      <label id="TextCislo" runat="server"/>
    </div>
  </form>
</body>
```

HTML + Code behind

```
<body>
  <form id="form1" runat="server">
    <div>
      You are visitor number <label id="TextCislo" runat="server"/> !
    </div>
  </form>
</body>
```

```
public partial class counter7 : System.Web.UI.Page
{ // analogie GUI události FormLoad
  protected void Page_Load(object sender, EventArgs e)
  { int n=Functions.CounterValue(this);
    TextCislo.InnerText=n.ToString();
    if(n%2==0) TextCislo.Attributes.CssStyle.Add(
      HtmlTextWriterStyle.Color,"Red");
  }
}
```

Příklad counter8

jako ASP web-control

Web Control

```
<body> <form id="form1" runat="server">
    <div> You are visitor number
    <asp:Label ID="LabelCislo" runat="server"></asp:Label> !
    </div>
</form> </body>
```

```
public partial class counter8 : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        int n = Functions.CounterValue(this);
        LabelCislo.Text = n.ToString();
        if (n % 2 == 0)
            LabelCislo.ForeColor = System.Drawing.Color.Red;
    }
}
```

Zpracování události (PostBack)

- Události generovány klientem, ale zpracovávány serverem
- Události se zapínají vlastností `AutoPostBack`
- Zda jde o první zobrazení nebo o PostBack zjistíte pomocí `Page.IsPostBack`
- S událostmi je třeba šetřit, obzvláště nejsme-li na LAN síti:
 - ☐ Zvyšují zátěž serveru
 - ☐ Zpomalují práci klienta
 - ☐ Ale zajišťují kompatibilitu
 - ☐ Zvyšují bezpečnost kódu
 - ☐ Dovolují využívat zdroje na serveru

V příkladu demonstrujeme:

1. vytvoření ASP.NET webu na lokální síti
 - a) nastavení virtuálního a souborového adresáře
 - b) vložení stránek s příkladem
2. použití PostBack na rychlé síti [PostBack1.aspx]
3. PostBack na pomalé síti [PostBack2.aspx]
4. HtmlEncode
 - ☐ "cross-site scripting (XSS)" – vnucení příkazu serveru
 - ☐ "Script Exploits" – mylné zpracování parametru

Příklad

postback a rychlost sítě

Prohlížeč: zobrazení formuláře

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> <html xmlns="http://www.w3.org/1999/xhtml" >
<head><title>Ahoj</title></head>
<body>
  <form name="_ctl1" method="post" action="ahoj.aspx" id="_ctl1">
    <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
      value="/wEPDwUKMTcxMjA2NDY3M2RknTn2qE2rHrp13liFDQKzGnejEAU=" />
    <input type="submit" name="_ctl3" value="Ahoj" />
    <input name="Jmeno" type="text" maxlength="32" id="Jmeno" />
    <input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
      value="/wEWAwKDo/tJApyE79cOAvb+2ZAK5UIZIPTwyjtmuyv21An04qSXw1w=" />
  </form>
</body>
</html>
```

Změna po stisku tlačítka

```
<input name="Jmeno" type="text" value="Ahoj!" maxlength="32" id="Jmeno" />
```

Programming Basics

Server Control Events

◆ Change events

- By default, **these execute only on next action event**
- E.g. `OnTextChanged`, `OnCheckedChanged`
- Change events fire in random order

◆ Action events

- Cause an **immediate postback to server**
 - ◆ E.g. `OnClick`

◆ Works with any browser

- No client script required, no applets, no ActiveX!

Programming Basics

Wiring Up Control Events

- ◆ Control event handlers are identified on the tag

```
<asp:button onclick="btn1_click" runat=server>  
<asp:textbox onchange="text1_changed" runat=server>
```

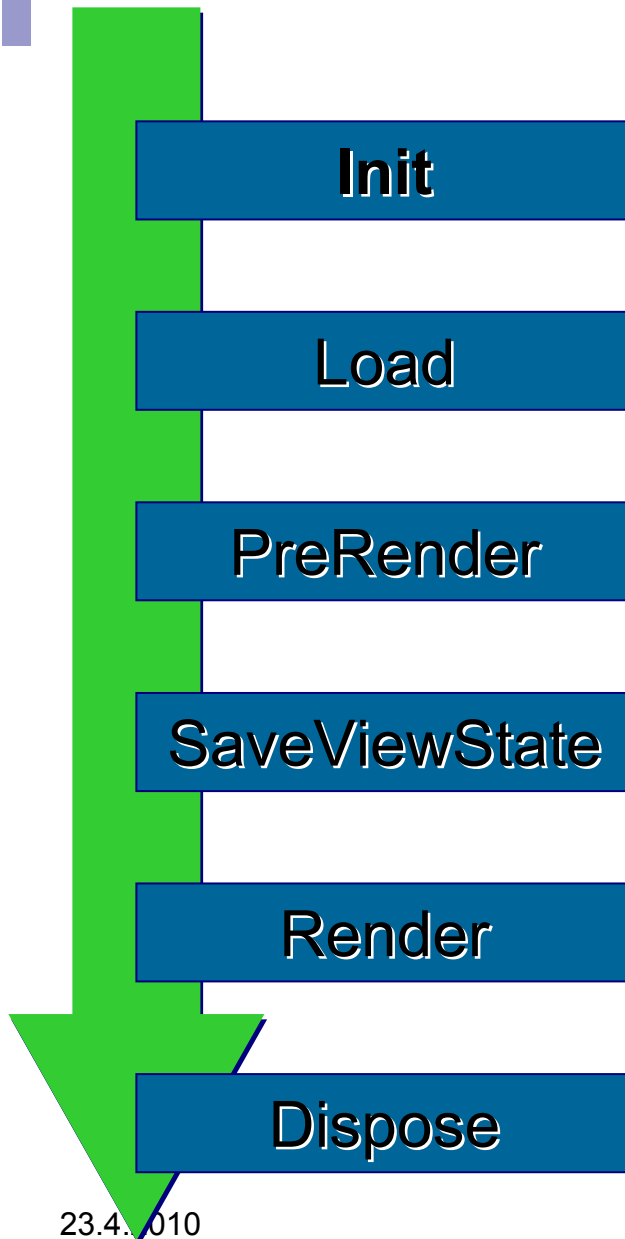
- ◆ Event handler code

```
protected void btn1_Click(Object s, EventArgs e) {  
    Message.Text = "Button1 clicked";  
}
```

Pokročilá témata ASP.NET

Objekty ASP.NET

- Třída **System.Web.UI.Page** vytváří hierarchii ovládacích prvků
 - Page tvoří kořen tohoto stromu
 - Statický text reprezentován jako třída **LiteralControl** ve stromu ovládacích prvků
- Na konci zpracování zavoláno **Page.Render()**
 - Postupuje hierarchií ovládacích prvků
 - Generuje HTML odeslané klientovi
 - Dokáže podporovat podporovat více typů klientů
 - DHTML, HTML 3.2, WML, atd.



Každý prvek je založen, nastaven do počátečního stavu, přidán do stromu ovládacích prvků

Spustí se uživatelský kód a testuje podmínka `!IsPostBack` pro nastavení počátečních hodnot

Provedení posledních změn těsně před uložením stavu a zobrazením
(`EnsureChildControls()` zajistí, že jsou prvky připraveny k vykreslení)

Prvky uloží aktuální stav (pokud je jiný než počáteční)

Každý prvek se vykreslí do výstupního proudu (Response)

Stránka a všechny prvky jsou odstraněny z paměti

- ViewState je kolekce reprezentující stav stránky a všech obsažených ovládacích prvků
- Stav prvků se neukládá na serveru
 - “ViewState” cestuje na klienta a zpět coby skryté pole formuláře
 - Snadná vizualizace pomocí trasování
- Možnost volby – ViewState je:
 - OFF – nastavení hodnoty, výpočty, databáze atd. při každém PostBacku
 - ON – úspora výpočetního času, ale více kB putujících mezi klientem a serverem

Zdroj: [JS]

- ViewState není třeba, pokud
 - ☐ stránka nepoužívá události
 - ☐ nenastavujete vlastnosti ovládacích prvků dynamicky
 - ☐ nebo dynamické vlastnosti se nastavují při každém zobrazení stránky opakovaně
- Vypnutí stavu stránky
 - ☐ Pro ovládací prvek:
`<asp: DataGrid EnableViewState=false ... />`
 - ☐ Pro celou stránku:
`<%@ Page EnableViewState=false ... %>`
 - ☐ Pro aplikaci (v souboru web.config):
`<Pages enableViewState="false" />`

Programový přístup k ViewState

- ViewState se hodí pro ukládání informací pro jednoho uživatele po dobu práce se stránkou aspx
- Využívají ho vestavěné webové ovládací prvky pro uchování hodnot mezi postbacky stránky

//přidání položky do ViewState

ViewState["key"] = hodnota

//čtení položky

Response.Write(ViewState["key"]);

Zpracování stránky/prvků

Post back



Init

LoadViewState

Load

Postback data

Postback events

PreRender

SaveViewState

Render

Dispose

Poslední stav prvku je obnoven z hodnot ViewState

Zaslaná data (z HTTP formuláře) jsou předána do odpovídajících ovládacích prvků

Události jsou spouštěny v pořadí daných stromem, s výjimkou události, která vyvolala odeslání formuláře (Post Back). Ta je volána jako poslední.

