# Implementing an XQuery/XSLT hybrid

Parsing and compiling Carrot

Evan Lenz
Software Developer, Community
MarkLogic Corporation

# Quick poll

# Praises for XSLT

- Template rules are really elegant and powerful
- It's mature in its set of features
- Powerful modularization features (`<xsl:import>`)

# Praises for XQuery

- Concise syntax
- Highly composable syntax
  - An element constructor is an expression
    - So you can write things like: `foo/<bar/>`

# Gripes about XSLT

- Two layers of syntax, which can’t be freely composed
  - You can’t nest an instruction inside an expression
    - E.g., you can’t apply templates inside an XPath expression
- Verbose syntax
  - In general
  - In particular, for function definitions and parameter-passing

# Gripes about XQuery

- Conflation of modules and namespaces
  - Don’t like being forced to use namespace URIs
- Distinction between main and library modules
  - You can’t reuse a main module
  - Reuse requires refactoring
- No template rules!

# A lot in common

- The same data model (XPath 2.0)
- Much of the same syntax (XPath 2.0)

# Feeling boxed-in

- XSLT’s lack of composability
- XQuery’s lack of template rules
- Don’t like having to pick between two languages all the time

# The solution?

# Disclaimer

- My own personal project & opinions

# The solution?

# “Carrot”

- A hybrid of XQuery and XSLT
- More than just an alternative syntax for XSLT
- Carrot combines:
  - the friendly syntax and composability of XQuery expressions
  - the power and flexibility of template rules in XSLT
- A “host language” for XQuery expressions

# Overall design approach

- 95% of semantics defined by reference to XQuery and XSLT
- 90% of syntax defined by reference to XQuery

# Carrot in a nutshell

# Intro by example

- A rule definition in XSLT:

```
<xsl:template match="para">
  <p>
    <xsl:apply-templates/>
  </p>
</xsl:template>
```

# Intro by example

- A rule definition in XSLT:

```
<xsl:template match="para">
  <p>
    <xsl:apply-templates/>
  </p>
</xsl:template>
```

- A rule definition in Carrot:

```
^(para) := <p>{^()}</p>;
```

# Intro by example

- A rule definition in XSLT:

```
<xsl:template match="para">
  <p>
    <xsl:apply-templates/>
  </p>
</xsl:template>
```

- A rule definition in Carrot:

```
^(para) := <p>{^()}</p>;
```

# Intro by example

- A rule definition in XSLT:

```
<xsl:template match="para">
  <p>
    <xsl:apply-templates/>
  </p>
</xsl:template>
```

- A rule definition in Carrot:

```
^(para) := <p>{^()}</p>;
```

# Intro by example

- A rule definition in XSLT:

```
<xsl:template match="para">
  <p>
    <xsl:apply-templates/>
  </p>
</xsl:template>
```

- A rule definition in Carrot:

```
^(para) := <p>{^()}</p>;
```

# Intro by example

- This:

```
^()
```

- Is short for this:

```
^(node())
```

- Just as, in XSLT, this:

```
<xsl:apply-templates/>
```

- Is short for this:

```
<xsl:apply-templates select="node()"/>
```

# Intro by example

- Another rule definition in Carrot:

```
^toc(section) := <li>{ ^toc() }</li>;
```

- The same rule definition in XSLT:

```
<xsl:template match="section" mode="toc">
  <li>
    <xsl:apply-templates mode="toc"/>
  </li>
</xsl:template>
```

# Intro by example

- Another rule definition in Carrot:

```
^toc(section) := <li>{ ^toc() }</li>;
```

- The same rule definition in XSLT:

```
<xsl:template match="section" mode="toc">
  <li>
    <xsl:apply-templates mode="toc"/>
  </li>
</xsl:template>
```

# Intro by example

- Another rule definition in Carrot:

```
^toc(section) := <li>{ ^toc() }</li>;
```

- The same rule definition in XSLT:

```
<xsl:template match="section" mode="toc">
  <li>
    <xsl:apply-templates mode="toc"/>
  </li>
</xsl:template>
```

# Intro by example

- Another rule definition in Carrot:

```
^toc(section) := <li>{ ^toc() }</li>;
```

- The same rule definition in XSLT:

```
<xsl:template match="section" mode="toc">
  <li>
    <xsl:apply-templates mode="toc"/>
  </li>
</xsl:template>
```

# Intro by example

- Another rule definition in Carrot:

```
^toc(section) := <li>{ ^toc() }</li>;
```

