Paisley, Pattern Matching à la carte

Baltasar Trancón y Widemann [1 2] Markus Lepper[2]

[1] Universität Bayreuth Baltasar.Trancon@uni-bayreuth.de

[2] <semantics/> GmbH, Berlin post@markuslepper.eu

Praha, ICMT2012, 29. May 2012

1. **Design Principles**
   - Project Context
   - Pattern Matching Standards
   - Paisley Design Goals
2. **Paisley Implementation**
   - All Building Blocks
   - Pattern (Abstract Class) Interface
   - Getter for Field Tests
   - InstanceOf Test and Cast
   - Variables
3. **Demonstration**
   - Fool's Day Matches
   - Xhtml Specification Matches

1. **Design Principles**
   - Project Context
   - Pattern Matching Standards
   - Paisley Design Goals
2. **Paisley Implementation**
   - All Building Blocks
   - Pattern (Abstract Class) Interface
   - Getter for Field Tests
   - InstanceOf Test and Cast
   - Variables
3. **Demonstration**
   - Fool's Day Matches
   - Xhtml Specification Matches

Authors in General

- Compiler Construction / Language Design
- OPAL / DSLs / Specification Languages (TTCN-3, TCI, Z) / Temporal Logics / XML / Signal Processing

Project Context

- meta–tools — Collection of all our tools for generic programming, compiler construction and language processing
- Incorporating declarative techniques / ways of thinking into established "object oriented" technological context ***(here: Java)***
- Everything
  is a model.
  Every computation is a model transformation.

Authors in General

- Compiler Construction / Language Design
- OPAL / DSLs / Specification Languages (TTCN-3, TCI, Z) / Temporal Logics / XML / Signal Processing

Project Context

- ^{meta}–tools — Collection of all our tools for generic programming, compiler construction and language processing
- Incorporating declarative techniques / ways of thinking into established "object oriented" technological context ***(here: Java)***
- Everything
  is a model.
  Every computation is a model transformation.

Authors in General

- Compiler Construction / Language Design
- OPAL / DSLs / Specification Languages (TTCN-3, TCI, Z) / Temporal Logics / XML / Signal Processing

Project Context

- meta–tools — Collection of all our tools for generic programming, compiler construction and language processing
- Incorporating declarative techniques / ways of thinking into established "object oriented" technological context *(here: Java)*
- Everything
  is a model.
  Every computation is a model transformation.

Authors in General

- Compiler Construction / Language Design
- OPAL / DSLs / Specification Languages (TTCN-3, TCI, Z) / Temporal Logics / XML / Signal Processing

Project Context

- meta–tools — Collection of all our tools for generic programming, compiler construction and language processing
- Incorporating declarative techniques / ways of thinking into established "object oriented" technological context ***(here: Java)***
- Everything
  is a model.
  Every computation is a model transformation.

Authors in General

- Compiler Construction / Language Design
- OPAL / DSLs / Specification Languages (TTCN-3, TCI, Z) / Temporal Logics / XML / Signal Processing

Project Context

- ^{meta}–tools — Collection of all our tools for generic programming, compiler construction and language processing
- Incorporating declarative techniques / ways of thinking into established "object oriented" technological context ***(here: Java)***
- Everything
  is a model.
  Every computation is a model transformation.

Authors in General

- Compiler Construction / Language Design
- OPAL / DSLs / Specification Languages (TTCN-3, TCI, Z) / Temporal Logics / XML / Signal Processing

Project Context

- meta–tools — Collection of all our tools for generic programming, compiler construction and language processing
- Incorporating declarative techniques / ways of thinking into established "object oriented" technological context ***(here: Java)***
- Everything which is inside a computer is a model.
  Every computation is a model transformation.

Authors in General

- Compiler Construction / Language Design
- OPAL / DSLs / Specification Languages (TTCN-3, TCI, Z) / Temporal Logics / XML / Signal Processing

Project Context

- ^{meta}–tools — Collection of all our tools for generic programming, compiler construction and language processing
- Incorporating declarative techniques / ways of thinking into established “object oriented” technological context ***(here: Java)***
- Everything which is inside a computer or in our head is a model.
  Every computation is a model transformation.

