

<bigwig>

A Programming Language  
for Developing  
Interactive Web Services

Claus Brabrand

**BRICS**, University of Aarhus, Denmark

# Plan

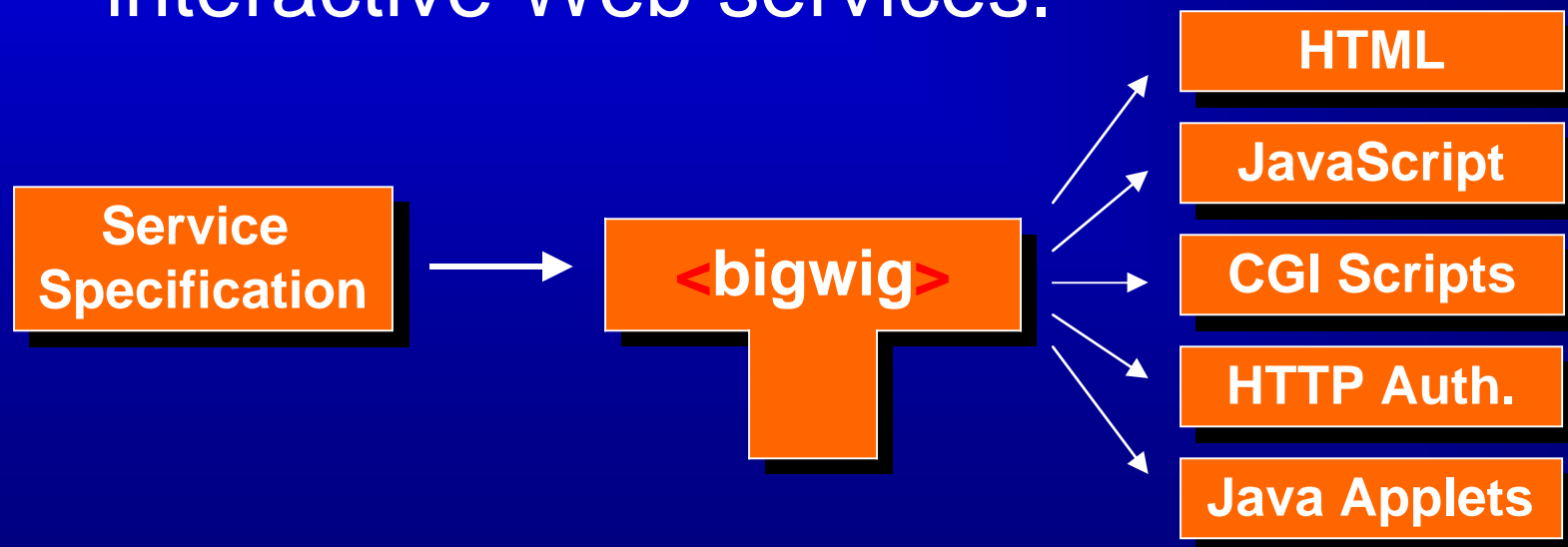
- Introduction
- Runtime Model
- Dynamic Documents
- PowerForms
- Conclusion

# Plan

- Introduction
- Runtime Model
- Dynamic Documents
- PowerForms
- Conclusion

# What is <bigwig>?

- A domain-specific high-level programming language for developing interactive Web services.



# A collection of DSLs

- C-like skeleton language with
  - Runtime system
  - Concurrency control
  - Database
  - Dynamic documents: *DynDoc*
  - Input validation language: *PowerForms*
  - Security
  - Cryptographic security
  - Syntactic-level macros

# A collection of DSLs

- C-like skeleton language *with*
  - *Runtime system* (briefly)
  - *Concurrency control* (briefly)
  - Database
  - *Dynamic documents: DynDoc*
  - *Input validation language: PowerForms*
  - Security
  - Cryptographic security
  - Syntactic-level macros

# DSL vs. GPL

- DSL ::= Domain Specific Language
- GPL ::= General Purpose Language
- DSL?
  - Targeted for specific problem domain
  - Abstraction level match problem domain
- Examples: Lex/Yacc, LaTeX

# DSL vs. GPL

- DSL ::= Domain Specific Language
- GPL ::= General Purpose Language
- DSL?
  - Targeted for specific problem domain
  - Abstraction level match problem domain
- Examples: Lex/Yacc, LaTeX, <bigwig>

# DSL Advantages (vs. GPL + library)

- Syntax
  - Design (and restrict) expressibility
  - Syntax conveys nature of constructs
- Analysis
  - Analyze domain specific aspects
- Implementation
  - Efficient implementation
    - Rely on syntax restrictions and analysis information

# Goals

- Lower development time (= cost):
  - Targeted at Web services
  - Low-level → high-level
- Increase functionality:
  - Compiler does “the dirty work”
- Increase reliability:
  - Catch errors during development
    - Runtime errors → Compile-time errors

# Assumptions

- “Rules of engagement”:
  - Lowest common denominator
    - Any browser/Web server combination
  - Only include basic primitives
    - Syntactic macro language does the rest

# Target Audience

- Programmers!
  - No expert Web knowledge required
  - No multiple choice questionnaires

“Reduce Web service development  
to a *standard* programming task.”

# Core Language Features

- C-like to minimize syntactic burdens
- Not C-like features:
  - Garbage collection
  - Relations, vectors, tuples
  - Strong type checking

# People

- 1x Michael I. Schwartzbach
- 1x Post Doc.
- 5x Ph.D. students
- 2x Programmers
- 2x Testers
- 1x External contributor