- The same rule definition in XSLT:

```
<xsl:template match="section" mode="toc">
  <li>
    <xsl:apply-templates mode="toc"/>
  </li>
</xsl:template>
```

# The identity transform

- In Carrot:

```
^(@*|node()) := copy{ ^(@*|node()) };
```

- In XSLT:

```
<xsl:template match="@* | node()">
  <xsl:copy>
    <xsl:apply-templates select="@* | node()"/>
  </xsl:copy>
</xsl:template>
```

# The identity transform

- In Carrot:

```
^(@*|node()) := copy{ ^(@*|node()) };
```

- In XSLT:

```
<xsl:template match="@* | node()">
  <xsl:copy>
    <xsl:apply-templates select="@* | node()"/>
  </xsl:copy>
</xsl:template>
```

# The identity transform

- In Carrot:

```
^(@*|node()) := copy{ ^(@*|node()) };
```

- In XSLT:

```
<xsl:template match="@* | node()">
  <xsl:copy>
    <xsl:apply-templates select="@* | node()"/>
  </xsl:copy>
</xsl:template>
```

# The identity transform

- In Carrot:

```
^(@*|node()) := copy{ ^(@*|node()) };
```

- In XSLT:

```
<xsl:template match="@* | node()">
  <xsl:copy>
    <xsl:apply-templates select="@* | node()"/>
  </xsl:copy>
</xsl:template>
```

# Note the asymmetry

- This definition is illegal (missing pattern):

```
^() := <foo/>;
```

- Just as this template rule is illegal:

```
<xsl:template match=""><foo/></xsl:template>
```

- However, when invoking, you can omit the argument:

```
^()
```

- Just as in XSLT:

```
<xsl:apply-templates/>
```

# An XSLT example

```
<xsl:transform version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <html>
      <head>
        <xsl:copy-of select="/doc/title"/>
      </head>
      <body>
       <xsl:apply-templates select="/doc/para"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="para">
    <p>
      <xsl:apply-templates/>
    </p>
  </xsl:template>

</xsl:stylesheet>
```

# An XSLT example

```
<xsl:transform version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <html>
      <head>
        <xsl:copy-of select="/doc/title"/>
      </head>
      <body>
       <xsl:apply-templates select="/doc/para"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="para">
    <p>
      <xsl:apply-templates/>
    </p>
  </xsl:template>

</xsl:stylesheet>
```

# The equivalent in Carrot

```
^(/) :=
 <html>
   <head>{ /doc/title    }</head>
   <body>{ ^(/doc/para) }</body>
 </html>;

^(para) := <p>{ ^() }</p>;
```

# The equivalent in Carrot

```
^(/)  :=
 <html>
   <head>{ /doc/title   }</head>
   <body>{ ^(/doc/para) }</body>
 </html>;

^(para)  := <p>{ ^() }</p>;
```

# A Carrot module

- Consists of a set of unordered *definitions*
- Three kinds of definitions:
  - Global variables
  - Functions
  - Rules
- Unlike XQuery, there is no top-level expression—only definitions
  - Carrot is like XSLT in this regard

# An example of each

```
namespace my="http://example.com";

$foo     := "a string";    (: VarDecl :)
my:foo() := $foo;          (: FunctionDecl :)
^(/)     := my:foo();      (: RuleDecl :)
```

# An example of each

```
namespace my="http://example.com";

$foo       := "a string";    (: VarDecl :)
my:foo()   := $foo;          (: FunctionDecl :)
^(/)       := my:foo();      (: RuleDecl :)
```

# An example of each

```
namespace my="http://example.com";

$foo     := "a string";    (: VarDecl :)
my:foo() := $foo;          (: FunctionDecl :)
^(/)     := my:foo();      (: RuleDecl :)
```

# An example of each

```
namespace my="http://example.com";

$foo     := "a string";   (: VarDecl :)
my:foo() := $foo;         (: FunctionDecl :)
^(/)     := my:foo();     (: RuleDecl :)
```

# An example of each

```
namespace my="http://example.com";

$foo     := "a string";   (: VarDecl :)
my:foo() := $foo;         (: FunctionDecl :)
^(/)     := my:foo();     (: RuleDecl :)
```

# An example of each

```
namespace my="http://example.com";

$foo     := "a string";    (: VarDecl :)
my:foo() := $foo;          (: FunctionDecl :)
^(/)     := my:foo();      (: RuleDecl :)
```

RHS is always an
expression (Expr)

