

•  
•  
•

# java front for xt

## *techniques and applications*

Martin Bravenboer

`martin@nbravenboer.org`

Institute of Information and Computing Sciences

University Utrecht

The Netherlands

# contents

- grammars
  - disambiguation
  - unit-testing
- applications
  - code-generation
    - current techniques
    - code generation with java front
    - inclusion mechanism
- pretty-printing

- 
- 
- 

# introduction

# goals

- tooling required for java meta programming
- compatible and up to date
- programmer friendly
- application of cool techniques

# what's in it right now?

- grammars for:
  - java 2 version 1.4
  - generic java
- pretty-printer
- code generation
  - grammars available 'in stratego'.
  - utils

# what should be in it?

- (de)sugaring
- format checkers and type-matching strategies
- semantic analysis
- javadoc grammar
- java bytecode access
  - required for semantic analysis
  - decompilation
  - compilation

- 
- 
- 

# grammars

# chic grammars

The sdf grammars for java 2 version 1.4 and generic java are completely handcrafted:

- context-free priorities
- lexical restrictions
- rejections

# disambiguation

two solutions for disambiguation (Aho):

1. *unambiguous grammars*: every priority level introduces a new non-terminal
2. *ambiguous grammars* with additional rules

The Syntax Definition Formalism (SDF) offers such mechanisms to resolve ambiguities.

# lexical disambiguation: comments

## java language specification:

TraditionalComment:

```
/* NotStar CommentTail
```

CommentTail:

```
* CommentTailStar
```

```
NotStar CommentTail
```

CommentTailStar:

```
/
```

```
* CommentTailStar
```

```
NotStarNotSlash CommentTail
```

# lexical disambiguation: comments

java front uses follow restriction:

lexical syntax

```
"/*" (~[\\*] | Asterisk)* "*" /" -> Comment  
"*" -> Asterisk
```

lexical restrictions

```
"/*" -/- [\\*]  
Asterisk -/- [\\/]
```

# lexical disambiguation: identifiers

## java language specification:

Identifier:

IdentifierChars

but not a Keyword

or BooleanLiteral

or NullLiteral

## java front uses reject productions:

Keyword -> Id {reject}

BooleanLiteral -> Id {reject}

NullLiteral -> Id {reject}

# context-free priorities: dangling else

## java language specification:

IfThenStm:       if ( Expr ) Stm

IfThenElseStm: if ( Expr ) StmNoShortIf else Stm

IfThenElseStmNoShortIf:

    if ( Expression ) StmNoShortIf else StmNoShortIf

StmNoShortIf:

    StmWithoutTrailingSubstm

    LabeledStmNoShortIf

    IfThenElseStmNoShortIf

    WhileStmNoShortIf

    ForStmNoShortIf

# context-free priorities: dangling else

java front uses context-free priorities:

```
context-free priorities
```

```
"if" "(" Expr ")" Stm "else" Stm -> Stm
> {
  Id ":" Stm -> Stm
  "if" "(" Expr ")" Stm -> Stm
  "while" "(" Expr ")" Stm -> Stm
  "for" "(" ForInit? ";" Expr? ";" ForUpdate? ")" Stm
    -> Stm
}
```

# context-free priorities: new array

## java language specification:

Primary:

PrimaryNoNewArray

ArrayCreationExpression

PrimaryNoNewArray:

Literal

( Expression ) ...

## java front uses reject productions:

Expr "[" Expr "]" -> ArrayAccess {cons("ArrayAccess")}

ArrayCreationExpr "[" Expr "]" -> ArrayAccess {reject}

ArrayCreationExpr -> Expr

# context-free priorities: expressions (1)

```
e++ e--  
++e --e + - ~ ! (t)e  
* / %  
+ -  
<< >> >>>  
instanceof < > <= >=  
== !=  
&  
^  
|  
&&  
||  
? :  
= *= /= \% = += -= <<= ...
```

# context-free priorities: expressions (2)

<code>e++ e--</code>	<code>PostfixExpr</code>
<code>++e --e + - ~ ! (t)e</code>	<code>UnaryExpr</code>
<code>* / %</code>	<code>MultiplicativeExpr</code>
<code>+ -</code>	<code>AdditiveExpr</code>
<code>&lt;&lt; &gt;&gt; &gt;&gt;&gt;</code>	<code>ShiftExpr</code>
<code>instanceof &lt; &gt; &lt;= &gt;=</code>	<code>RelationalExpr</code>
<code>== !=</code>	<code>EqualityExpr</code>
<code>&amp;</code>	<code>AndExpr</code>
<code>^</code>	<code>ExclusiveOrExpr</code>
<code> </code>	<code>InclusiveOrExpr</code>
<code>&amp;&amp;</code>	<code>ConditionalAndExpr</code>
<code>  </code>	<code>ConditionalOrExpr</code>
<code>? :</code>	<code>ConditionalExpr</code>
<code>= *= /= \%= += -= &lt;&lt;= ...</code>	<code>AssignmentExpr</code>