# Plan

- Introduction
- Runtime Model
- Dynamic Documents
- PowerForms
- Conclusion

# Plan

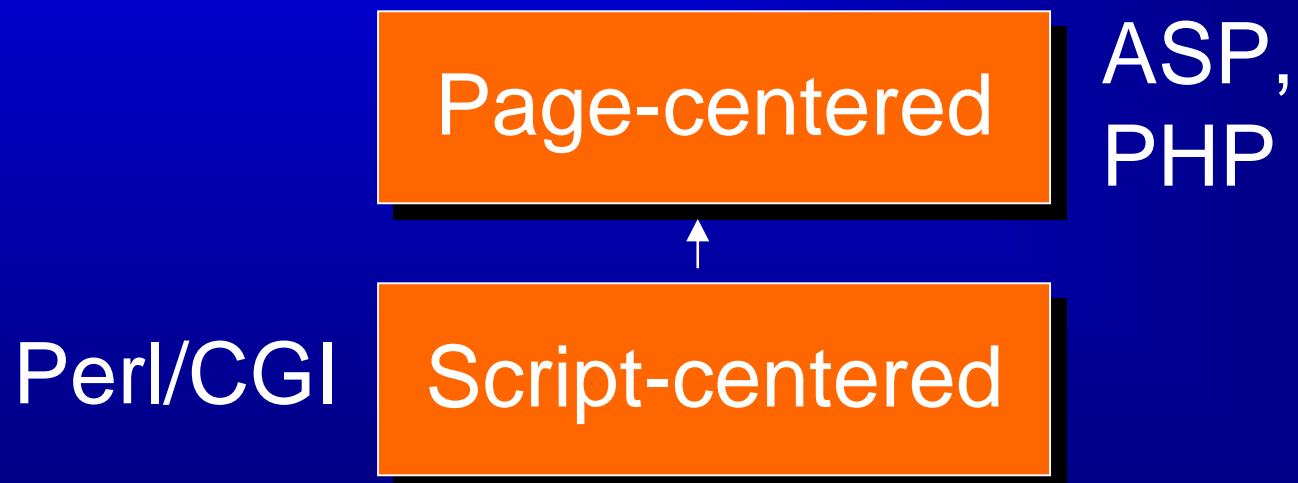
- Introduction
- Runtime Model
- Dynamic Documents
- PowerForms
- Conclusion