Pattern Matching Standards

- String processing with regular expressions
- Functional programming with algebraic datatypes
- XML processing with XPath and XSLT
- Logic programming with goals & unification
- Model query and transformation

Pattern Matching Standards

- String processing with regular expressions
- Functional programming with algebraic datatypes
- XML processing with XPath and XSLT
- Logic programming with goals & unification
- Model query and transformation

Pattern Matching Standards

- String processing with regular expressions
- Functional programming with algebraic datatypes
- XML processing with XPath and XSLT
- Logic programming with goals & unification
- Model query and transformation

Pattern Matching Standards

- String processing with regular expressions
- Functional programming with algebraic datatypes
- XML processing with XPath and XSLT
- Logic programming with goals & unification
- Model query and transformation

Pattern Matching Standards

- String processing with regular expressions
- Functional programming with algebraic datatypes
- XML processing with XPath and XSLT
- Logic programming with goals & unification
- Model query and transformation

Pattern Matching Standards

- String processing with regular expressions
- Functional programming with algebraic datatypes
- XML processing with XPath and XSLT
- Logic programming with goals & unification
- Model query and transformation

Paisley Design Goals

1. Statically type-safe variables
2. Statically type-safe patterns
3. No language extension: independent of host compiler
4. No assumptions on host language beyond standard OOP
   $\Rightarrow$ ***algorithms easy portable out of Java***
5. No adaptation of model datatypes required
6. Support for multiple views per type
7. Declarative, readable, writeable, customizable
8. Full reification: no parsing/compilation overhead at runtime
9. Support for continuation-style nondeterminism
10. Nondeterminism incurs no significant cost unless actually used

Paisley Design Goals

1. Statically type-safe variables
2. Statically type-safe patterns
3. No language extension: independent of host compiler
4. No assumptions on host language beyond standard OOP
   $\Rightarrow$ ***algorithms easy portable out of Java***
5. No adaptation of model datatypes required
6. Support for multiple views per type
7. Declarative, readable, writeable, customizable
8. Full reification: no parsing/compilation overhead at runtime
9. Support for continuation-style nondeterminism
10. Nondeterminism incurs no significant cost unless actually used

Paisley Design Goals

1. Statically type-safe variables
2. Statically type-safe patterns
3. No language extension: independent of host compiler
4. No assumptions on host language beyond standard OOP
   $\Rightarrow$ ***algorithms easy portable out of Java***
5. No adaptation of model datatypes required
6. Support for multiple views per type
7. Declarative, readable, writeable, customizable
8. Full reification: no parsing/compilation overhead at runtime
9. Support for continuation-style nondeterminism
10. Nondeterminism incurs no significant cost unless actually used

Paisley Design Goals

1. Statically type-safe variables
2. Statically type-safe patterns
3. No language extension: independent of host compiler
4. No assumptions on host language beyond standard OOP
   ⇒ ***algorithms easy portable out of Java***
5. No adaptation of model datatypes required
6. Support for multiple views per type
7. Declarative, readable, writeable, customizable
8. Full reification: no parsing/compilation overhead at runtime
9. Support for continuation-style nondeterminism
10. Nondeterminism incurs no significant cost unless actually used

Paisley Design Goals

1. Statically type-safe variables
2. Statically type-safe patterns
3. No language extension: independent of host compiler
4. No assumptions on host language beyond standard OOP
   $\Rightarrow$ ***algorithms easy portable out of Java***
5. No adaptation of model datatypes required
6. Support for multiple views per type
7. Declarative, readable, writeable, customizable
8. Full reification: no parsing/compilation overhead at runtime
9. Support for continuation-style nondeterminism
10. Nondeterminism incurs no significant cost unless actually used

Paisley Design Goals

1. Statically type-safe variables
2. Statically type-safe patterns
3. No language extension: independent of host compiler
4. No assumptions on host language beyond standard OOP
   $\Rightarrow$ ***algorithms easy portable out of Java***
5. No adaptation of model datatypes required
6. Support for multiple views per type
7. Declarative, readable, writeable, customizable
8. Full reification: no parsing/compilation overhead at runtime
9. Support for continuation-style nondeterminism
10. Nondeterminism incurs no significant cost unless actually used

Paisley Design Goals

1. Statically type-safe variables
2. Statically type-safe patterns
3. No language extension: independent of host compiler
4. No assumptions on host language beyond standard OOP
   $\Rightarrow$ ***algorithms easy portable out of Java***
5. No adaptation of model datatypes required
6. Support for multiple views per type
7. Declarative, readable, writeable, customizable
8. Full reification: no parsing/compilation overhead at runtime
9. Support for continuation-style nondeterminism
10. Nondeterminism incurs no significant cost unless actually used