# context-free priorities: expressions (3)

```
    "!" Expr -> Expr
    "(" Type ")" Expr -> Expr
  }
> {left:
    Expr "*" Expr -> Expr
    Expr "/" Expr -> Expr
    Expr "%" Expr -> Expr
  }
> {left:
    Expr "+" Expr -> Expr
    Expr "-" Expr -> Expr
  }
> {left:
    Expr "<<" Expr -> Expr
```

# parse-unit

Grammars are controlled by unit-tests.

parse-unit advantages:

- tests specified in *ATerms*: no need to compile
- no configuration trouble, just call 1 tool

creation of test-suites:

- distant future: all tests from JACKS
- SGLR doesn't support Unicode

# parse-unit: test-suite

```
TestSuite(  
  Name("Expressions")  
, Sort("Expr")  
, ParseTable(File("../grammar/basic/Java.tbl"))  
, Tests(  
  [ Test(Descr("null literal")  
    , String("null"), Lit(Null()))  
  ]  
)  
)
```

# parse-unit: tests

```
, Test(Descr("always take longest match for --")
, String("1--2"), Failure()
)
, Test(Descr("addition is left associative")
, String("1 + 2 + 3")
, Plus(Plus(Lit(Deci("1")), Lit(Deci("2"))), Lit(Deci("3")))
)
, Test(Descr("multiplication has higher priority than addition")
, String("1 + 2 * 3")
, Plus(Lit(Deci("1")), Mul(Lit(Deci("2")), Lit(Deci("3"))))
)
, Test(Descr("/**/ comment separates tokens")
, String("class T3710 {int/* */i;}"), Success()
)
, Test(Descr("/**/ comment cannot appear in literal")
, String("class T3712 {float f = 1./* */0;}"), Failure()
)
```

- 
- 
- 

# applications

# some directions

- java to java transformations
- java code generation
- java language extensions
- documentation generation

# java language extensions

SDF's modularity is magnificent for implementing language extensions:

- embedded SQL
- embedded XML
- embedded userinterface language
- small extensions: for-each, enums, tuple/list syntax, assertions
- functional extensions
- AST construction with concrete syntax

# java to java transformations

- refactoring implementations with concrete syntax
- source code instrumentation: debugging, code-coverage
- all usual optimizations (common subexpression elimination, constant propagation, partial evaluation, inlining), but concrete syntax is often not very useful there.

# code generation

# code generation: some areas

- compilation to Java  
→ implementation of Domain Specific Languages
- object-relational mappings
- XML data binding
- parser generators
- visitor combinators ;)

# code generation: goals

→ tools for generative programming

*ASF+SDF MetaEnvironment* : “From the perspective of generative programming, this term rewriting system is interesting because it allows programming in concrete syntax.”

Stratego can now do this as well !

# code generation: current techniques

- abstract syntax based
  - construction of abstract syntax *trees*
  - creation of objects, structs, records
- concrete syntax based
  1. as string (literal) in meta language (Java, XSLT)
  2. templates engine

# abstract versus concrete syntax (1)

```
1 < 2 == 3 < 4      Eq( Lt(Lit(Deci("1")), Lit(Deci("2")))
                    , Lt(Lit(Deci("3")), Lit(Deci("4"))))
```

```
x = 1                Assign(ExprName("x"), Lit(Deci("1")))
```

```
x = var.method()    Assign(ExprName("x"), Invoke(Method(
                    MethodName(AmbName(["var"]), "method")), []))
```

```
country = _loader.getCountry(countryID)
```

```
Assign( ExprName("country"),
        Invoke(Method(MethodName(AmbName(["_loader"]), "getCountry")),
                [Name(ExprName("countryID"))]))
```

# abstract versus concrete syntax (2)

```
if(a)
  while(b)
    if(c)
      c = 1;
    else
      d = 2;
```

```
If (Name (ExprName ( " a " ) ) ,
  While (Name (ExprName ( " b " ) ) ,
    If (Name (ExprName ( " c " ) ) ,
      Expr (Assign (ExprName ( " c " ) , Lit (Deci ( " 1 " ) ) ) ) )
    , Expr (Assign (ExprName ( " d " ) , Lit (Deci ( " 2 " ) ) ) ) ) ) ) )
```

# code generation: current tools

**XSLTC:** custom AST + Apache's BCEL

**JAXB:** custom AST (pretty printing in AST)

**Castor:** combination of AST and text

**RMIC:** text

**Jasper:** text

**Bistro:** text

**SableCC:** text (macro mechanism)

**JJTree/JavaCC:** text

# code generation with java front (1)

Meta programming with concrete syntax object syntax:

The absent yet present abstract syntax

1. *concrete syntax* but embedded in the meta language
2. compile time conversion to *abstract syntax*

# code generation with java front (2)

```
[java:class-body-dec*[\n    private ~type ~field ;\n\n    public ~type ~getname () {\n        return ~id:field;\n    }\n\n    public void ~setname (final ~type ~param) {\n        if(~id:param == null) {\n            throw new IllegalArgumentException("cannot be null");\n        }\n\n        ~field = ~id:param;\n    }\n    ]]
```

