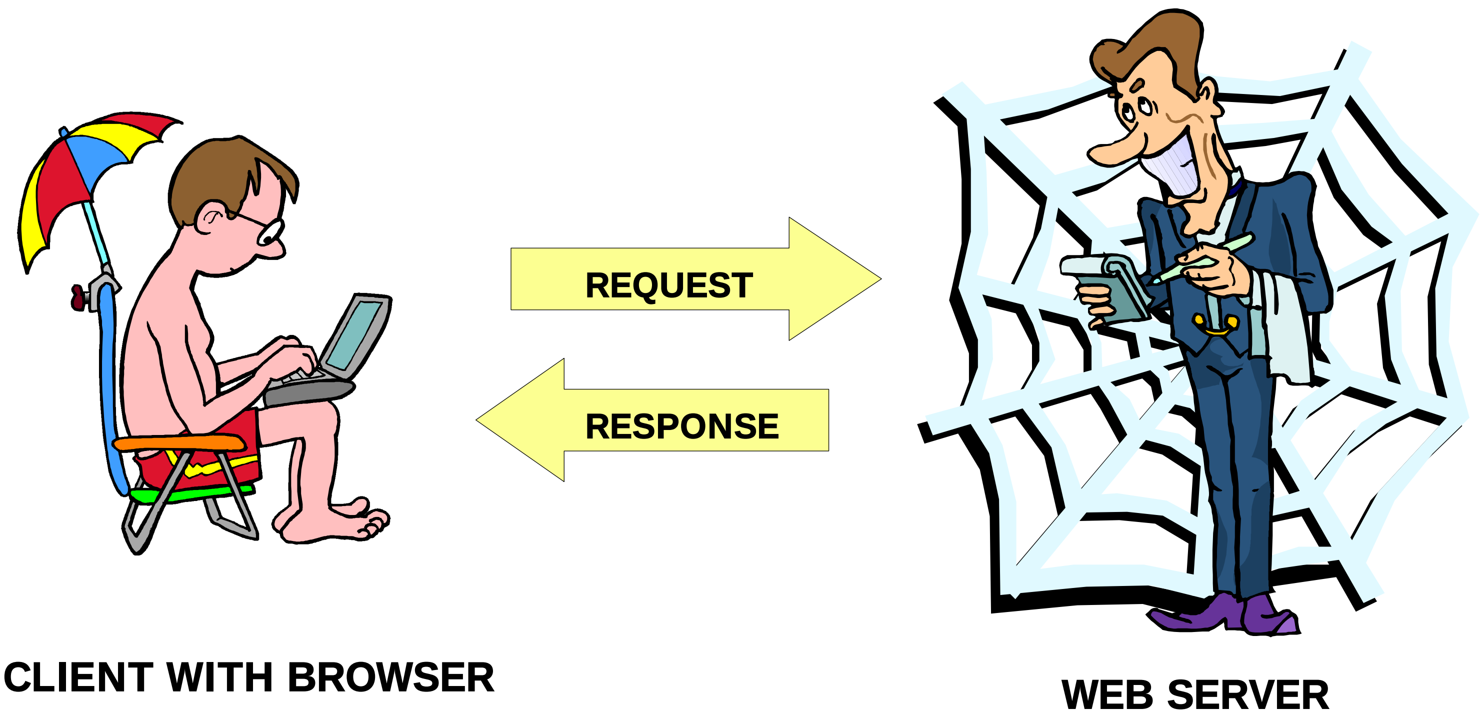
Static Analysis for Dynamic XML
- The JWIG Project

Anders Møller
Aske Simon Christensen
Michael I. Schwartzbach

University of Aarhus
BRICS

Interactive Web Services



Two central aspects of Web service development:

- session management
- dynamic construction of Web pages

Problems with Existing Technologies

CGI/Perl, Servlets, JSP, ASP, PHP, ...

Session management:

- URL rewriting / hidden form fields / cookies
- hidden control-flow!
- complicated session state management!
- impossible to statically verify correspondence between generated Web pages and received form fields!