Paisley Design Goals

1. Statically type-safe variables
2. Statically type-safe patterns
3. No language extension: independent of host compiler
4. No assumptions on host language beyond standard OOP
   $\Rightarrow$ ***algorithms easy portable out of Java***
5. No adaptation of model datatypes required
6. Support for multiple views per type
7. Declarative, readable, writeable, customizable
8. Full reification: no parsing/compilation overhead at runtime
9. Support for continuation-style nondeterminism
10. Nondeterminism incurs no significant cost unless actually used

Paisley Design Goals

1. Statically type-safe variables
2. Statically type-safe patterns
3. No language extension: independent of host compiler
4. No assumptions on host language beyond standard OOP
   $\Rightarrow$ ***algorithms easy portable out of Java***
5. No adaptation of model datatypes required
6. Support for multiple views per type
7. Declarative, readable, writeable, customizable
8. Full reification: no parsing/compilation overhead at runtime
9. Support for continuation-style nondeterminism
10. Nondeterminism incurs no significant cost unless actually used

Paisley Design Goals

1. Statically type-safe variables
2. Statically type-safe patterns
3. No language extension: independent of host compiler
4. No assumptions on host language beyond standard OOP
   $\Rightarrow$ ***algorithms easy portable out of Java***
5. No adaptation of model datatypes required
6. Support for multiple views per type
7. Declarative, readable, writeable, customizable
8. Full reification: no parsing/compilation overhead at runtime
9. Support for continuation-style nondeterminism
10. Nondeterminism incurs no significant cost unless actually used

Paisley Design Goals

1. Statically type-safe variables
2. Statically type-safe patterns
3. No language extension: independent of host compiler
4. No assumptions on host language beyond standard OOP
   $\Rightarrow$ ***algorithms easy portable out of Java***
5. No adaptation of model datatypes required
6. Support for multiple views per type
7. Declarative, readable, writeable, customizable
8. Full reification: no parsing/compilation overhead at runtime
9. Support for continuation-style nondeterminism
10. Nondeterminism incurs no significant cost unless actually used

Paisley Design Goals

1. Statically type-safe variables
2. Statically type-safe patterns
3. No language extension: independent of host compiler
4. No assumptions on host language beyond standard OOP
   $\Rightarrow$ ***algorithms easy portable out of Java***
5. No adaptation of model datatypes required
6. Support for multiple views per type
7. Declarative, readable, writeable, customizable
8. Full reification: no parsing/compilation overhead at runtime
9. Support for continuation-style nondeterminism
10. Nondeterminism incurs no significant cost unless actually used

1. Design Principles
   - Project Context
   - Pattern Matching Standards
   - Paisley Design Goals

2. **Paisley Implementation**
   - All Building Blocks
   - Pattern (Abstract Class) Interface
   - Getter for Field Tests
   - InstanceOf Test and Cast
   - Variables

3. Demonstration
   - Fool's Day Matches
   - Xhtml Specification Matches

Paisley Building Blocks

- Algebraic constructors do have an inverse.
  Object-oriented constructors ***do not***.
- Instead use **getter** patterns
- and pre-defined **primitive type patterns**
  eq, equal, less, NaN, etc.
- and **class test/casting** and other **reflection based** patterns.
- Library of **basic combinators**
  - and of generic **combinators for collections**.
- Arbitrary **user defined** classes adhering to the
  `paisley.Pattern<A>` interface.
  (E.g. random generator !-)
- Independent of all these:
  Explicit bindings of **Variables**

Paisley Building Blocks

- Algebraic constructors do have an inverse.
  Object-oriented constructors ***do not***.
- Instead use **getter** patterns
- and pre-defined **primitive type patterns**
  eq, equal, less, NaN, etc.
- and **class test/casting** and other **reflection based** patterns.
- Library of **basic combinators**
  - and of generic **combinators for collections**.
- Arbitrary **user defined** classes adhering to the
  `paisley.Pattern<A>` interface.
  (E.g. random generator !-)
- Independent of all these:
  Explicit bindings of **Variables**