Basics events of ASP.NET

Initialize

Page_Init

Restore Control State

Load Page

Page_Load

Control Events

1. Change Events

Textbox1_Changed

2. Action Events

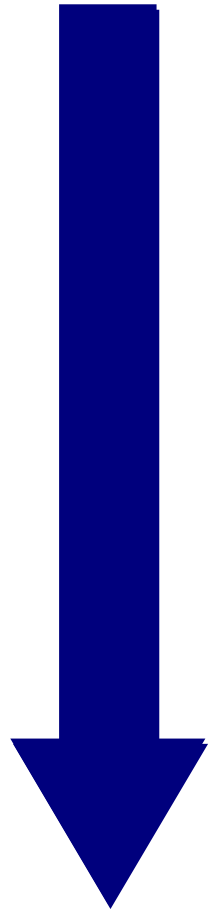
Button1_Click

Save Control State

Render

Unload Page

Page_Unload



- Page_Load fires at beginning of request after controls are initialized
 - Input control values already populated

```
protected void Page_Load(Object s, EventArgs e)
{
    message.Text = textbox1.Text;
}
```

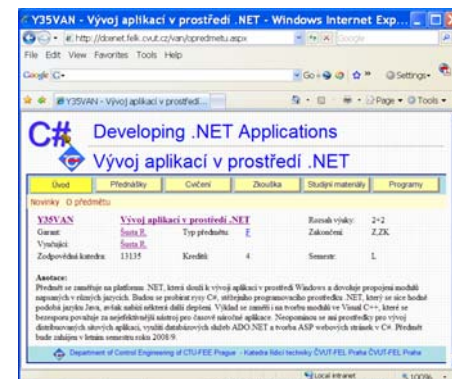
ASP.NET Page Life Cycle Events

Highly recommended reading: [ASP.NET Page Life Cycle Overview](#)

Main Page Events

- [PreInit](#) - *you can: create dynamic elements or set master pages*
- [Init](#) - *you can: initialize control properties*
- [InitComplete](#) – all was initilized
- [PreLoad](#) - all view states were loaded
- [Load](#) – recursively called for all children
- Control events... *ButtonClick, TextChanged*
- [LoadComplete](#)
- [PreRender](#) – *you can make final changes*
- [SaveStateComplete](#) – view states saved, no further changes will be recorded
- [Render](#) - *no event, but processing page*
- [Unload](#) – *event called recursively, you can clean up*

➔ Send to client



- Page_Load fires on every request
 - Use Page.IsPostBack to execute conditional logic
 - If a Page/Control is maintaining state then need only initialize it when IsPostBack is false

```
protected void Page_Load(Object s, EventArgs e)
{
    if (! Page.IsPostBack) {
        // Executes only on initial page load
        Message.Text = "initial value";
    }
    // Rest of procedure executes on every request
}
```

Příklad counter 9

s ViewState a Page.IsPostBack

Counter9

```
<body><form id="form1" runat="server"> <div>
    <asp:Button ID="btnAhoj" runat="server"
        onclick="btnAhoj_Click" Text="Ahoj" />
    <asp:Label ID="LabelCislo" runat="server"></asp:Label>
</div></form></body>
```

```
public partial class counter9 : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!Page.IsPostBack)
            ViewState["visitor"] = Functions.CounterValue(this);
    }
    protected void Button1_Click(object sender, EventArgs e)
    {
        object obj = ViewState["visitor"];
        if (obj is int) {
            int n = (int)obj;
            LabelCislo.Text = n.ToString();
            if (n % 2 == 0)
                LabelCislo.ForeColor = System.Drawing.Color.Red;
        }
    }
}
```

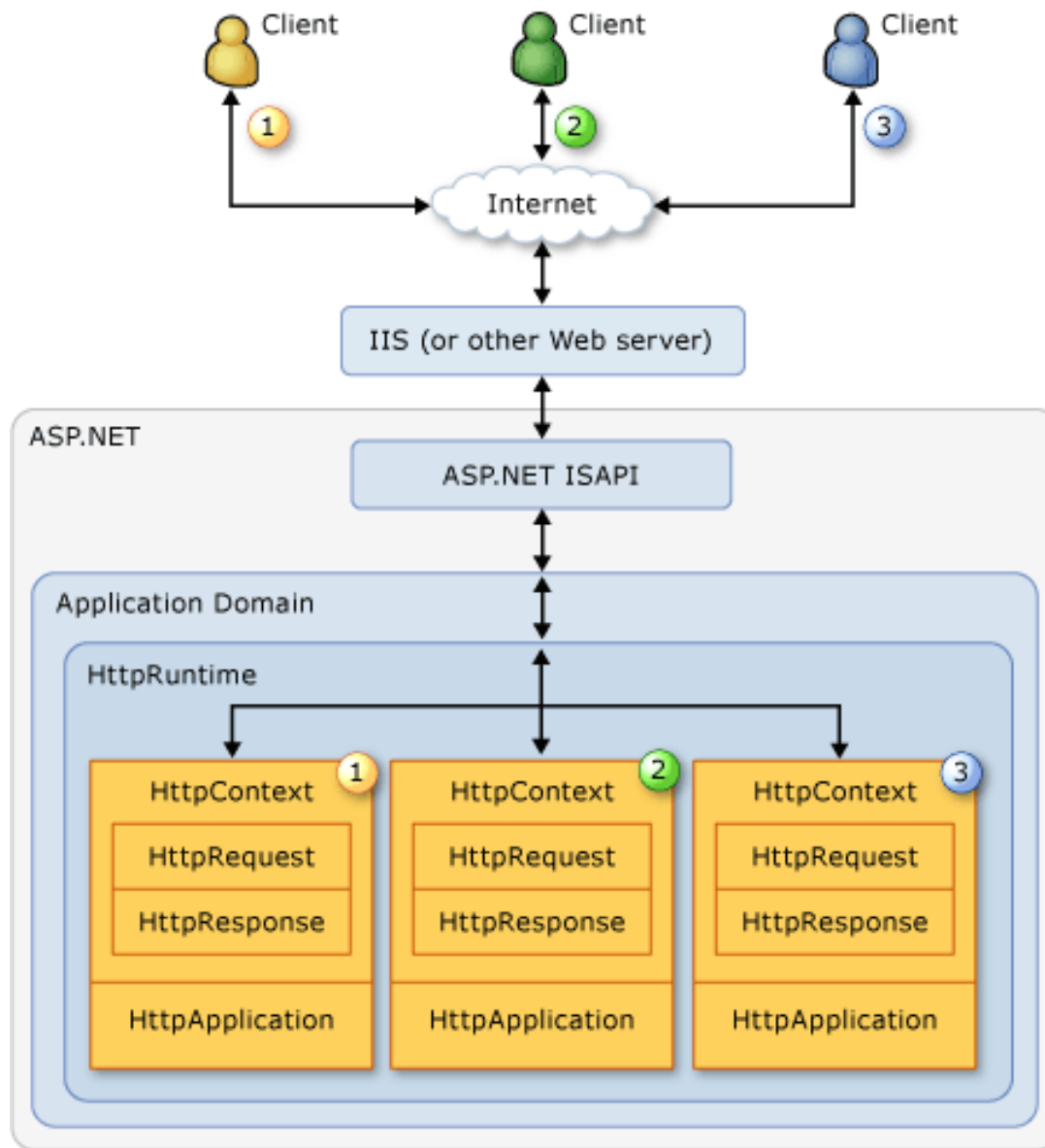
Stav stránky v ASP.NET

- **View state**
- **Session...**

Stavové informace (*State Management*)

- Proč mluvíme o stavu stránky?
 - ☐ HTTP nevytváří žádné trvalé spojení, provádí jednorázový přenos Request → Response
- Kdy potřebujeme uchovávat data?
 - ☐ registrace uživatele
 - ☐ sběr dat rozprostřený na více stránek
 - ☐

HTTPApplication



Zdroj: VS2005 help:
ASP.NET Application
Life Cycle Overview

- data sdílená všemi uživateli ASP.NET aplikace
(*tj. zpravidla příslušného subwebu*)
- ukládají se v asociativním poli jako páry klíč-hodnota
- stav aplikace fyzicky reprezentován property Application
 - (*System.Web.HttpApplicationState*) **System.Web.HttpContext .Application**
- synchronizace přístupu ke stavu aplikace

```
Application.Lock ();
```

```
if(Application["HitCount"] == null) Application["Hitcount"] = 0;
```

```
Application["HitCount"] = (int) Application["HitCount"] + 1;
```

```
Application.UnLock ();
```

- global.asax (*ASP.NET application file – nepovinný soubor pro událostí aplikační úrovně*)

```
void Application_Start() { Application[„HitCount“] = ...načteme ze souboru...; }
```

```
void Application_End() {... uložíme do souboru... = Application[„HitCount“]; }
```

Identifikace série požadavků pocházejících od jednoho uživatele

- stav session fyzicky reprezentován property Session
 - (*System.Web.HttpSessionState*) **System.Web.HttpContext.Session**
 - hodnota též dostupná i přes objekt Page
(*System.Web.HttpSessionState*) **System.Web.UI.Page.Session**