# 3 Approaches

Perl/CGI

Script-centered

# 3 Approaches



# 3 Approaches

Mawl,  
<bigwig>

Session-centered



Page-centered

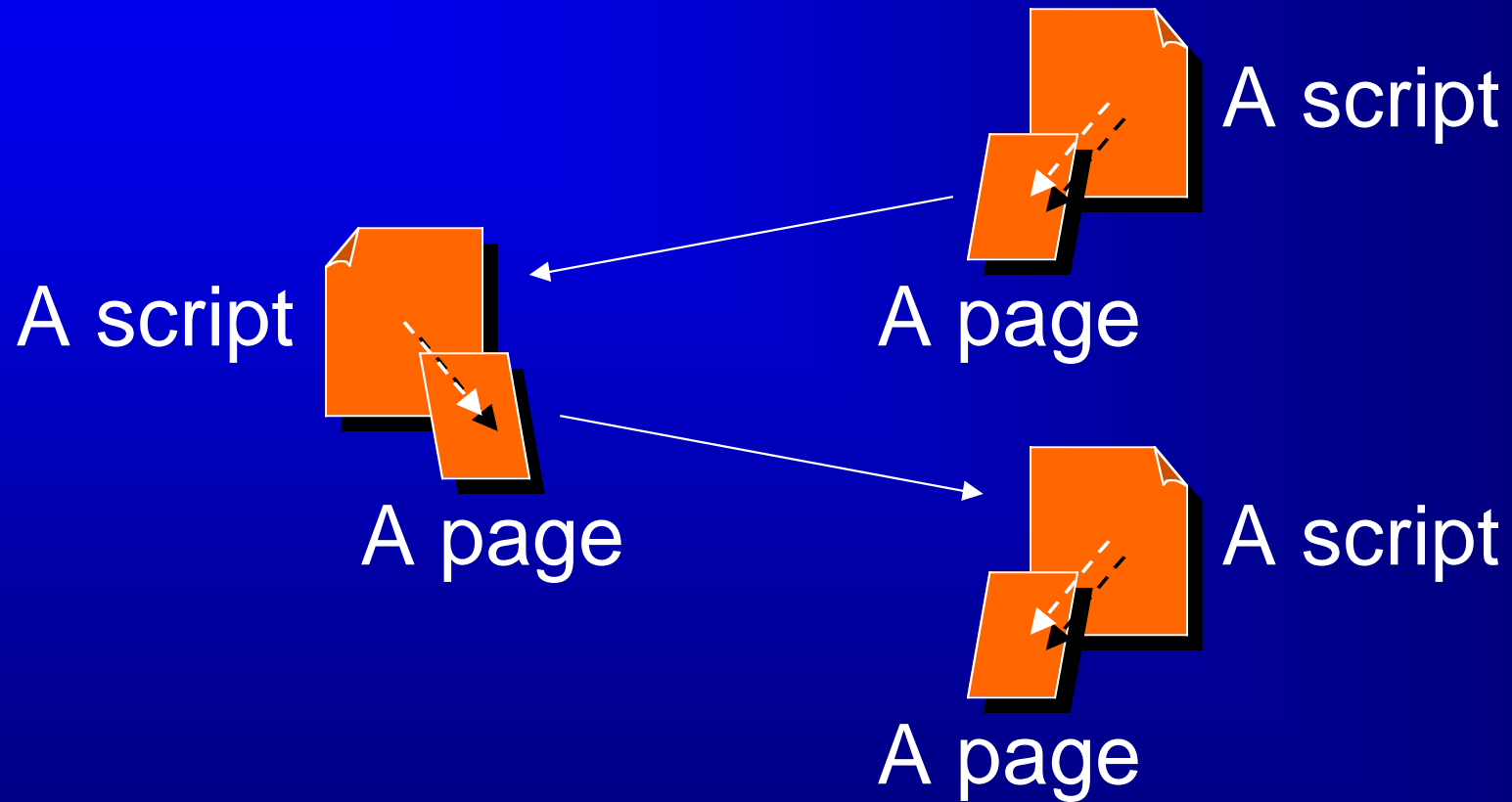
ASP,  
PHP



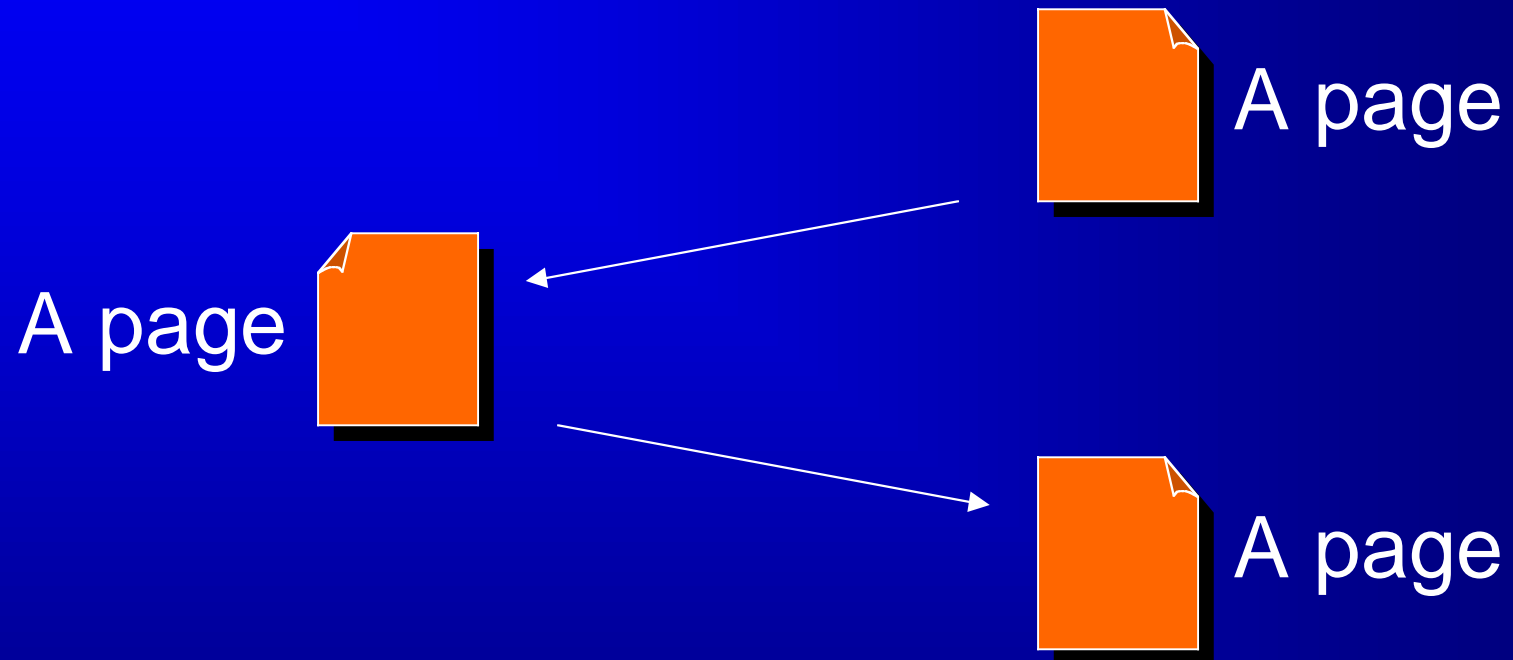
Perl/CGI

Script-centered

# Script-Centered



# Page-Centered



# Page-Centered

“Service code embedded in tags and interpreted by specialized Web server”

- Increased level of abstraction
- Easy to add dynamics to static pages
- Scalability

# Script- vs. Page-Centered

- As the service complexity increases:
  - Page-centered → Script-centered
- Script-centered:
  - default programming, escape printing.
- Page-centered:
  - default printing, escape programming.

# (Fundamental) Problems

- Interpretation-based:
  - Typically no static checks
  - (Efficiency)
- Not domain specific:
  - Cannot check Web related issues
- Implicit control-flow:
  - A service = A collection of scripts/pages!

# Language Requirements

- Compilation-based:
  - Static checks
  - (Efficiency)
- Domain specific:
  - Check Web related issues
- Explicit control-flow:
  - A clear service specification

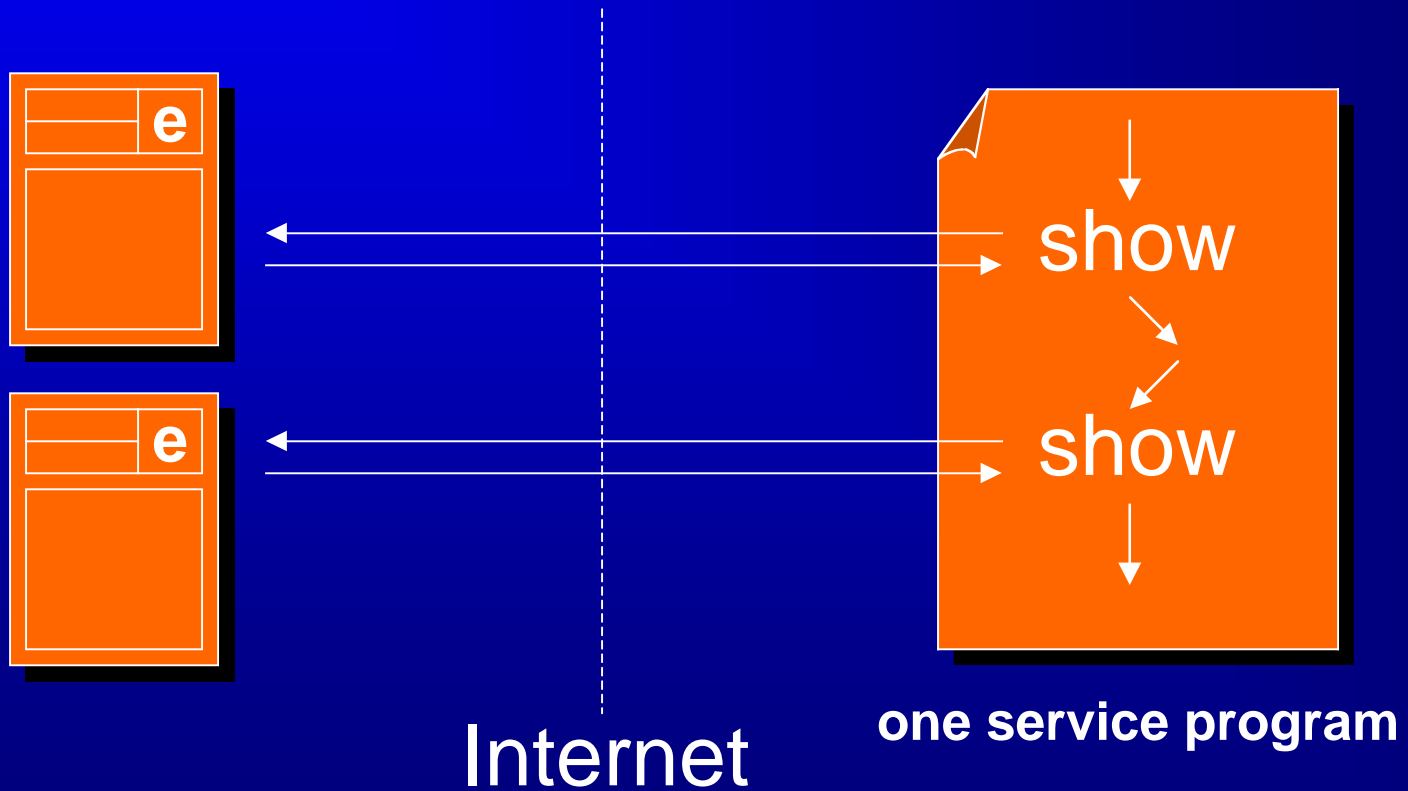
# Language Requirements

- Compilation-based:
  - Static checks
  - (Efficiency)
- Domain specific:
  - Check Web related issues
- Explicit control-flow:
  - A clear service specification