# Carrot expressions

- Same as an expression in XQuery, with these additions:
  1. ruleset invocations — `^mode(nodes)`
  2. shallow `copy{…}` constructors
  3. text node literals — `` `my text node` ``

# Implementation approaches

# Three broad approaches

1. Native implementation
2. Compilation to XSLT
3. Compilation to XQuery

# Three broad approaches

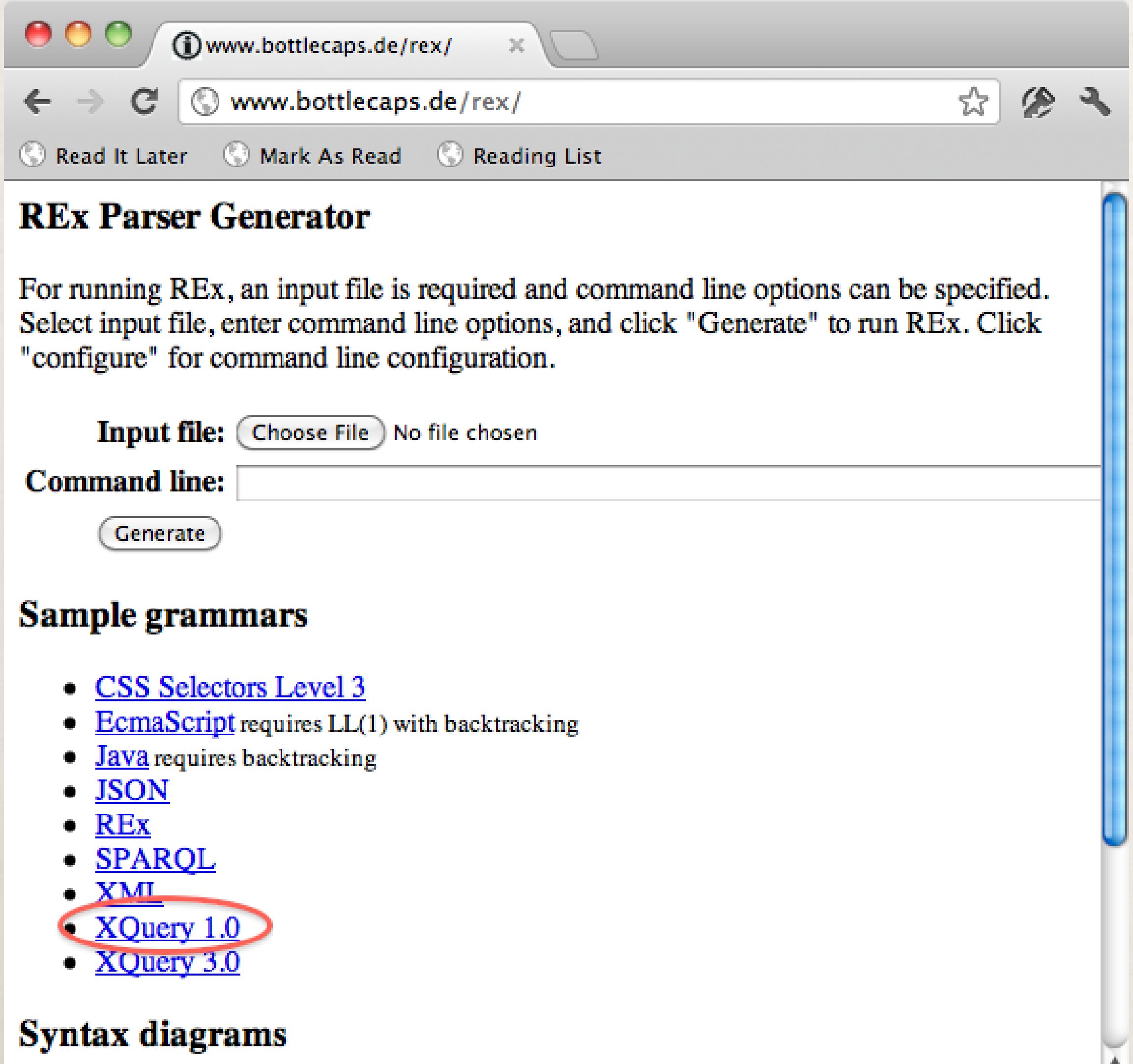
1. Native implementation
2. Compilation to XSLT
3. Compilation to XQuery

# Parsing Carrot

# Three steps to implementation

1. Start with the XQuery 1.0 EBNF grammar:
   - http://www.bottlecaps.de/rex/XQueryV10.ebnf
2. Convert it to a Carrot grammar by hand:
   - https://github.com/evanlenz/Carrot/blob/master/parser/Carrot.ebnf
3. Auto-generate the parser, using this tool:
   - http://www.bottlecaps.de/rex/

# Step 1: Download the XQuery grammar



## REx Parser Generator

For running REx, an input file is required and command line options can be specified. Select input file, enter command line options, and click "Generate" to run REx. Click "configure" for command line configuration.