Omezení v klasickém ASP:

- nekompatibilita s webovými farmami
- zničení Session při restartu IIS nebo rebootování serveru
- prohlížeče bez cookies

■ global.asax

```
void Session_Start() {  
    // initialize  
    Session["Name"] = "";  
}
```

■ write_session_state.aspx

```
void Page_Load(Object Src, EventArgs e) {  
    Session["Name"] = TextBox1.Text;  
}
```

■ read_session_state.aspx

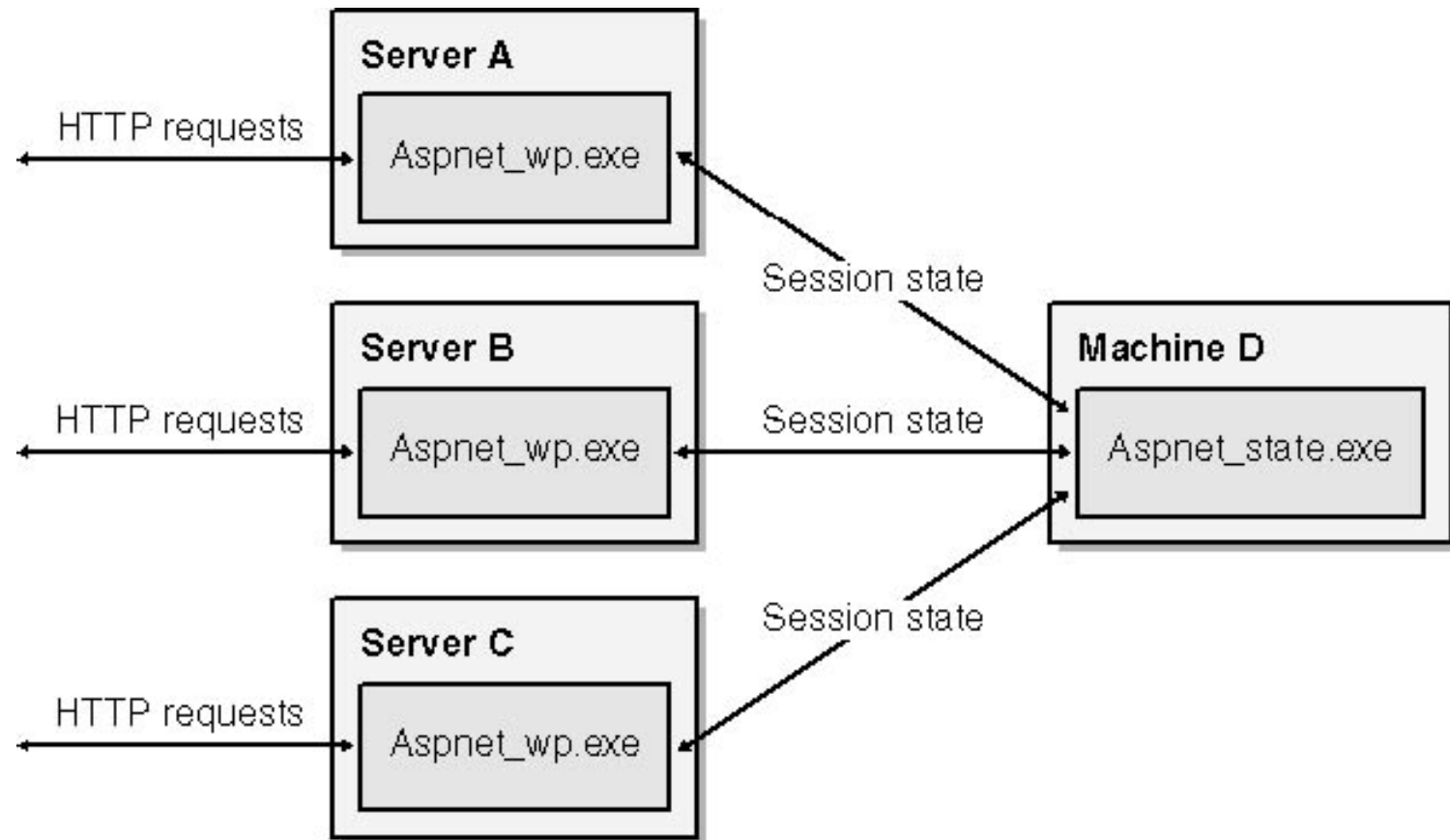
```
void Page_Load(Object Src, EventArgs e) {  
    TextBox1.Text = (string)Session["Name"];  
}
```

- „Původní“ ASP – ukládání vždy v paměti,
- ASP.NET – 3 modely, nastavuje se ve Web.config
 - In-proc

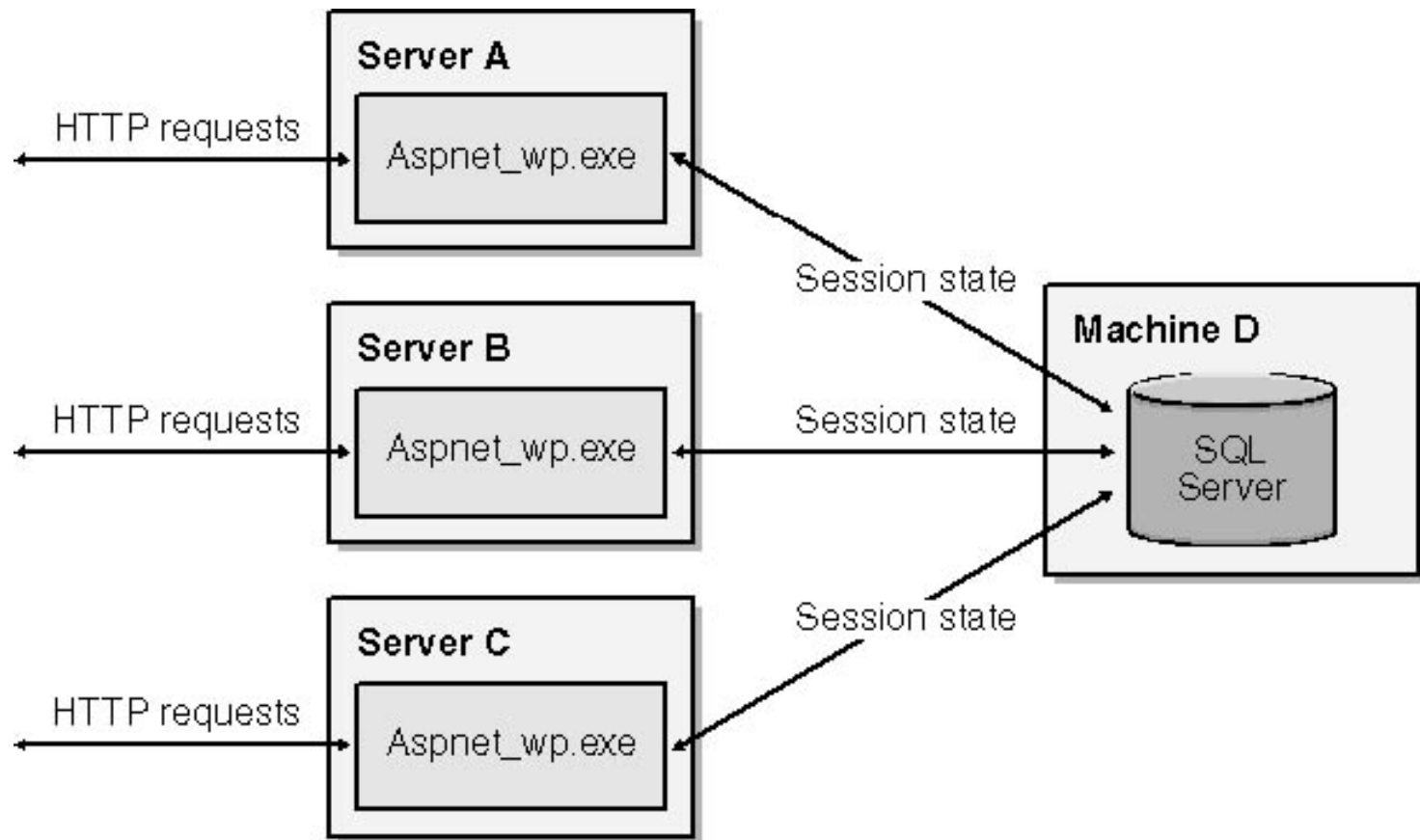
Aspnet_wp.exe, odpovídá klasickému modelu, je nejrychlejší
 - State server

mimo Aspnet_wp.exe, instance služby Aspnet_state.exe, pomalejší, náročnější na výkon
 - SQL server

mimo Aspnet_wp.exe, v databázi serveru SQL, uchování dat i při výpadcích



Zdroj: [JS]



Zdroj: [JS]

Funkcionální programování



"Scientists confirmed today that everything we know about the structure of the universe is wrongedy-wrong-wrong."