# Session-Centered

*client:*

*server:*



# Hello World

```
service {  
    session Hello() {  
        html D = <html>Hello World!</html>;  
        show D;  
    }  
}
```

# Hello World

```
service {  
  session Hello() {  
    show <html>Hello World!</html>;  
  }  
}
```

# A Page Counter

```
service {  
  session Access() {  
    shared int counter;  
    string name;  
    show EnterName receive [name=name];  
    counter = counter + 1;  
    show AccessDoc <[counter = counter];  
  }  
}
```

# A Page Counter

```
:  
if (counter == 100) {  
    counter = 0;  
    show Congratulations <[name = name];  
} else {  
    counter = counter + 1;  
    show AccessDoc <[counter = counter];  
}  
:
```

# CGI Shortcomings

- Stateless protocol
  - Session model requires state
- No bookmarking
  - CGI URL bookmarked, not HTML URL
- Back-button problem
  - “Step-back-in-time” does not make sense
- Long response times
  - Clients get impatient

# Our Solution

- Runtime System (based on CGI)
  - “Any browser/Web server combination”
- Problems:
  - Stateless protocol
  - No bookmarking
  - Back-button problem
  - Long response times
- Solutions:

# Our Solution

- Runtime System (based on CGI)
  - “Any browser/Web server combination”

- Problems:

- Stateless protocol
- No bookmarking
- Back-button problem
- Long response times

- Solutions:

*connector process*

# Our Solution

- Runtime System (based on CGI)
  - “Any browser/Web server combination”

- Problems:
  - Stateless protocol
  - No bookmarking
  - Back-button problem
  - Long response times

Solutions:  
*connector process*  
*use html reply file*

# Our Solution

- Runtime System (based on CGI)
  - “Any browser/Web server combination”

- Problems:
  - Stateless protocol
  - No bookmarking
  - Back-button problem
  - Long response times

Solutions:  
*connector process*  
*use html reply file*  
*use html reply file*

# Our Solution

- Runtime System (based on CGI)
  - “Any browser/Web server combination”

- Problems:

- Stateless protocol
- No bookmarking
- Back-button problem
- Long response times

- Solutions:

*connector process*  
*use html reply file*  
*use html reply file*  
*+ refresh + timeout*

# Runtime System

- Availability:
  - in <bigwig> compiler
  - as stand-alone package
- Underway...
  - Specialized runtime system:
    - CGI → Specialized Web server
      - efficiency, scalability

# Concurrency Control

- Problem:
  - Parallel service processes.
    - Access shared resources.
    - Require synchronization.

- Example:

```
counter = counter + 1;
```

# Counter Example

:

```
counter = counter + 1;
```

:

# Counter Example

...add *checkpoints*

```
:  
wait A;  
counter = counter + 1;  
wait B;  
:
```

# Counter Example

...and *constraints* (“M2L-Str” logic)

$$\forall t, t'': A(t) \wedge A(t'') \wedge t < t'' \Rightarrow \exists t': t < t' < t'' \wedge B(t')$$

:

wait A;

counter = counter + 1;

wait B;

:

# Counter Example

*“Ensure that service obeys constraints”*

$$\forall t, t'': A(t) \wedge A(t'') \wedge t < t'' \Rightarrow \exists t': t < t' < t'' \wedge B(t')$$

:

wait A;

counter = counter + 1;

wait B;

:



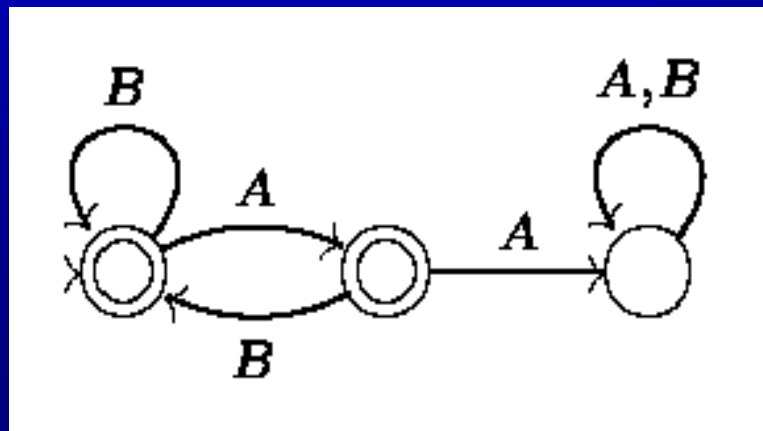
Exclusive  
access

# Compilation Process

$$\forall t, t'': A(t) \wedge A(t'') \wedge t < t'' \Rightarrow \exists t': t < t' < t'' \wedge B(t')$$

Mona

DFA:



# In practice...

```
service {  
  session Access() {  
    shared int counter;  
    :  
    counter = counter + 1;  
    :  
  }  
}
```

# Syntax Macros

```
service {  
    session Access() {  
        region shared int counter;  
        :  
        exclusive (counter) {  
            counter = counter + 1;  
        }  
    }  
}
```

# Demo Example: Mutex

label A, B;

$\forall t, t'': A(t) \wedge A(t'') \wedge t < t'' \Rightarrow \exists t': t < t' < t'' \wedge B(t')$

# Demo Example: Mutex

label A, B;

$\forall t, t'': A(t) \wedge A(t'') \wedge t < t'' \Rightarrow \exists t': t < t' < t'' \wedge B(t')$

```
session A() {  
  while (!quit) {  
    wait A;  
    show Passed_A;  
  }  
}
```

# Demo Example: Mutex

label A, B;

$\forall t, t'': A(t) \wedge A(t'') \wedge t < t'' \Rightarrow \exists t': t < t' < t'' \wedge B(t')$

```
session A() {  
  while (!quit) {  
    wait A;  
    show Passed_A;  
  }  
}
```

```
session B() {  
  while (!quit) {  
    wait B;  
    show Passed_B;  
  }  
}
```

# Plan

- Introduction
- Runtime Model
- Dynamic Documents
- PowerForms
- Conclusion

# Plan

- Introduction
- Runtime Model
- Dynamic Documents
- PowerForms
- Conclusion

# Documents

- Traditionally: `printf(...)` / `<% print(...) %>`
- Problems:
  - Only linear construction
  - Programmer/Designer tasks not separated
  - Show/Receive correspondence?
  - Legal/sensible HTML generated?