Paisley Building Blocks

- Algebraic constructors do have an inverse.
  Object-oriented constructors ***do not***.
- Instead use **getter** patterns
- and pre-defined **primitive type patterns**
  eq, equal, less, NaN, etc.
- and **class test/casting** and other **reflection based** patterns.
- Library of **basic combinators**
  - and of generic **combinators for collections**.
- Arbitrary **user defined** classes adhering to the
  `paisley.Pattern<A>` interface.
  (E.g. random generator !-)
- Independent of all these:
  Explicit bindings of **Variables**

Paisley Building Blocks

- Algebraic constructors do have an inverse.
  Object-oriented constructors ***do not***.
- Instead use **getter** patterns
- and pre-defined **primitive type patterns**
  eq, equal, less, NaN, etc.
- and **class test/casting** and other **reflection based** patterns.
- Library of **basic combinators**
  - and of generic **combinators for collections**.
- Arbitrary **user defined** classes adhering to the `paisley.Pattern<A>` interface.
  (E.g. random generator !-)
- Independent of all these:
  Explicit bindings of **Variables**

Paisley Building Blocks

- Algebraic constructors do have an inverse.
  Object-oriented constructors ***do not***.
- Instead use **getter** patterns
- and pre-defined **primitive type patterns**
  eq, equal, less, NaN, etc.
- and **class test/casting** and other **reflection based** patterns.
- Library of **basic combinators**
  - and of generic **combinators for collections**.
- Arbitrary **user defined** classes adhering to the `paisley.Pattern<A>` interface.
  (E.g. random generator !-)
- Independent of all these:
  Explicit bindings of **Variables**

Paisley Building Blocks

- Algebraic constructors do have an inverse.
  Object-oriented constructors ***do not***.
- Instead use **getter** patterns
- and pre-defined **primitive type patterns**
  eq, equal, less, NaN, etc.
- and **class test/casting** and other **reflection based** patterns.
- Library of **basic combinators**
  - and of generic **combinators for collections**.
- Arbitrary **user defined** classes adhering to the `paisley.Pattern<A>` interface.
  (E.g. random generator !-)
- Independent of all these:
  Explicit bindings of **Variables**

Paisley Building Blocks

- Algebraic constructors do have an inverse.
  Object-oriented constructors ***do not***.
- Instead use **getter** patterns
- and pre-defined **primitive type patterns**
  eq, equal, less, NaN, etc.
- and **class test/casting** and other **reflection based** patterns.
- Library of **basic combinators**
  - and of generic **combinators for collections**.
- Arbitrary **user defined** classes adhering to the `paisley.Pattern<A>` interface.
  (E.g. random generator !-)
- Independent of all these:
  Explicit bindings of **Variables**

Paisley Building Blocks

- Algebraic constructors do have an inverse.
  Object-oriented constructors ***do not***.
- Instead use **getter** patterns
- and pre-defined **primitive type patterns**
  eq, equal, less, NaN, etc.
- and **class test/casting** and other **reflection based** patterns.
- Library of **basic combinators**
  - and of generic **combinators for collections**.
- Arbitrary **user defined** classes adhering to the `paisley.Pattern<A>` interface.
  (E.g. random generator !-)
- Independent of all these:
  Explicit bindings of **Variables**

Paisley Building Blocks

- Algebraic constructors do have an inverse.
  Object-oriented constructors ***do not***.
- Instead use **getter** patterns
- and pre-defined **primitive type patterns**
  eq, equal, less, NaN, etc.
- and **class test/casting** and other **reflection based** patterns.
- Library of **basic combinators**
  - and of generic **combinators for collections**.
- Arbitrary **user defined** classes adhering to the `paisley.Pattern<A>` interface.
  (E.g. random generator !-)
- Independent of all these:
  Explicit bindings of **Variables**

Paisley Building Blocks

- Algebraic constructors do have an inverse.
  Object-oriented constructors ***do not***.
- Instead use **getter** patterns
- and pre-defined **primitive type patterns**
  eq, equal, less, NaN, etc.
- and **class test/casting** and other **reflection based** patterns.
- Library of **basic combinators**
  - and of generic **combinators for collections**.
- Arbitrary **user defined** classes adhering to the
  `paisley.Pattern<A>` interface.
  (E.g. random generator !-)
- Independent of all these:
  Explicit bindings of **Variables**