[The New Yorker Collection 1998, Jack Ziegler]

C# tries to implement some features of functional programming

C# 3.0 Language Innovations

```
var contacts =  
  from c in customers  
  where c.State == "WA"  
  select new { c.Name, c.Phone };
```

Query
expressions

Expression
Trees

Automatic
Properties

Partial
Methods

Lambda
expressions

```
var contacts =  
  customers  
  .Where(c => c.State == "WA")  
  .Select(c => new { c.Name, c.Phone });
```

Local variable
type inference

Extension
methods

Anonymous
types

Object
initializers

4 main programming paradigms

Angl. "paradigm", mn. číslo "paradigm", ale i "paradigmata"
pochází z řeckého "paradeigma" (v latině paradigma)
para – vztah k něčemu, deiknynai – ukázat
Význam: – standardní nebo typický příklad, vzor

- Imperative paradigm

First do this and next do that...

- Functional paradigm

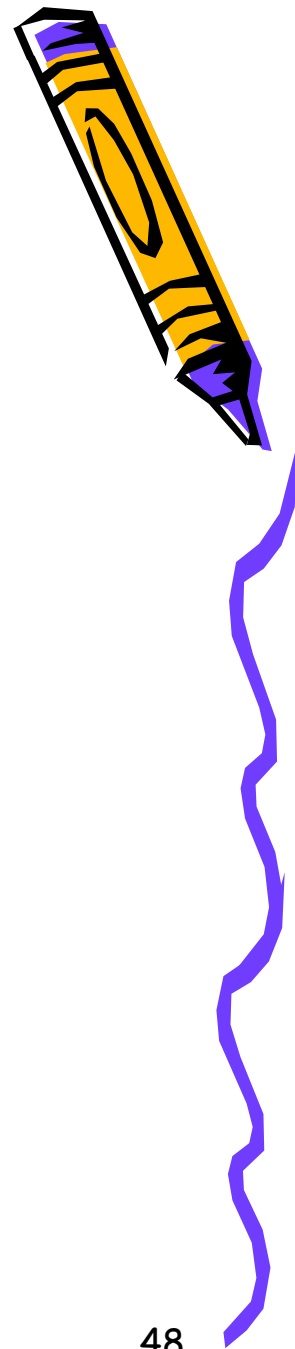
Evaluate a function and then its results...

- Logic paradigm

Answer a question via search for a solution...

- Object-oriented paradigm

Use objects and their interactions



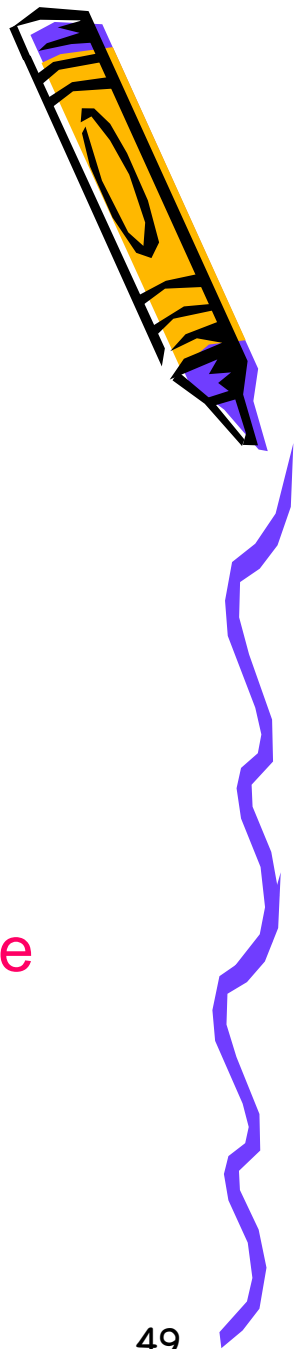
Imperative versus Functional

The design of the **imperative languages** is based directly on the von Neumann architecture.

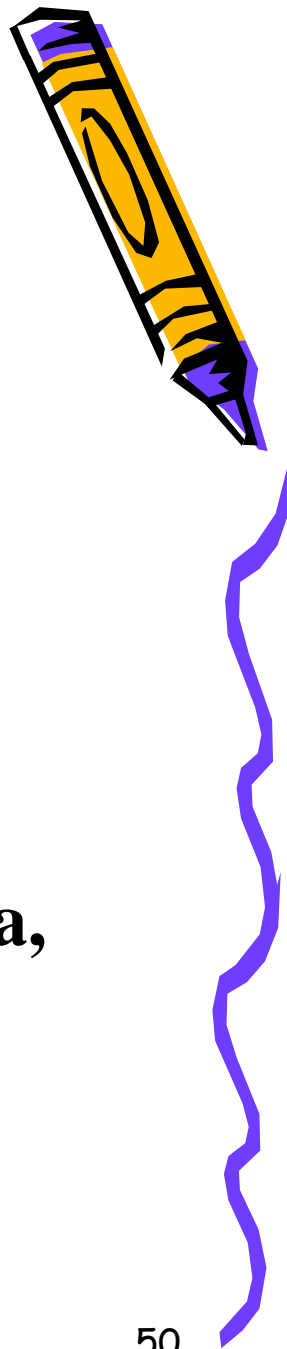
Efficiency is the primary concern, which leads to **a lesser suitability** of the language for software development.

The design of the functional languages is based on mathematical functions.

A solid theoretical basis that is also **closer to the user**, but relatively **unconcerned with the architecture** of the machines on which programs will run.



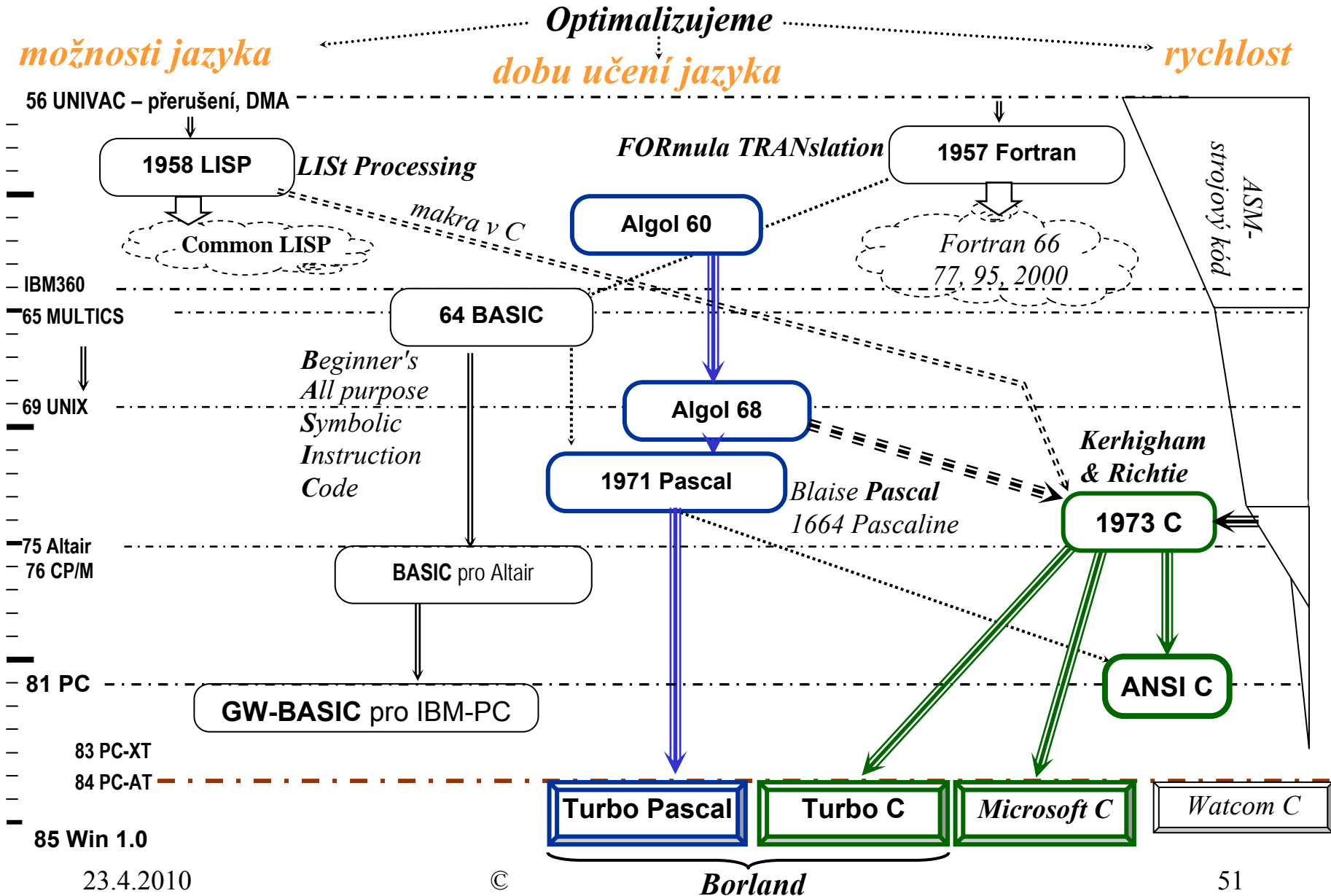
Functional Programming



- **Idea: everything is a function**
- **Based on sound theoretical frameworks (e.g., the lambda calculus)**
- **Examples of FP languages**
 - **First (and most popular) FP language: Lisp**
 - **Other important FPs: ML, Haskell, Miranda, Scheme, Logo**