Dynamic construction of Web pages:

- printing string fragments to output stream
- unflexible, requires linear construction!
- HTML and code is mixed together!
- impossible to statically verify that only valid HTML is generated!

The JWIG Solution

JWIG: a novel Java-based framework for
Web service development

JWIG features:

- an explicit **session** concept
- **shared state** through usual scope mechanisms
- **XML templates** as first-class values
- **static guarantees** about the behavior of running services
  - all received form data is as expected
  - all dynamically generated XML documents are guaranteed to be valid (X)HTML

An Example JWIG Program

```
import dk.brics.jwig.*;
public class ExampleService extends Service {
  int users = 0;
  synchronized int next() { return ++users; }
  XML wrapper = [[ <html>
                    <head><title>JWIG</title></head>
                    <body><[BODY]></body>
                  </html> ]];
  public class ExampleSession extends Session {
    String name;
    public void main() {
      XML ask = [[ <form>Your name? <input name="NAME"/>
                   <input type="submit"/></form> ]];
      show wrapper <[ BODY = ask ];
      name = receive NAME;
      XML goodbye = wrapper <[ BODY = [[
                     Hello <[WHO]>, you are visitor number <[COUNT]>!
                    ]] ] <[ WHO = name, COUNT = next() ];
      show goodbye;
} } }
```

Higher-Order XML Templates

XML templates:

- a built-in first-class data type
- well-formed fragments of XML containing named gaps
- constants + template/string plug operations
- allow separation of XML and code
- efficient implementation: constant time *plug*, linear time *show*

```
<html>
  <head><title>JWIG</title></head>
  <body><[BODY]></body>
</html>
```

```
<[BODY = Hello <[WHO]>, you are
         visitor number <[COUNT]>! ]
```

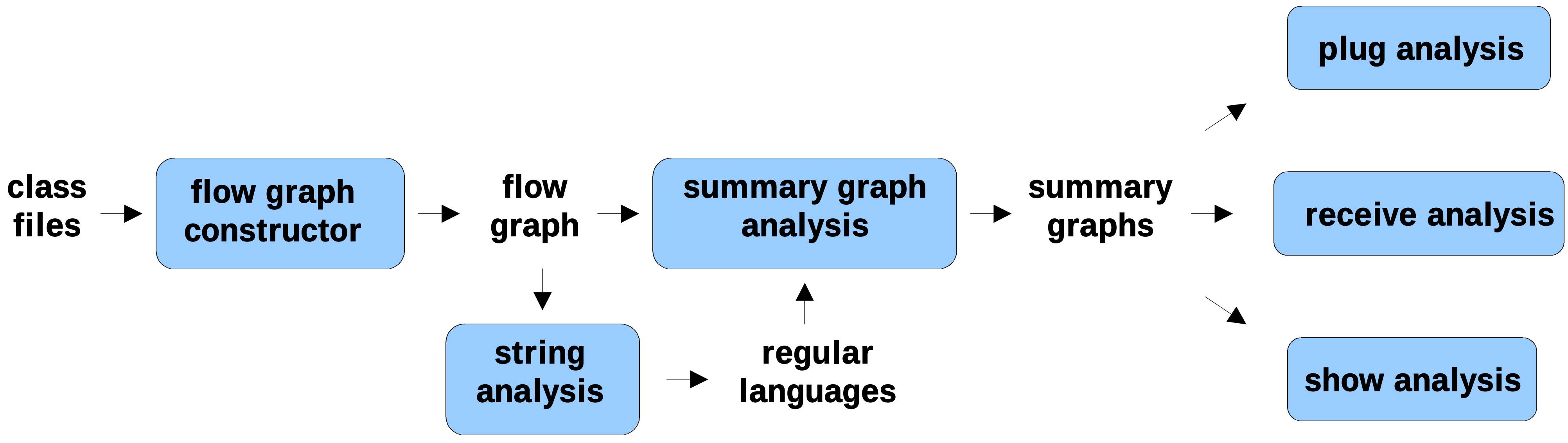
⬇

```
<html>
  <head><title>JWIG</title></head>
  <body> Hello <[WHO]>, you are
         visitor number <[COUNT]>! </body>
</html>
```