# Documents

- Traditionally: `printf(...)` / `<% print(...) %>`
- Problems:
  - Only linear construction
  - Programmer/Designer tasks not separated
  - Show/Receive correspondence?
  - Legal/sensible HTML generated?

# Our Solution: Document Templates

- HTML → HTML with named gaps

```
<html>  
  <body bgcolor=[color]>  
    <h1>Hello <[what]>!</h1>  
    <input type="text" name="name">  
  </body>  
</html>
```

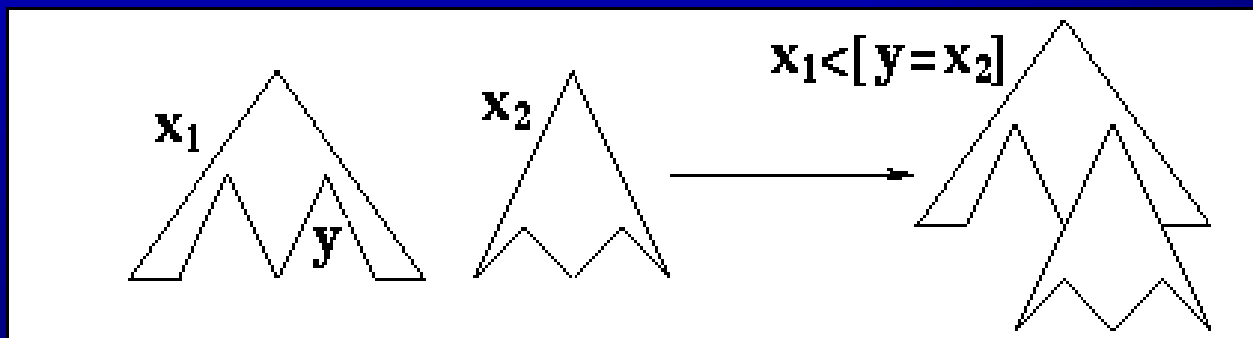
...*gaps* are plugged at runtime

# Dynamic Documents

- Domain specific type: html (with **gaps**)
  - $type ::= \underline{int} \mid \underline{float} \mid \underline{string} \mid \dots \mid \underline{html}$
- Domain specific (sub)language: *DynDoc*
  - $exp ::= \dots \mid c \mid id \mid id = exp \mid exp <[id = exp]$
  - $stm ::= \dots \mid \underline{show} \ exp; \mid$   
 $\underline{show} \ exp \underline{receive} \ [ \ id = id \ ];$

# Plug

- Syntax:
  - $exp ::= exp <[id = exp]$
- Semantics: (no side-effects!)



# Hello World (revisited)

```
session Hello() {  
  html H = <html>Hello <[what]>!</html>;  
  html W = <html><b>World</b></html>;  
  html D;  
  
  D = H <[what = W];  
  show D;  
}
```

## Rec. Example: Genealogy

```
html GenDoc = <html><ul><li>...</ul></html>;
```

```
html genTree(int n, string s) {  
  if (n == 0) return <html></html>;  
  else return GenDoc <[mother = s + "mother",  
    mothers_tree = genTree(n-1, "mother's"),  
    father = s + "father",  
    father_tree = genTree(n-1, "father's")];  
}
```

# Highly Efficient Runtime Representation

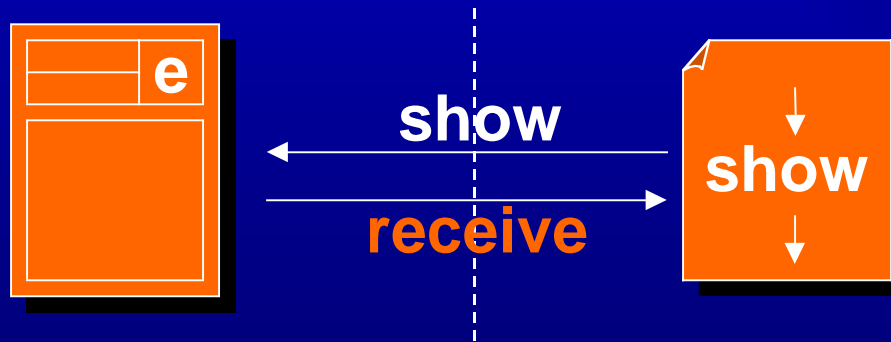
- Time:
  - Plug:  $O(1)$  (constant time).
  - Show:  $O(|D|)$  (linear time).
- Space (maximum sharing):
  - Doc.:  $O(\#plugs)$  (not  $O(|D|)$  !)

# Show / Show-Receive

- Syntax:

–  $stm ::= \underline{\text{show}} \text{ exp}; \mid$   
 $\underline{\text{show}} \text{ exp} \underline{\text{receive}} [ id = id ];$

- Semantics:



# Example: EnterData

```
string name, email;
```

```
html Input = <html>
```

```
    name: <input name="name">
```

```
    email: <input name="email">
```

```
</html>;
```

```
show Input receive [name = name,  
                    email = email];
```

# Documents

- Problems:
  - Only linear construction
  - Programmer/Designer tasks not separated
  - Show/Receive correspondence?
  - Legal/sensible HTML generated?

# Documents

- Problems:
  - Only linear construction ✓
  - Programmer/Designer tasks not separated
  - Show/Receive correspondence?
  - Legal/sensible HTML generated?

# Documents

- Problems:
  - Only linear construction ✓
  - Programmer/Designer tasks not separated ✓
  - Show/Receive correspondence?
  - Legal/sensible HTML generated?