Programování do roku 1985



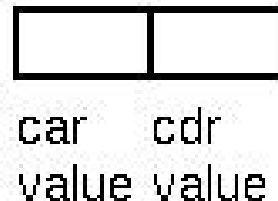
LISP: data mají stejnou strukturu jako program

=> funkce může vracet funkci

navrhnul 1960 McCarthy z MIT

Názvy car, cdr pocházejí z IMB implementace
Content of Address/Data Register

Basic object is the list structure:



Examples:

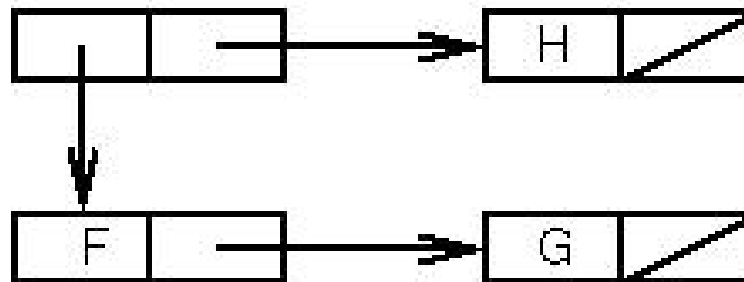
(A B)



(C D E)



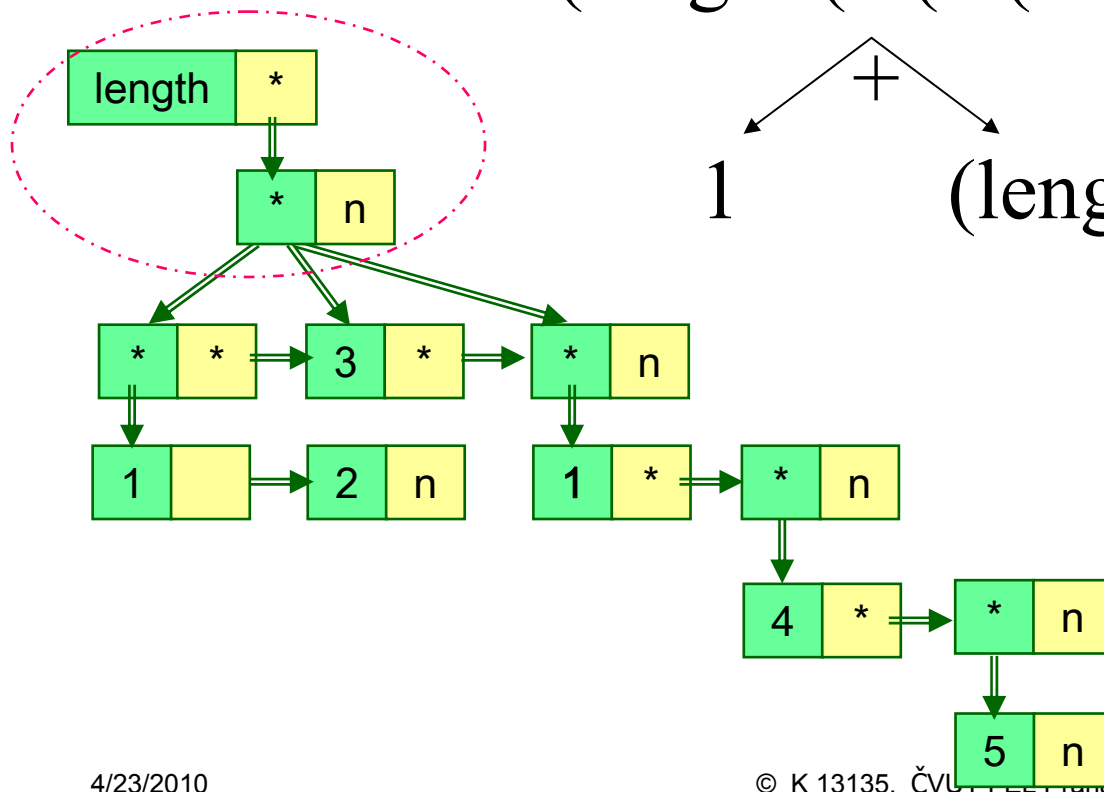
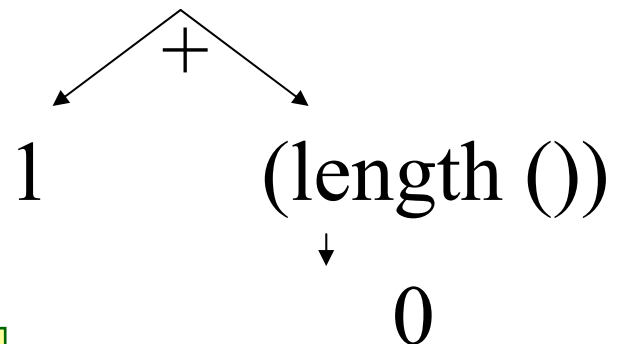
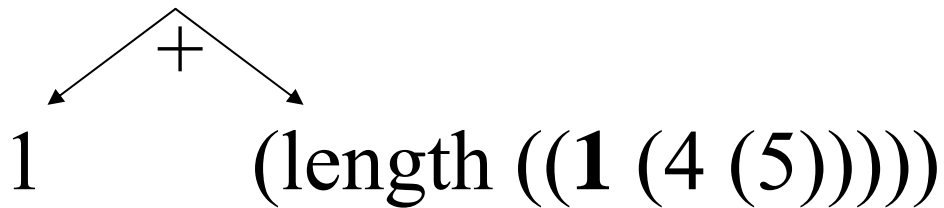
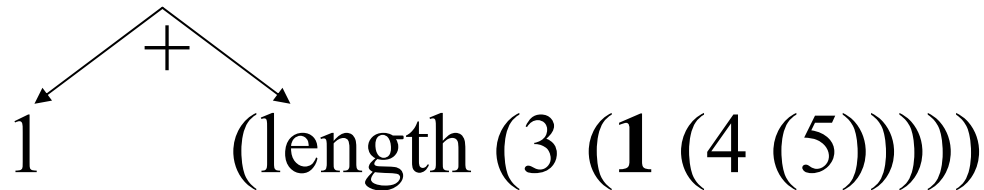
((F G) H)



FP: Linear or recursion



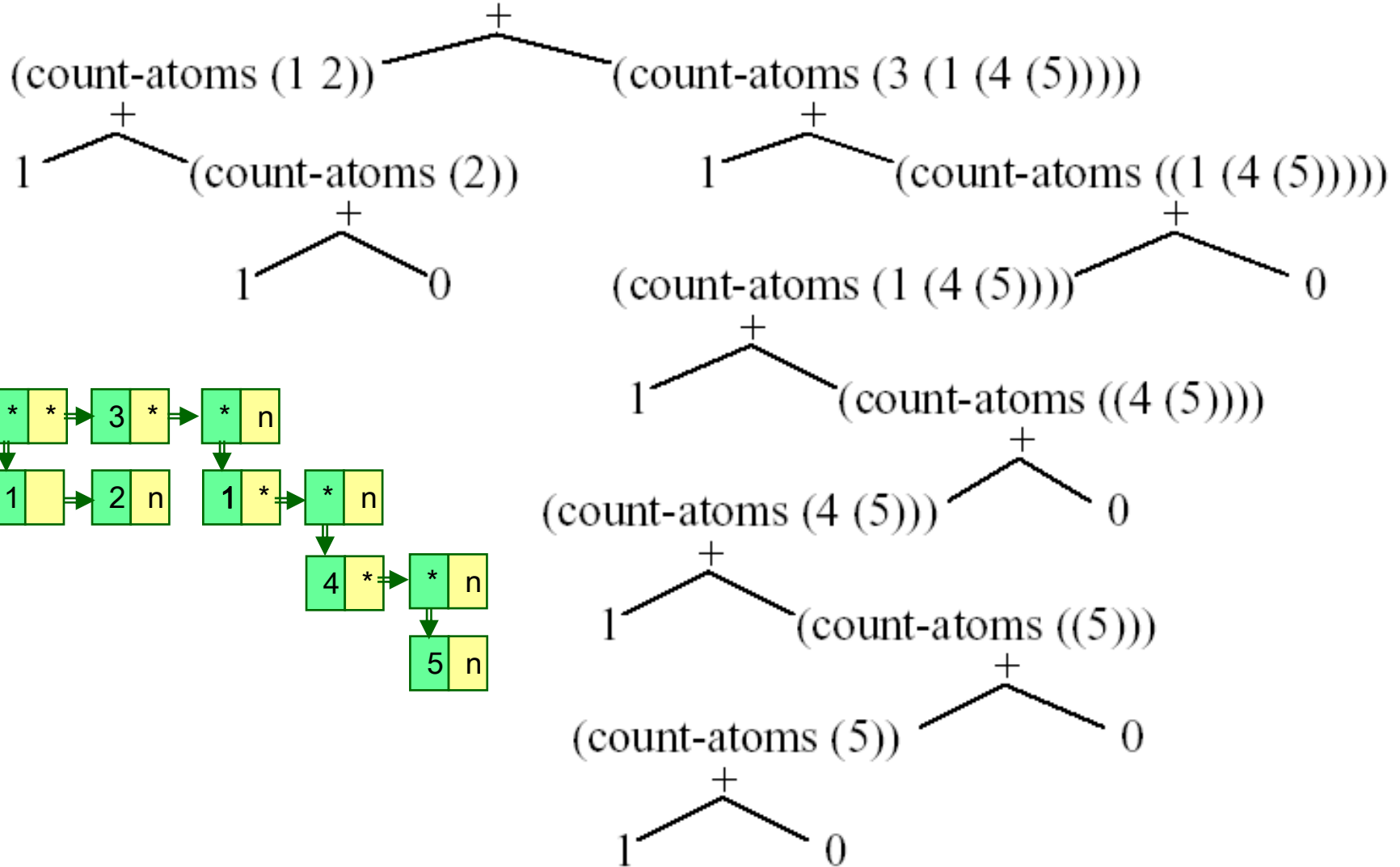
$(\text{length } ((1\ 2)\ 3\ (1\ (4\ (5))))) \longrightarrow 3$