Client Interactions

- RPC-like **show** statement
- **`receive`** expression for reading form data
- "type checking":
  - argument is valid XHTML
  - result is expected set of name-value pairs

Static Guarantees using Program Analysis



Flow Graphs

A **flow graph** of a JWIG program captures

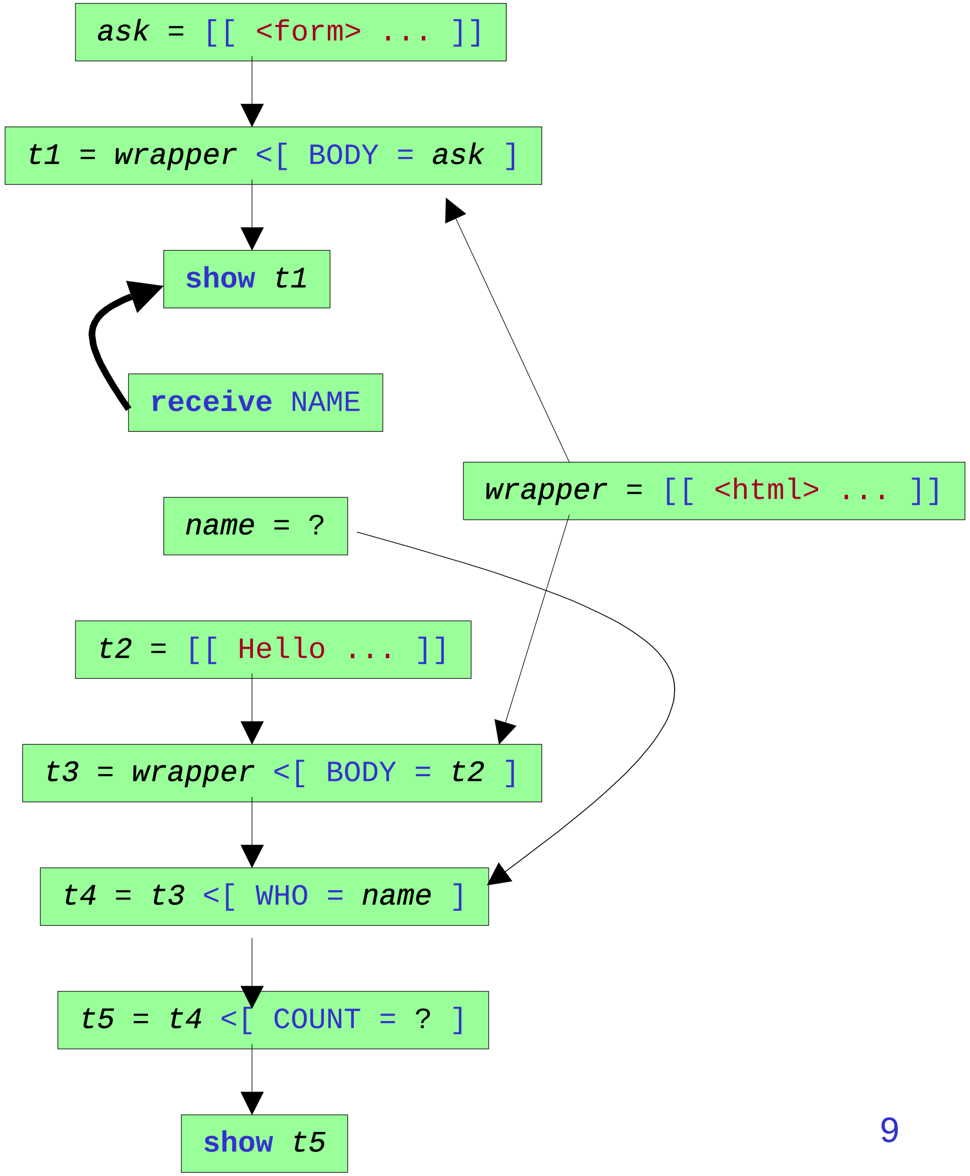
- the flow of **string** and **XML** values
- the correspondence between **show** and **receive** operations

while abstracting everything else away

node ~ abstract statements

flow edges ~ data flow

receive edges ~ show/receive relation



Construction of Flow Graphs

JWIG program $\rightarrow$ Flow graph

1. Individual methods
2. Code gaps
3. Method invocations (monovariant, using CHA)
4. Exceptions
5. Show and receive operations
6. Arrays (using weak updating and aliasing)
7. Field variables (flow-insensitive)
8. Graph simplification (reaching definitions, def-use edges, copy propagation)