# code generation with java front (3)

```
(oname, Reference(Name(name), Type(Object(tname))))  
->  
[java:block-stm* [  
    int ~id:idvar = resultSet.getInt(~string:idvar);  
  
    ~storedtype ~localvar = null;  
    try {  
        ~id:localvar = _loader . ~id:getname ( ~id:idvar );  
    } catch( LoadException exc ) {  
        throw new SQLException("... " + exc.getMessage());  
    }  
]]
```

# code generation with java front (4)

```
Object(Name(name), Properties(ps), Relations(rs))
->
[java:compilation-unit[
  package ~<domain-package>;

  public interface ~name extends ~<i-domain-object> {

    public <E extends Exception> void
      acceptVisit( ~id:visitor <E> visitor) throws E;

    ~*members
  }
]]
```

# exploiting non-terminals

```
Prim(String()) -> [java:type[ String  ]]
```

```
Prim(Integer()) -> [java:type[ int    ]]
```

```
Prim(Boolean()) -> [java:type[ boolean ]]
```

```
gen-expr: s -> [java:expr[ ~id:s  ]]  
          where <is-string> s
```

```
gen-expr: i -> [java:expr[ ~deci:s ]]  
          where <is-int> i  
                ; <int-to-string> i => s
```

# code generation utils (1)

path:

```
(root-dir, CompilationUnit(None(), _, _)) -> root-dir
```

path:

```
(root-dir, CompilationUnit(Some(  
    PackageDec(PackageName(ids))), _, _))
```

->

```
<separate-by(!"/"); concat-strings> [root-dir | ids]
```

filename:

```
CompilationUnit(_, _, [type-dec])
```

->

```
<typename> type-dec
```

## code generation utils (2)

```
get-method = <conc-strings> ("get", <id>)
```

```
set-method = <conc-strings> ("set", <id>)
```

```
localvar-name = first-lower-case
```

```
param-name = first-lower-case
```

```
field-name = <conc-strings> ("_", <first-lower-case>)
```

```
first-lower-case =
```

```
  string-as-list( [lc | id] )
```

# inclusion mechanism (1)

→ Extending generated code is a problem.

Simple but effective inclusion mechanism solves this:

1. add import declarations
2. include any class/interface body declaration
3. extend some superclass (class)
4. implement/extend interfaces

# inclusion mechanism (2)

include-jtree:

```
(include-dir, content) -> new-content
```

```
  where <jtree-file> (include-dir, content) => file
```

```
    ; <file-exists> file
```

```
    ; <debug(!"File found to include: ")> file
```

```
    ; <inject> (<ReadFromFile> file, content)
```

```
      => new-content
```

```
    ; <debug(!"Included extra source from ")> file
```

# inclusion mechanism (3)

inject:

```
( CompilationUnit(_, i-imports, [i-type-dec])  
  , CompilationUnit(p, imports, [type-dec])  
  )
```

->

```
CompilationUnit(p, new-imports, [new-type-dec])
```

```
  where <conc> (i-imports, imports) => new-imports
```

```
        ; <inject-typedec> (i-type-dec, type-dec)
```

```
          => new-type-dec
```

# pretty printing

# pretty printing in xt

BOX language: generic pretty printing

1. *pretty print table*:

- + pleasant concrete syntax for BOX
- limited control

2. *stratego with BOX abstract syntax*:

- verbose/unclear abstract syntax for BOX
- + full control

# java pretty printer (1)

- priorities of expressions
- many parameters, big expressions, else if
- overloading of constructor names
- different layouts

Solution:

- *Stratego with BOX concrete syntax:*
  - + pleasant concrete syntax for BOX
  - + full control

# java pretty printer (2)

```
Cast(t, e) -> -- H hs=0 ["(" ~t ")" ~e] --
```

```
Throw(e) -> -- H hs=1 [KW["throw"] H hs=0[~e ";" ]] --
```

```
Param(Some(Final()), type, vardecid)
```

```
-> -- H hs=1 [KW["final"] ~type ~vardecid] --
```

```
ConstructorBody(None(), stms)
```

```
-> <block-structure> (0, stms)
```

```
ConstructorBody(Some(cinvoke), stms)
```

```
-> <block-structure> (0, [cinvoke | stms])
```

```
TypeParams(params) -> -- H hs=0 ["<" ~parameters ">"] --
```

```
where <separate-by-comma> params => parameters
```

# format checker and type-matching

- No need to write a format checker by hand.
- fc-gen generates format-checkers and type-matching strategies from Stratego signatures.
- Stratego signature can be generated from a SDF definition
- Problem: injections

- 
- 
- 



# Conclusions



- 
- 
- 
- 
- 
- 
- 
- 
-

# to do

near future:

- minor grammar tweaking
- extend pretty printer with priorities
- add much more unit-tests

unpredictable:

- semantic analysis and desugaring
- bytecode tools
- javadoc grammar

# contributions

- Stratego is now one of the coolest languages for Java code generation.
- *BOX in Stratego* can be used to implement pretty printers for other languages.
- *parse-unit* can be used for testing other grammars.
- Experience in *defining priorities* of C-like languages can be used to produce grammars for C, C#, VB .NET et cetera very quickly.

Questions? Remarks?