FP: Tree of recursion



(count-atoms ((1 2) 3 (1 (4 (5))))) → 6



Lze řešit i imperativními jazyky



Definice seznamu

```
class List // definice prvku seznamu
```

```
{  Object _car=null;  // string nebo odkaz na List
    List _cdr=null;
```

```
public List(Object car, List cdr) { this._car = car; this._cdr = cdr; }
```

```
public List car { get { return _car as List; } }
```

```
public String atom { get { return _car as string; } }
```

```
public List cdr { get { return _cdr; } }
```

```
public override string ToString()
```

```
{ return (atom != null ? atom
```

```
                : (car!=null ? "(" + car.ToString() + ")" : ""))
```

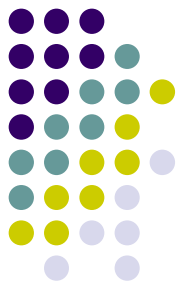
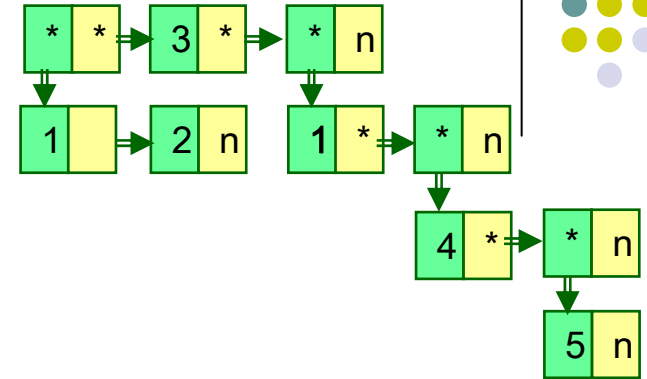
```
    + (cdr != null ? cdr.ToString() : "");
```

```
}
```

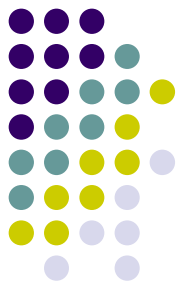
```
} 4/23/2010
```

Vybudování seznamu

```
List seznam = new List(  
    new List("1", new List("2", null)),  
    new List("3",  
        new List(new List("1",  
            new List(new List("4", new List(new List("5", null), null)),  
                null)),  
            null))  
        );  
string s = seznam.ToString(); // s = "(12)3(1(4(5)))"
```



Rekursivní funkce



```
static int Length(List list)
{ if (list != null) return 1+Length(list.cdr);
  else return 0; }
```

```
static int CountAtoms(List list)
{ if (list == null) return 0;
  if(list.atom!=null) return 1+CountAtoms(list.cdr);
  else return CountAtoms(list.car)+CountAtoms(list.cdr);
}
```

```
/*...*/
```

```
int i = Length(seznam);           // i=3
```

```
int j = CountAtoms(seznam);       // j=6
```


Je C# FP jazyk na úrovni LISPU?

- Moc ne, ale F# ano
- Jeho základem jsou Lambda výrazy



Lambda calculus - *plur.* calculi / calculuses

česky Lambda kalkul



Expressions and Functions

◆ Expressions

$x + y$

$x + 2 * y + z$

◆ Functions

$\lambda x. (x + y)$

$\lambda z. (x + 2 * y + z)$

◆ Application

$(\lambda x. (x + y)) 3 = 3 + y$

$(\lambda z. (x + 2 * y + z)) 5 = x + 2 * y + 5$

Parsing: $\lambda x. f (f x) = \lambda x. (f (f (x)))$

Higher-Order Functions

- ◆ Given function f that returns function $f \circ f$

$\lambda f. \lambda x. f (f x)$

- ◆ How does this work?

$$\begin{aligned} & (\lambda f. \lambda x. f (f x)) (\lambda y. y+1) \\ &= \lambda x. (\lambda y. y+1) ((\lambda y. y+1) x) \\ &= \lambda x. (\lambda y. y+1) (x+1) \\ &= \lambda x. (x+1)+1 \end{aligned}$$

Same result if step 2 is altered.

Pro zajímavost: stejný postup v Lisp syntaxi

Given function f , return function $f \circ f$

```
(lambda (f) (lambda (x) (f (f x))))
```

How does this work?

```
((lambda (f) (lambda (x) (f (f x)))) (lambda (y) (+ y 1)))
```

```
= (lambda (x) ((lambda (y) (+ y 1))  
               ((lambda (y) (+ y 1)) x))))
```

```
= (lambda (x) ((lambda (y) (+ y 1)) (+ x 1))))
```

```
= (lambda (x) (+ (+ x 1) 1))
```

Same procedure, v C-like syntax

◆ Given function f , return function $f \circ f$

```
function (f) { return function (x) { return f(f(x)); } ; }
```

◆ How does this work?

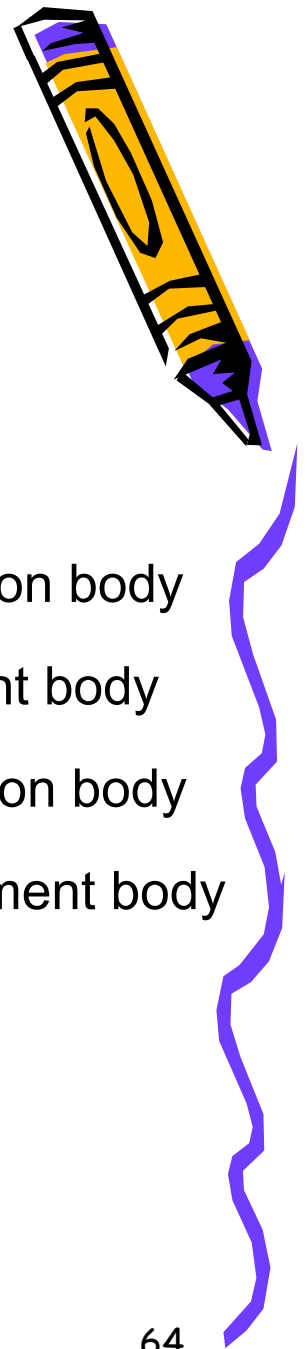
```
(function (f) { return function (x) { return f(f(x)); } ; }  
  (function (y) { return y + 1; }))
```

```
function (x) { return (function (y) { return y + 1; })  
                      ((function (y) { return y + 1; })  
                       (x)); }
```

```
function (x) { return (function (y) { return y + 1; }) (x +  
1); }
```

```
function (x) { return ((x + 1) + 1); }
```

C# Lambda Expressions



Expression or statement body

Implicitly or explicitly typed parameters

- Examples:

```
x => x + 1
```

// Implicitly typed, expression body

```
x => { return x + 1; }
```

// Implicitly typed, statement body

```
(int x) => x + 1
```

// Explicitly typed, expression body

```
(int x) => { return x + 1; } // Explicitly typed, statement body
```

```
(x, y) => x * y
```

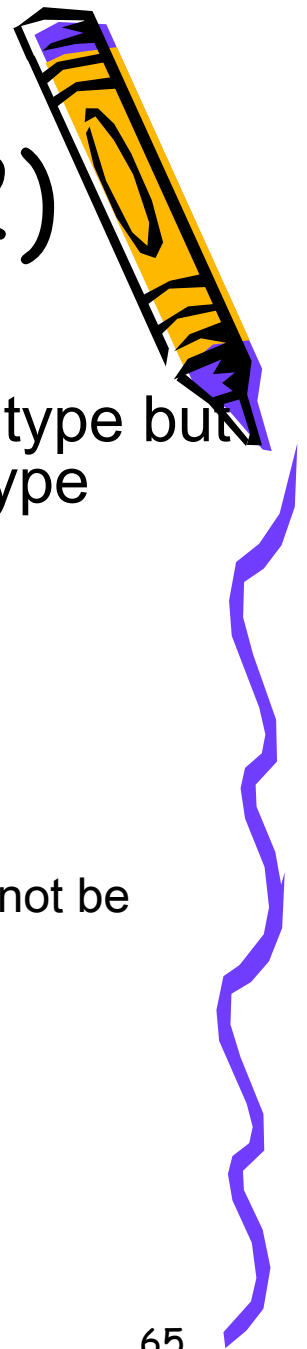
// Multiple parameters

```
() => Console.WriteLine()
```