Paisley Building Blocks

- Algebraic constructors do have an inverse.
  Object-oriented constructors ***do not***.
- Instead use **getter** patterns (⟵ user must code)
- and pre-defined **primitive type patterns**
  eq, equal, less, NaN, etc.
- and **class test/casting** and other **reflection based** patterns.
- Library of **basic combinators**
  - and of generic **combinators for collections**.
- Arbitrary **user defined** classes adhering to the
  `paisley.Pattern<A>` interface. (⟵ user may code)
  (E.g. random generator !-)
- Independent of all these:
  Explicit bindings of **Variables**

Paisley Building Blocks

- Algebraic constructors do have an inverse.
  Object-oriented constructors ***do not***.
- Instead use **getter** patterns (⟵ user must code)
- and pre-defined **primitive type patterns**
  eq, equal, less, NaN, etc.
- and **class test/casting** and other **reflection based** patterns.
- Library of **basic combinators** ("`either`" ⇒ nondet.!)
  - and of generic **combinators for collections**. (⇒ **nondet.!**)
- Arbitrary **user defined** classes adhering to the
  `paisley.Pattern<A>` interface. (⟵ user may code)
  (E.g. random generator !-) (may imply nondet.)
- Independent of all these:
  Explicit bindings of **Variables**

Pattern (Abstract Class) Interface

```
public abstract class Pattern<A> {
  public abstract boolean match(A) ;
  public abstract boolean matchAgain() ;

  // some meta & factory methods:
  public final void cut(){...}
  public final void clear(){...}
  public boolean preserves(Variable<?> var, ...)  {...}
  ...
}
```

Getter for Field Tests

```
class D {
 public int f = 17 ;
 ...  }
```

```
static Pattern<D> match_D_f(Pattern<Integer> p){
  return p.transform
    ( new Function<D,Integer>(){
        public Integer apply(final D d) { return d.f; }}
     );
```

(⟵ Model compilers from meta–tools generate this automatically or even THIS !-)

Getter for Field Tests

```
class D {
 public int f = 17 ;
 ...  }
```

```
static Pattern<D> match_D_f(Pattern<Integer> p){
  return p.transform
    ( new Function<D,Integer>(){
        public Integer apply(final D d) { return d.f; }}
     );
```

(⟵ Model compilers from meta–tools generate this automatically or even THIS !-)

Getter for Field Tests

```
class D {
 public int f = 17 ;
 ...  }
```

```
static Pattern<D> match_D_f(Pattern<Integer> p){
  return p.transform
    ( new Function<D,Integer>(){
        public Integer apply(final D d) { return d.f; }}
     );
```

(⟵ Model compilers from ^{meta}–tools generate this automatically
or even THIS !-)

Getter for Field Tests

```
class D {
 public int f = 17 ;
 ...  }
```

```
static Pattern<D> match_D_f(Pattern<Integer> p){
  return p.transform
    ( new Function<D,Integer>(){
        public Integer apply(final D d) { return d.f; }}
     );
```

(⟵ Model compilers from $^{\text{meta}}$–tools generate this automatically or even THIS !-)

InstanceOf Test and Cast

```
class D extends C {
 ...  }

Pattern<D> pattern_d = ...;
Pattern<? super C> pattern_c =
  pattern_d.forInstancesOf(D.class);
C c = ...;
if (pattern_c.match(c)) { ...}
```

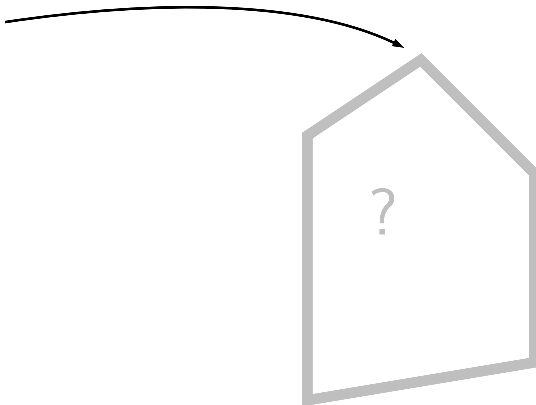
InstanceOf Test and Cast

```
class D extends C {
 ...  }

Pattern<D> pattern_d = ...;
Pattern<? super C> pattern_c =
  pattern_d.forInstancesOf(D.class);
C c = ...;
if (pattern_c.match(c)) { ...}
```