Summary Graphs

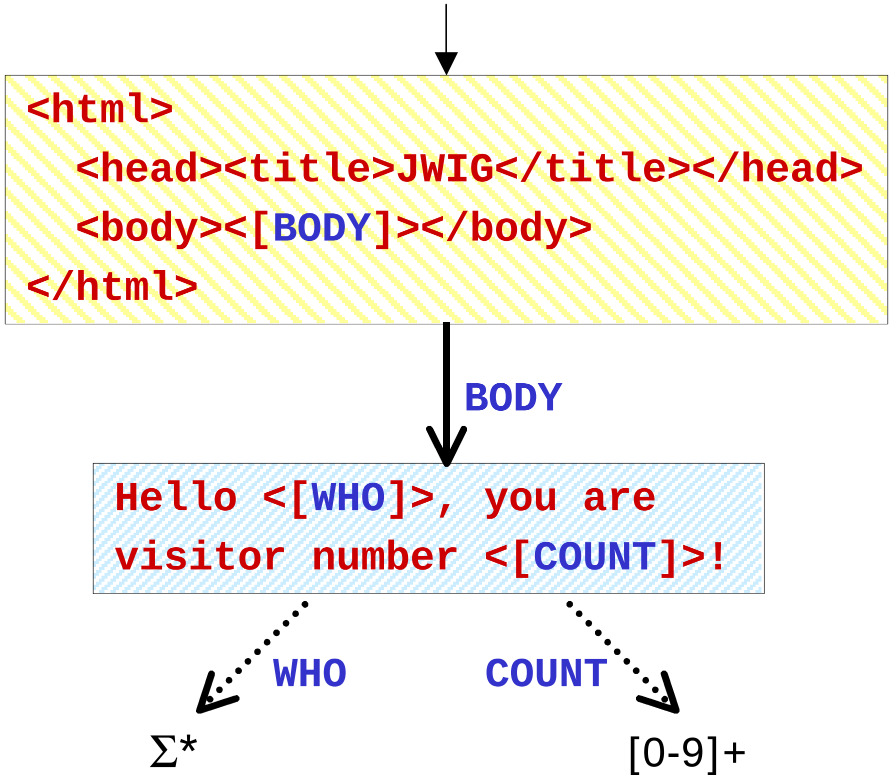
A **summary graph** is a convenient representation of a **set of XML values**

node ~ XML template constant

template edges ~ template plug operations

string edge ~ string plug operations

+ gap presence information



Summary Graphs as Mathematical Structures

$SG = (R, T, S, P)$

- $R \subseteq N$ is a set of *root nodes*
- $T \subseteq N \times G \times N$ is a set of *template edges*
- $S: N \times G \rightarrow REG$ is a *string edge* map
- $P: G \rightarrow 2^N \times \Gamma \times \Gamma$ is a *gap presence* map

$\Gamma = 2^{\{\text{OPEN},\text{CLOSED}\}}$

$L(SG) = \{\ close(d) \in XML \mid d \in unfold(SG)\ \}$

Construction of Summary Graphs

Flow graph $\rightarrow$ Summary graphs
(one for each XML variable at each program point)

- summary graphs form a **lattice**
- apply **standard data-flow analysis framework**
- relatively straightforward **transfer functions**

Catching Errors

Example: If the programmer forgets the **name** attribute:

```
...
XML ask = [[ <form>Your name? <input name="NAME"/>
             <input type="submit"/></form> ]];
...
```

then the JWIG program analyzer will find out:

```
*** Field `NAME' is never available on line 15
*** Invalid XHTML at line 14
--- element 'input': requirement not satisfied:
<or>
  <attribute name="type">
    <union>
      <string value="submit" />
      <string value="reset" />
    </union>
  </attribute>
  <attribute name="name" />
</or>
```