// No parameters



C# Lambda Expressions (2)



- A lambda expression is a value, that does not have a type but can be implicitly converted to a compatible delegate type

```
delegate R Func<A,R>(A arg);
```

```
Func<int,int> f1 = x => x + 1;           // Ok
```

```
Func<int,double> f2 = x => x + 1;       // Ok
```

```
Func<double,int> f3 = x => x + 1;      // Error – double cannot be  
// implicitly converted to int
```



Usage of Lambda Expressions 1/4

```
public delegate bool Predicate<T>(T obj);

public class List<T>
{
    public List<T> FindAll(Predicate<T> test) {
        List<T> result = new List<T>();
        foreach (T item in this)
            if (test(item)) result.Add(item);
        return result;
    }
    ...
}
```

Usage of Lambda Expressions 2/4

```
public class MyClass
{
    public static void Main() {
        List<Customer> customers = GetCustomerList();
        List<Customer> locals =
            customers.FindAll(
                new Predicate<Customer>(StateEqualsWA)
            );
    }

    static bool StateEqualsWA(Customer c) {
        return c.State == "WA";
    }
}
```

Usage of Lambda Expressions 3/4

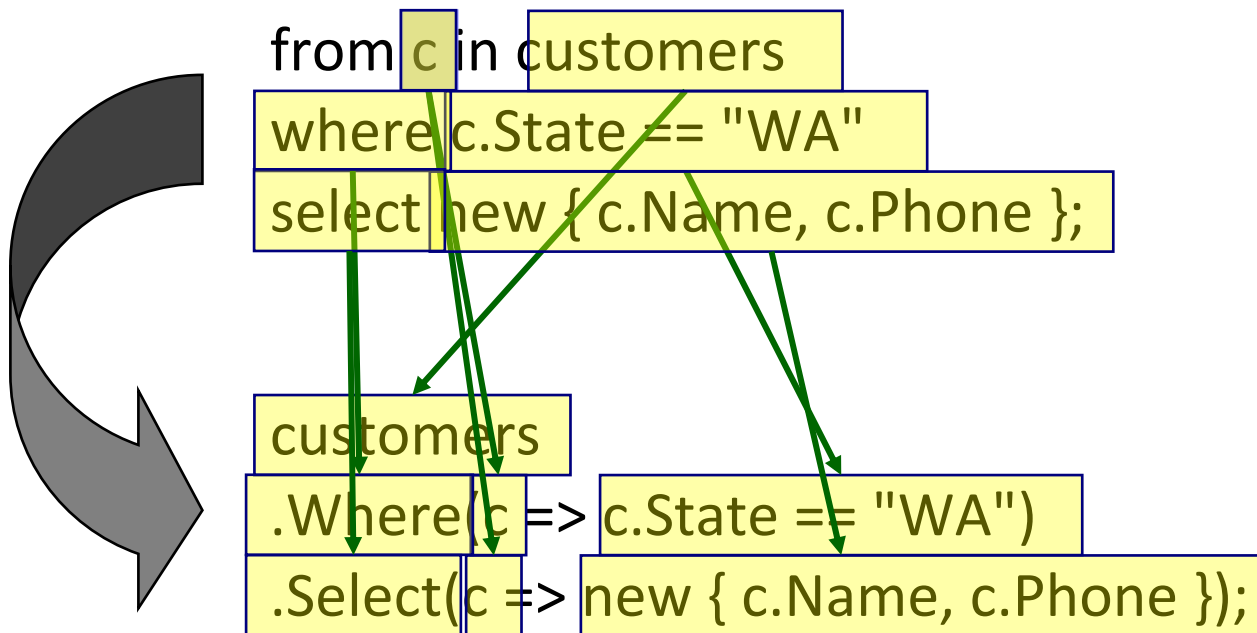
```
public class MyClass
{
    public static void Main() {
        List<Customer> customers = GetCustomerList();
        List<Customer> locals =
            customers.FindAll(
                delegate(Customer c) { return c.State == "WA"; }
            );
    }
}
```

Usage of Lambda Expressions 4/4

```
public class MyClass
{
    public static void Main() {
        List<Customer> customers = GetCustomerList();
        List<Customer> locals =
            customers.FindAll(c => c.State == "WA");
    }
}
```

Lambda expression

- Queries translate to method invocations
 - Where, Join, OrderBy, Select, GroupBy, ...



Query Expressions

Starts with
from

Zero or more **from**,
join, **let**, **where**, or
orderby

from *id* **in** *source*
{ **from** *id* **in** *source* |
join *id* **in** *source* **on** *expr* **equals** *expr* [**into** *id*] |
let *id* = *expr* |
where *condition* |
orderby *ordering*, *ordering*, ... }
select *expr* | **group** *expr* **by** *key*
[**into** *id query*]

Ends with **select**
or **group by**

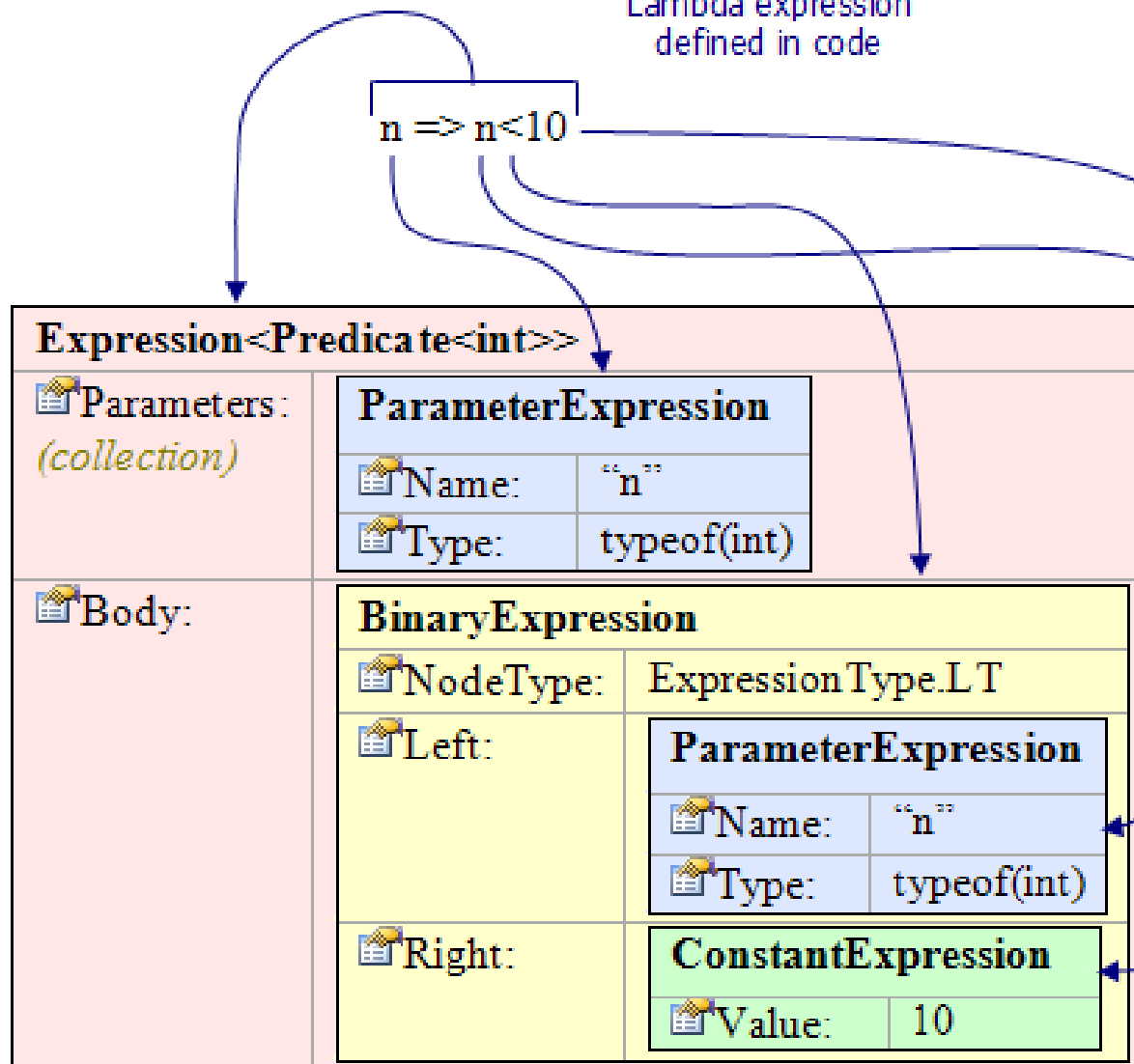
Optional **into**
continuation

Expression tree – code as data

Expression<Predicate<int>> expression = $n \Rightarrow n < 10$;


Lambda expression
defined in code

Expression tree
representation



Pozor na stejné proměnné!

```
Expression<Predicate<int>> expression =  
    Expression.Lambda<Predicate<int>>  
        (Expression.LessThan(  
            Expression.Parameter(typeof(int), "n"),  
            Expression.Constant(10)  
        ),  
        Expression.Parameter(typeof(int), "n"));
```