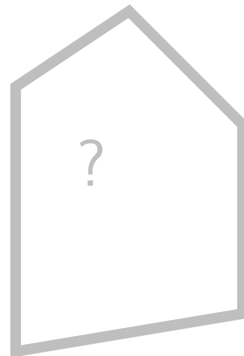
Variables

```
final D datum = ...
```



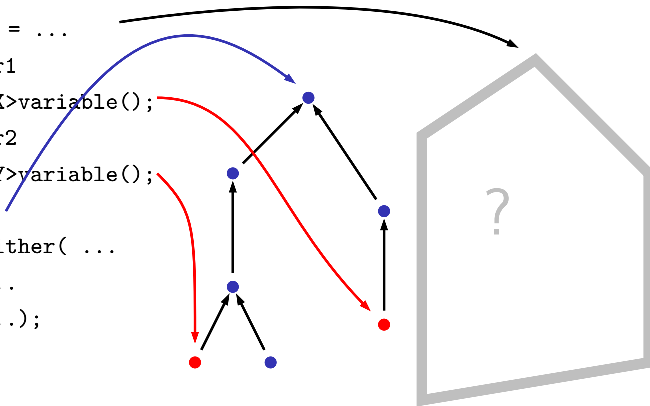
Variables

```
 final D datum = ...
Variable<X> var1
   = Pattern.<X>variable();
Variable<Y> var2
   = Pattern.<Y>variable();
```



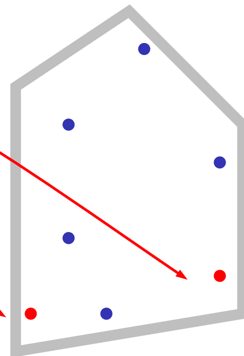
Variables

```
 final D datum = ...
Variable<X> var1
   = Pattern.<X>variable();
Variable<Y> var2
   = Pattern.<Y>variable();
Pattern<D> p
   = Pattern.either( ...
   ...(var1) ...
   ...(var2) ...);
```



Variables

```
 final D datum = ...
Variable<X> var1
   = Pattern.<X>variable();
Variable<Y> var2
   = Pattern.<Y>variable();
Pattern<D> p
   = Pattern.either( ...
   ...(var1) ...
   ...(var2) ...);
if (p.match(datum)){...
   // maybe var1/var2 is meaningful
}
```



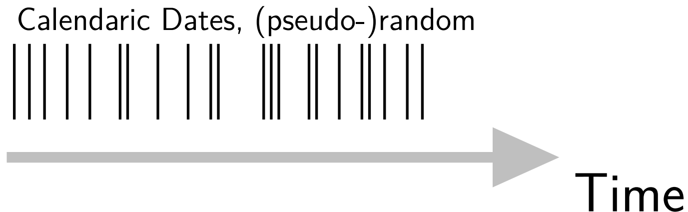
1 Design Principles
- Project Context
- Pattern Matching Standards
- Paisley Design Goals

2 Paisley Implementation
- All Building Blocks
- Pattern (Abstract Class) Interface
- Getter for Field Tests
- InstanceOf Test and Cast
- Variables

3 **Demonstration**
- Fool's Day Matches
- Xhtml Specification Matches

Fool's Day Matches

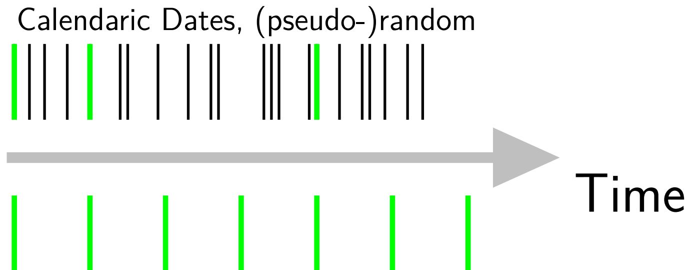
Fool's Day Matches



Fool's Day Matches

Do they hit a "First of April" ?

Fool's Day Matches



Do they hit a "First of April" ?

Xhtml Specification Matches

Search the Xhtml specification for external links
contained in “definition lists”.

```
<html xmlns="http://www.w3.org/1999/xhtml">
|____<h1>

|____<p>
|____<dl>
     |____
     |____<dt> any text, should be a "definition term"
     |____<dd> mixed content, being the "explanation"
               |____<a href="hr1">
               |____<a href="hr2">
     |____<dt>
     |____<dd>
               |____
```

More ...

- ... in the proceedings
- ... ^{meta}–tools users' guide at `http://bandm.eu`
- ... Paisley ***demo download*** at `http://bandm.eu/metatools/paisley`