Analysis Performance

Benchmark	Lines	Templates	Shows	Analysis Time (sec.)
Chat	80	4	3	5.3
Guess	94	8	7	7.1
Calendar	133	6	2	7.0
Memory	167	9	6	9.7
TempMan	238	13	3	7.7
WebBoard	766	32	24	9.7
Bachelor	1078	88	14	115.6
Jaoo	3923	198	9	36.0

- **soundness** guarantees that no errors are missed
- **no false positives** encountered

Related Work

Mawl [Ball *et al.*]

- introduced **session**-based model
- **first-order** XML templates

Related Work

XDuce [Pierce *et al.*]

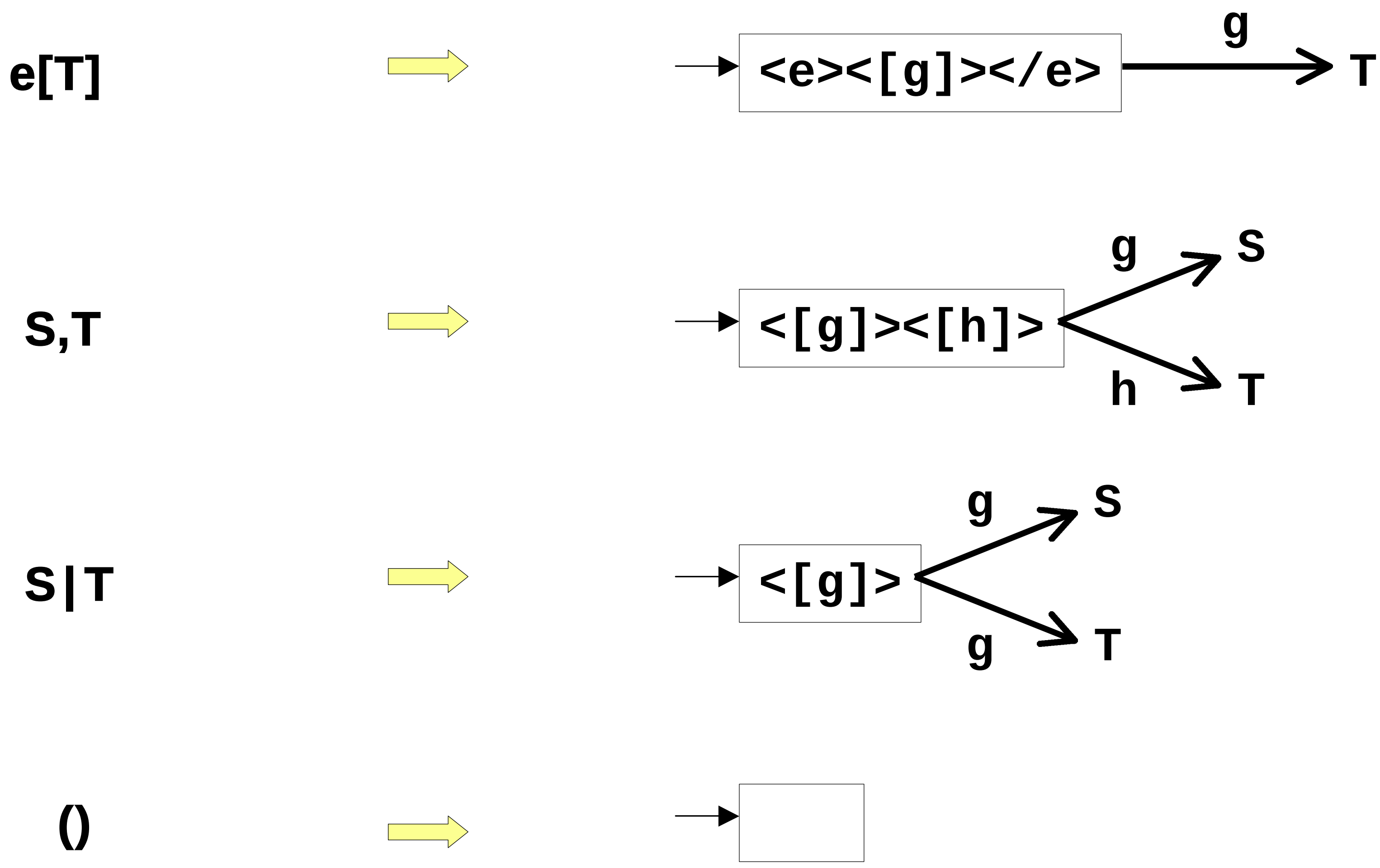
- functional language for general XML tree manipulation
  - functions perform deconstruction+construction
- regular expression types
  - element **e[T]**, concatenation **S,T**, union **S|T**, empty **()**, recursion
  - right-linearity requirement ensures regularity ("wellformedness")
- advanced pattern matching
  - first+longest match strategy
- type checking
  - subtyping based on regular language inclusion