# Documents

- Problems:
  - Only linear construction ✓
  - Programmer/Designer tasks not separated ✓
  - Show/Receive correspondence?
  - Legal/sensible HTML generated?

# Static Guarantees?

- Documents well-formed?
  - **Field** consistency?
- Plug operation:
  - **Gap** present?
  - Consistent **field** union?
- Show/Receive correspondence:
  - All **fields** received?
  - **Field** receive types match?

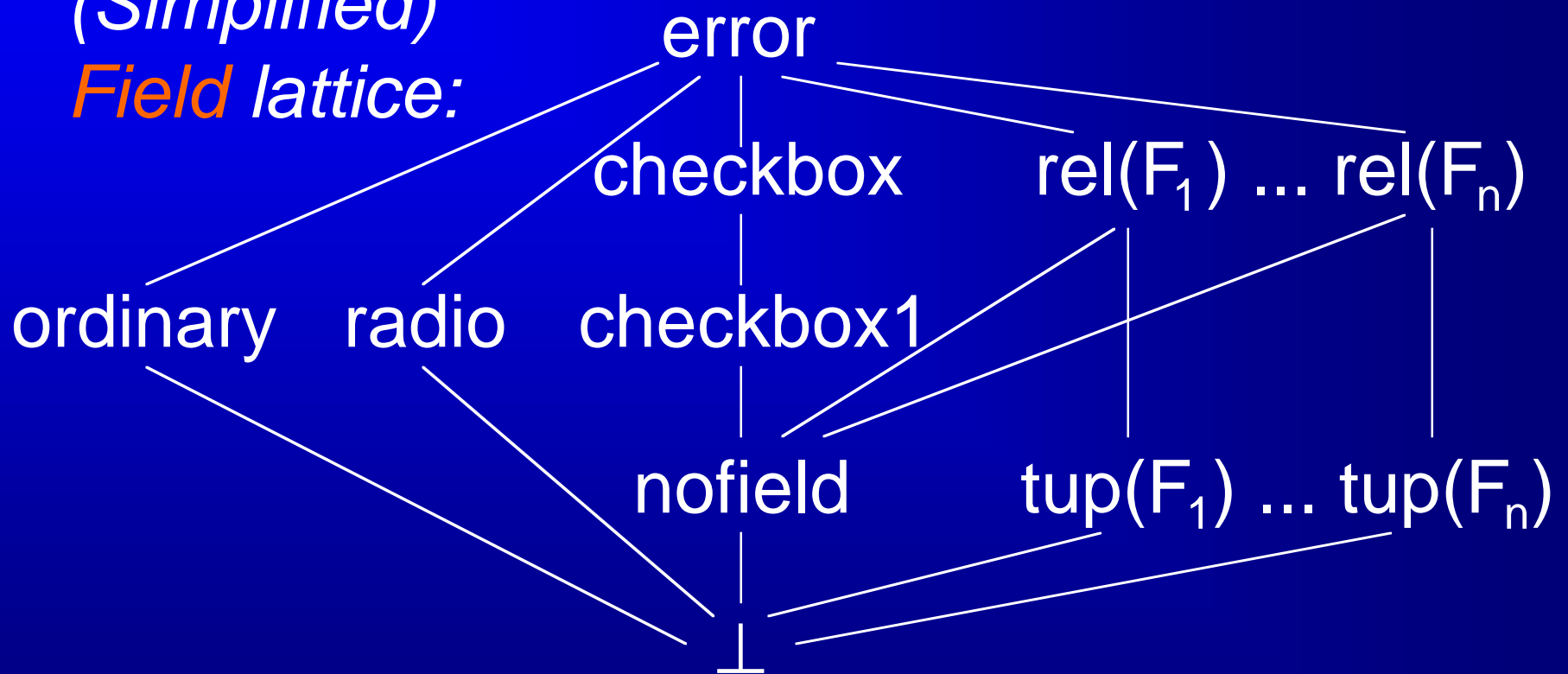
# Answer:

## Domain Specific Analysis

- Interprocedural data-flow analysis:
  - Infer exact types of all documents in program: (gaps, fields).
  - Check:
    - documents well-formed
    - plug operations
    - show/receive correspondence

# Highly Domain Specific

(Simplified)  
*Field lattice:*



# Documents

- Problems:
  - Only linear construction ✓
  - Programmer/Designer tasks not separated ✓
  - Show/Receive correspondence?
  - Legal/sensible HTML generated?

# Documents

- Problems:
  - Only linear construction ✓
  - Programmer/Designer tasks not separated ✓
  - Show/Receive correspondence? ✓
  - Legal/sensible HTML generated?

# Documents

- Problems:
  - Only linear construction ✓
  - Programmer/Designer tasks not separated ✓
  - Show/Receive correspondence? ✓
  - Legal/sensible HTML generated? (✓ )

# Future Plan

- Analyze generated HTML documents
  - with respect to:
    - HTML 3.2 / 4.01 / ...
    - DTD / DSD / ...
- Ensure that only “legal” documents are generated