Různé
proměnné
pro C#

Predicate<int> predicate=expression.Compile();

Error: "Lambda Parameter not in scope"

```
ParameterExpression parn =  
    Expression.Parameter(typeof(int), "n");  
Expression<Predicate<int>> expression2 =  
    Expression.Lambda<Predicate<int>>  
        (Expression.LessThan(parn, Expression.Constant(10)),  
        parn);  
  
Predicate<int> predicate=expression.Compile();
```

```
public delegate bool Predicate<T>(T item);
```



```
Predicate<Customer> test = c => c.State == "WA";
```

```
Predicate<Customer> test = new Predicate<Customer>(XXX);
```

```
private static bool XXX(Customer c) {  
    return c.State == "WA";  
}
```

```
public delegate bool Predicate<T>(T item);
```



```
Expression<Predicate<Customer>> test = c => c.State == "WA";
```

```
ParameterExpression c = Expression.Parameter(typeof(Customer), "c");  
Expression expr =  
    Expression.Equal(  
        Expression.Property(c, typeof(Customer).GetProperty("State")),  
        Expression.Constant("WA")  
    );  
Expression<Predicate<Customer>> test =  
    Expression.Lambda<Predicate<Customer>>(expr, c);
```

Pro programové labužníky: LISP v C#

```
// Line 1: (define f (lambda (a) (+ a 1)))
```

```
Func<int, int> f = a => a + 1;
```

```
// Line 2: (define e (quote (lambda (a) (+ a 1))))
```

```
Expression<Func<int, int>> e = a => a + 1;
```

```
// Line 3: (define elong (list 'lambda (list 'a ) (list '+ 'a 1 )))
```

```
ParameterExpression ap = Expression.Parameter(typeof(int), "a");
```

```
Expression<Func<int, int>> elong =
```

```
    Expression.Lambda<Func<int, int>>
```

```
        (Expression.Add(ap, Expression.Constant(1)), ap);
```

```
// Line 4: (display (car (car (cddr e)))) ; prints +
```

```
Console.WriteLine(e.Body.NodeType); // prints Add
```

```
// Line 5: (display (elong 4)) ; runtime error
```

```
//      Console.WriteLine(elong(4)); // compile-time error 'elong' is variable
```

```
// Line 6: (define f2 (eval elong (scheme-report-environment 5)))
```

```
Func<int, int> f2 = elong.Compile();
```

```
// Line 7: (display (f2 4)) ; prints 5
```

```
Console.WriteLine(f2(4)); // prints 5
```

Quicksort

head *tail*

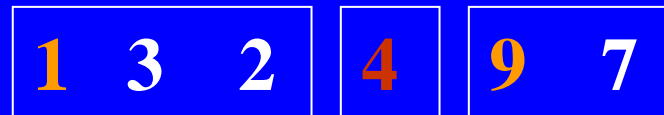
Krok 0



Krok 1

$n < 4$

$n > 4$



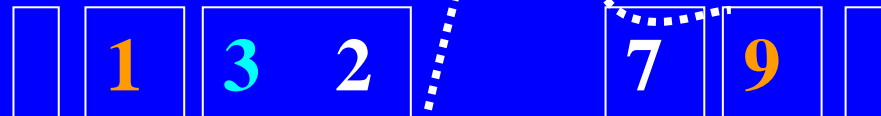
Krok 2

$n < 1$

$n > 1$

$n < 9$

$n > 9$



Krok 3

$n < 3$

$n > 3$



QuickSort v C# programu

```
static int[] QuickSort(int[] list, int ixend)
{
    if (list.Length == 0) return list;
    if (ixend == 0) return list;
    int pivot = list[0]; int ixlow = 0; int ixhigh = 0;
    int[] lt = new int[list.Length];
    int[] gteq = new int[list.Length];
    for (int i = 1; i < ixend; i++)
    {
        if (list[i] < pivot) { lt[ixlow++] = list[i]; }
        else { gteq[ixhigh++] = list[i]; }
    }
    lt = QuickSort(lt, ixlow);
    lt[ixlow++] = pivot;
    gteq = QuickSort(gteq, ixhigh);
    for (int i = 0; ixlow < lt.Length; i++) lt[ixlow++] = gteq[i];
    return lt;
}
```


Quicksort v C# Lambda výrazech



C# 3.0

parameterized type of functions

```
Func<intlist, intlist> Sort =
```

```
Ts.Case(
```

```
() => Ts,
```

```
(head, tail) =>
```

```
    QuickSort(tail.Where(t => t < head))
```

```
    .Concat(new[] { head })
```

```
    .Concat(QuickSort(tail.Where(t => t >= head)))
```

```
);
```

higher-order function

lambda expression

append

type inference

recursion

filter



Quicksort v F#

```
let rec qsort = function
```

```
    | [] -> []
```

```
    | x::xs' -> let (l, r) = List.partition ((>) x) xs'
```

```
                List.concat [(qsort l);[x];(qsort r)]
```

```
let pole = [1; 5; 8; 2; 3; 6];
```

```
let pole2 = qsort pole;
```

```
printfn "%A" pole2;
```

Mini-Úvod F#...

- **.NET jazyk**
- **kombinace Lisp, ML, Scheme, Haskell, v kontextu .NET**
- **Funkcionální, matematicky orientovaný**
- **Podobný ML (Meta-Language) jazyku Robina Milnera**



F# na jednom snímku

Automatické typy

- `let data = (1,2,3)`
- `let sqr x = x * x`
- `let f (x,y,z) = (sqr x, sqr y, sqr z)`
- `let sx,sy,sz = f (10,20,30)`
- `print "hello world"; 1+2`

`val data: int * int * int`

`val sqr: int -> int`

Nepovinné závorky

pattern
matching

`let show x y z =`

- `printf "x = %d y = %d y = %d \n" x y z;`
- `let sqrs= f (x,y,z) in`
- `print "Hello world\n";`
- `sqrs`
- `let (|>) x f = f x`

Řetězení operací

Lokální operace

pipelining operator

Zjednodušení života



Automatické typy -
jednoduchost skriptu
spojená s typovou
bezpečností

Statická hodnota

Statická funkce

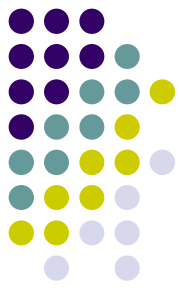
Lokální hodnota

lokální funkce

```
let data = (1,2,3)
```

```
let f(a,b,c) =  
  let sum = a + b + c in  
  let g(x) = sum + x*x in  
  g(a), g(b), g(c)
```

Typový systém



- Typová bezpečnost (stejně jako C#)
 - Většinou ale není potřeba typ psát
 - Odvozuje typy z kontextu (type inference)

```
// Hodnota celočíselného typu (int)
let n = 42
```

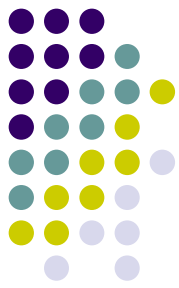
```
// Hodnota typu řetězec (string)
let str = "Hello world!"
```

```
// Funkce (int -> int)
let add10(n) =
```

- $n + 10$
• Využívá typové parametry (generics v .NET 2.0)

```
// Funkce - vrací parametr ('a -> 'a)
```

```
let id(sth) = sth
```



Tuple

- n-tuple argument

let args = (1,2,3)

Tuples
as Data

Multiple Args

let f (a,b,c) = (a+b , b+c, a+b)

Multiple
Returns

let x,y,z = f(f(f args))

printf "res = %d,%d,%d" x y z
res = 17,16,15

Funkce se podobají lambda-výrazům



```
(fun x -> x + 1)
```

```
let f x y = x * y
```

```
let g x y = x + y
```

```
let p = (f,g)
```

One simple
mechanism,
many
sophisticated
uses

predicate = 'a -> bool

send = 'a -> unit

threadStart = unit -> unit

comparer = 'a -> 'a -> int

hasher = 'a -> int

equality = 'a -> 'a -> bool

Typový systém



- Funkce je typ jako každý jiný
 - Lze ji předávat jako parametr a vracet jako výsledek
 - Lze ji vytvářet kdekoliv v kódu

```
// Funkce (int -> int)
let add10_a(n) = n + 10
```

```
// Ekvivalentní funkce (int -> int)
let add10_b = fun n -> n + 10
```

```
// Funkce bere jako parametr funkci
// Typ: (int -> int -> int) -> int
let volani(func) = 1 + func 2 3
```

```
// Předání funkce jako parametru
volani(fun a b -> a + b)
```



Možné mutování typů

- Objekty jako javascript s možným mutováním typů...

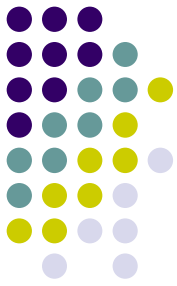
```
type person =  
  { Name : string;  
    mutable Birth : DateTime; }
```

Mutability explicit

Mutation

```
let me = {Name = "damien";  
          Birth= new DateTime(1969, 9, 3)}  
  
me.Birth <- new DateTime(1969, 9,
```

Used well, this
isolates and
controls complexity



další informace

- <http://research.microsoft.com/fsharp>
<http://blogs.msdn.com/dsyme>