Summary Graphs vs. Regular Expression Types?

- **Summary graphs** and **regular expression types** both define **sets of XML trees**

- They have *practically* the **same expressive power!** (if disregarding attributes and character data restrictions)

Regular Expression Types ® Summary Graphs



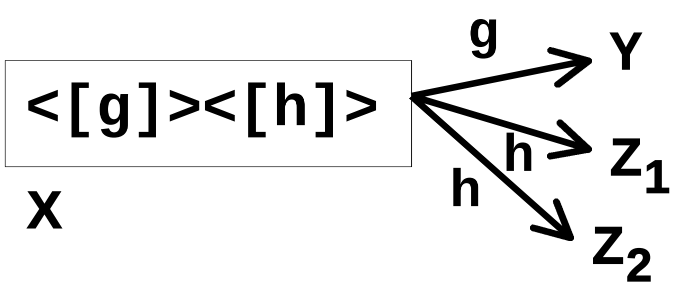
Summary Graphs ® Regular Expression Types

1. **Normalize** the SG such that template constants are of the form

   `<e><[g]></e>` or `<[g]><[h]>` or ` `

2. Assign **type variable** to each node

3. Read corresponding **type equations**, e.g.



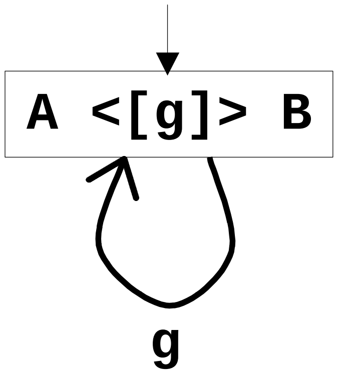
$$X = Y,Z$$
$$Z = Z_1 \mid Z_2$$

Wellformedness?

What happened to the right-linearity requirement?

When checking **validity** of a summary graph,
we **approximate** context-free languages with
regular languages...

Approximating Context-Free by Regular Languages



$L(SG) = A^n B^n$

approximation: A*B*
or: (A+B)*

– right/left linear context free grammars always define regular languages

JWIG as an XML Transformation Language

- So far, JWIG can only **construct** XML values (plug)
- We also need type-safe **deconstruction** (“unplug”)

x **>[** *path* **]** (“select”)
x **>[** *path* **=** *g* **]** (“gapify”)

(*path* is an **XPath** location path)

– the analysis can be extended to handle this!

Analyzing *Select* and *Gapify*

- Evaluate XPath location paths **symbolically** on summary graphs
- Each element is assigned a status value:
  - *all*
  - *some*
  - *definite*
  - *none*
  - *don't-know*
- Construct summary graph for select/gapify

Translating DTDs into Summary Graphs

- Essential when XML values come from an external source

- Preliminary experiments suggest that translation is feasible

Conclusion

JWIG provides a convenient **programming framework**:

- session-centered
- higher-order XML templates
- plug+unplug operations

In addition, the language design permits **static analysis:**

- plug, unplug, receive, and show analyses
- key idea: **summary graphs**

`http://www.brics.dk/JWIG/`