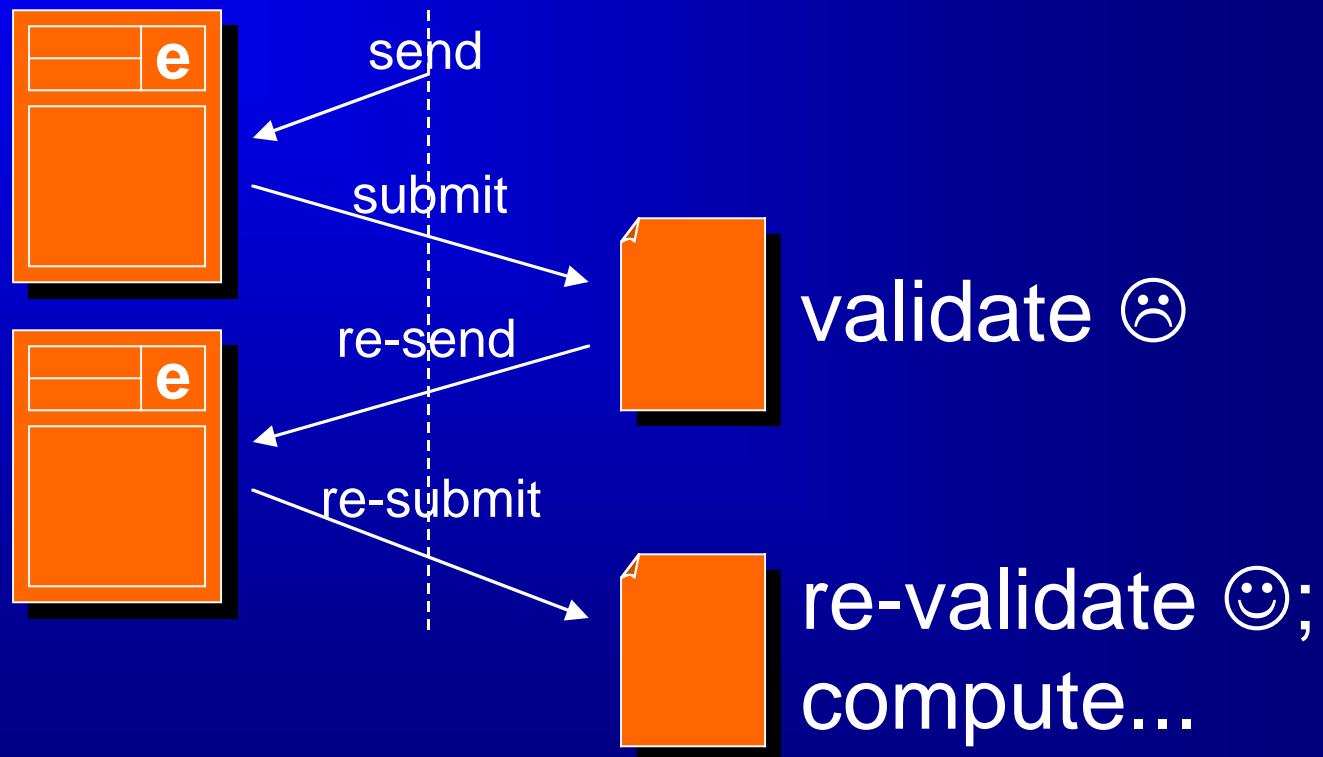
# Plan

- Introduction
- Runtime Model
- Dynamic Documents
- PowerForms
- Conclusion

# Plan

- Introduction
- Runtime Model
- Dynamic Documents
- PowerForms
- Conclusion

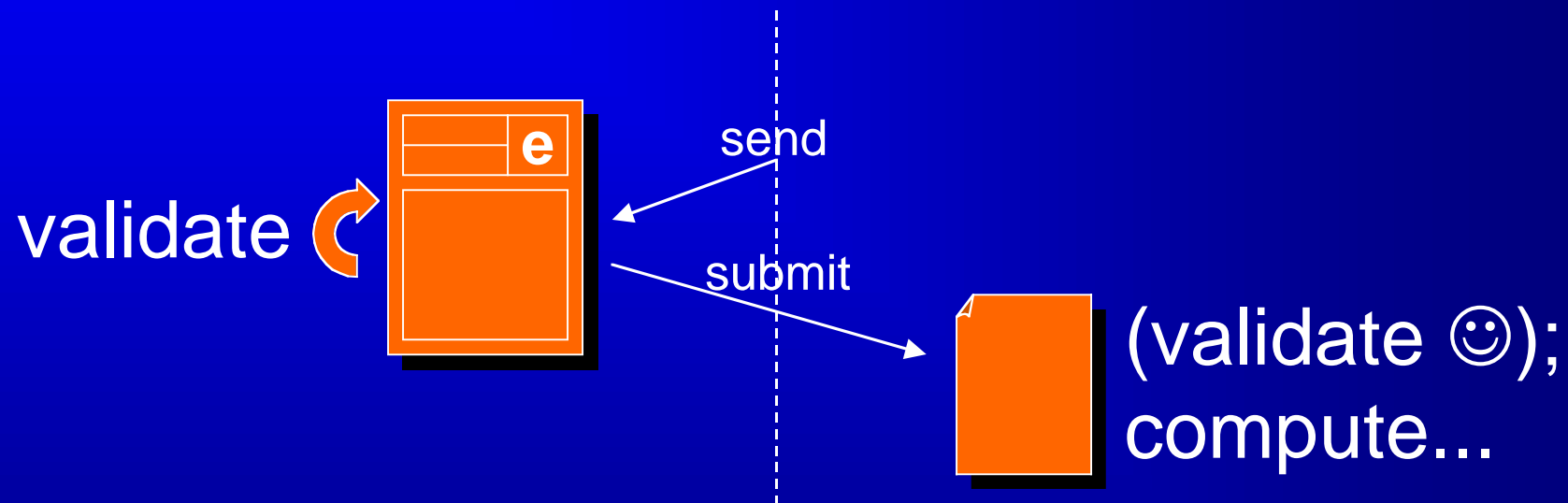
# Server-side Input Validation



# Drawbacks

- It takes time
- Excess network traffic
- Requires explicit programming
  - Affects all parties involved:
    - client
    - server
    - programmer

# Client-side Input Validation



# Drawbacks

- It takes time
- Excess network traffic
- Requires explicit programming

# Drawbacks

- It takes time  $\surd$
- Excess network traffic
- Requires explicit programming

# Drawbacks

- It takes time ✓
- Excess network traffic ✓
- Requires explicit programming

# Drawbacks

- It takes time ✓
- Excess network traffic ✓
- Requires explicit programming:
  - re-showing of pages
  - actual validation

# Drawbacks

- It takes time ✓
- Excess network traffic ✓
- Requires explicit programming:
  - re-showing of pages ✓
  - actual validation

# Drawbacks

- It takes time ✓
- Excess network traffic ✓
- Requires explicit programming:
  - re-showing of pages ✓
  - actual validation ☹

# Drawbacks

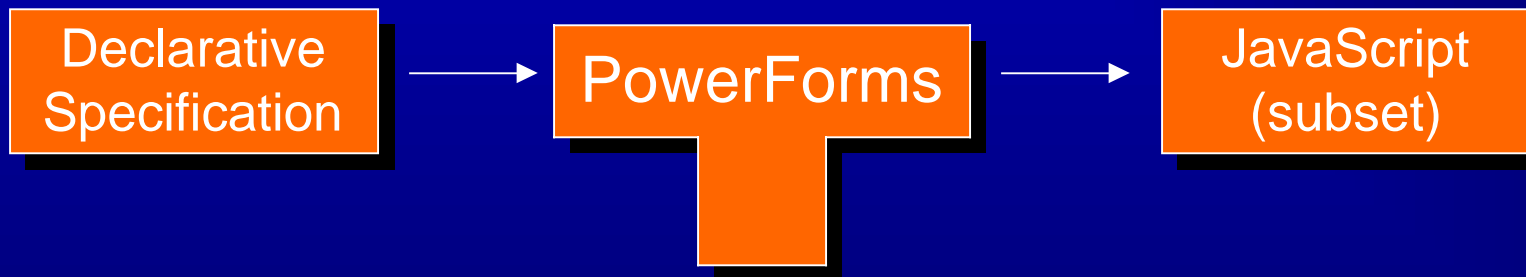
- It takes time ✓
  - Excess network traffic ✓
  - Requires explicit programming:
    - re-showing of pages ✓
    - actual validation ☹
- 
- Client, server: ☺
  - Programmer: ☹

# Obvious Language: JavaScript

- Why avoid JavaScript?:
  - GPL for very specific task
  - Operational form
  - Diverging browser implementations:
    - Explorer vs. Netscape

# Our Solution: *PowerForms*

- Domain specific language:
  - targeted uniquely for input validation
- Declarative nature (regexps):
  - abstracts away operational details



# Syntax

- $decl ::= \underline{\text{format}} \textit{id} = \textit{regexp} ;$
- $regexp ::= \textit{id} \mid \text{stringconst}$ 
  - |  $\text{union} ( \textit{regexp}^* )$
  - |  $\text{concat} ( \textit{regexp}^* )$
  - |  $\text{star} ( \textit{regexp} ) \mid \dots$

$\langle \text{input type}=\text{"text"} \text{ name}=\text{"N"} \text{ format}=\text{"F"} \rangle$

# Example: Email Format

```
format Alpha = union(range('a','z'),range('A','Z'));  
format Word = ...;  
format Email = concat(Word,"@",Word,  
                        star(concat(".",Word)));
```

# Example: EnterData (revisited)

html Input = <html>

name: <input name="name">

email: <input name="email">

</html>;

show Input receive [name=name,email=email];

# Example: EnterData (revisited)

```
format Email = ...;
```

```
html Input = <html>  
  name: <input name="name">  
  email: <input name="email">  
</html>;
```

```
show Input receive [name=name,email=email];
```

# Example: EnterData (revisited)

```
format Email = ...;
```

```
html Input = <html>
```

```
  name: <input name="name">
```

```
  email: <input name="email" format="Email">
```

```
</html>;
```

```
show Input receive [name=name,email=email];
```

# Example: EnterData (revisited)

```
format Email = ...;
```

```
html Input = <html>
```

```
  name: <input name="name">
```

```
  email: <input name="email" format="Email">
```

```
</html>;
```

```
show Input receive [name=name,email=email];
```

# Field Interdependency

Have you attended past WWW conferences?  Yes  No

If Yes, how did WWW8 compare?  Better  Same  Worse

...usually only handled on server-side

# PowerForms (also)

- Extend (declarative specification):
  - formats depend on values of other fields
  - Update accordingly
    - text / password: status icons *updated*
    - radio / checkbox: illegal options *deselected*
    - select: illegal options *filtered* (and *deselected*)
  - Note: Fixed-point process ( $\leq$  #fields)

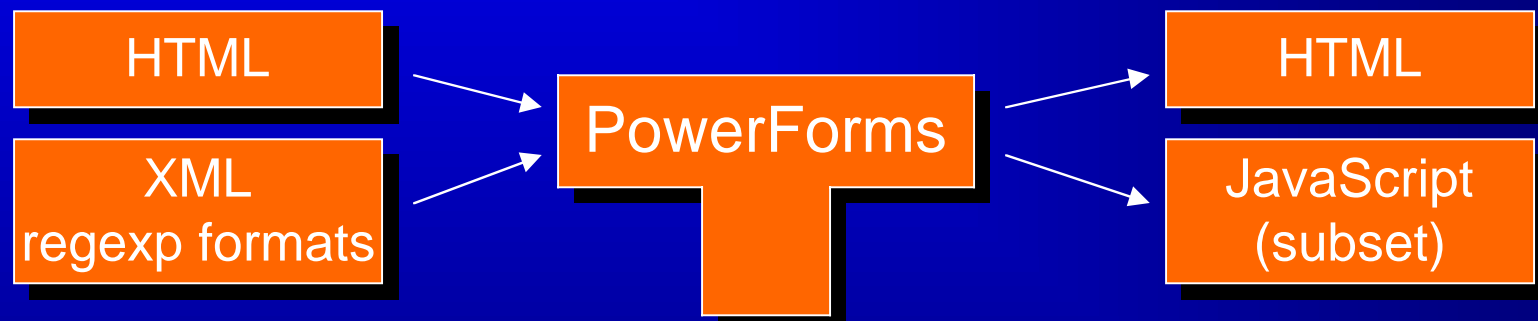
# Example Demos

...speak for themselves...

- “Spouse”
  - Basic interdependency
- “Vowels and Consonants”
  - Select filtering
- “NYC Office”
  - Complex interdependency

# PowerForms

...also as *Stand-alone Tool*:



# Plan

- Introduction
- Runtime Model
- Dynamic Documents
- PowerForms
- Conclusion

# Plan

- Introduction
- Runtime Model
- Dynamic Documents
- PowerForms
- Conclusion

# Availability?

- `<bigwig>`: 1.5 MB C source
  - for UNIX/Linux
  - Also as stand-alone packages:
    - Runtime System
    - PowerForms
- License?
  - Gnu Public License

# <bigwig> Publications

- <bigwig>
  - Runtime System *...submitted*  
WWW 8, Toronto
  - Concurrency Control *...submitted*  
FASE'98, Lisbon
  - Database *...submitted*  
(IPL'92)
  - Dynamic Documents *...submitted*  
POPL'00, Boston
  - PowerForms *...submitted*
  - Macros *...underway*
- Planned:
  - Security / Cryptographic security / Workflow

# Current Activities

- Scalable specialized runtime system.
- External database integration.
- Better dynamic documents.
- *“Next generation”* syntax macros.
- Security (information flow).
- Cryptographic protocol integration.
- Service management.
- *and more...*

# What is <bigwig>?

- Runtime System
- Concurrency Control
- Database
- Dynamic Documents
- PowerForms
- Security / Cryptographic security
- Syntactic-level Macros

<http://www.brics.dk/bigwig/>

# Adding a Safety Controller