**Input file:** Choose File No file chosen

**Command line:**

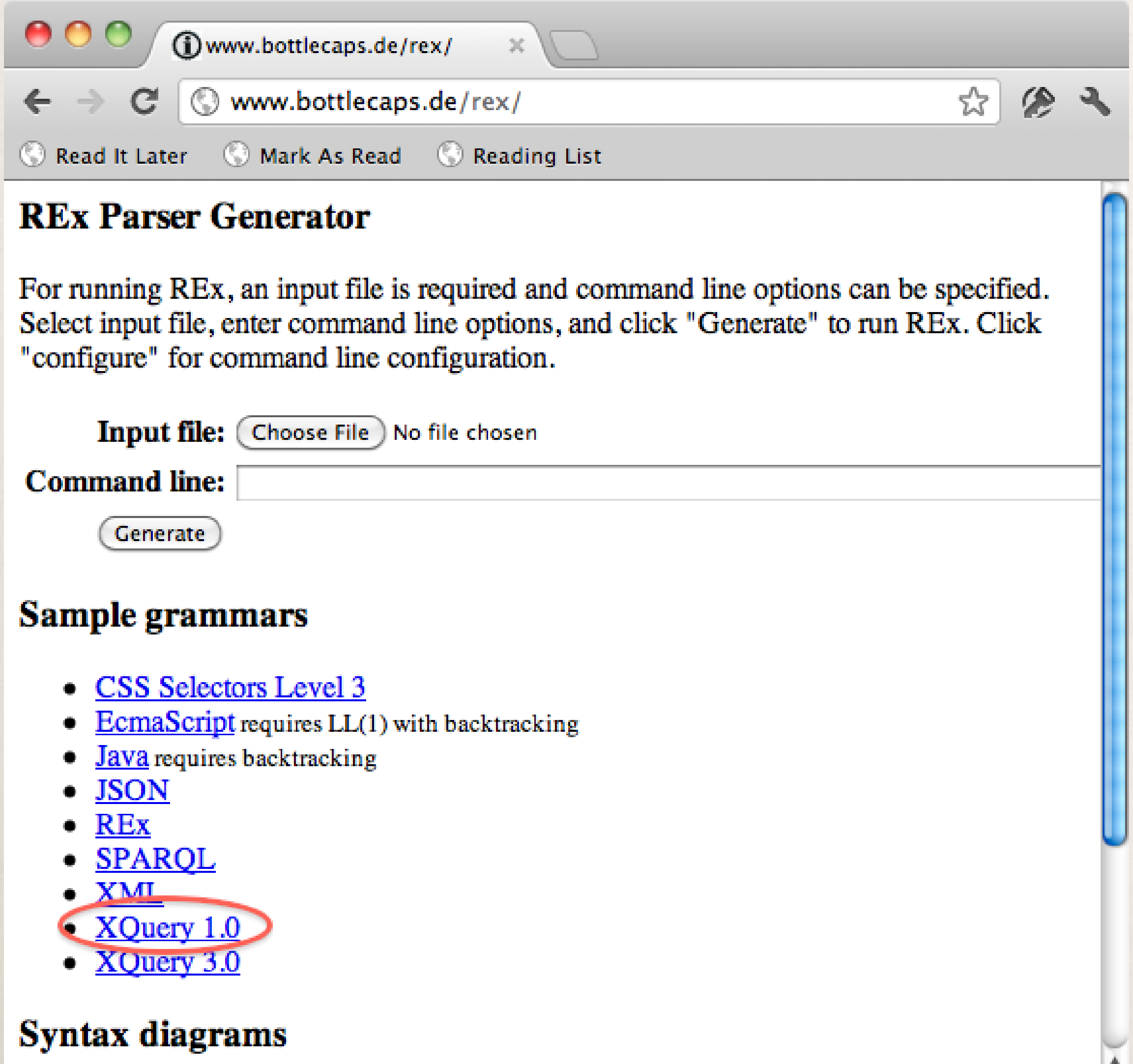
Generate

## Sample grammars

- CSS Selectors Level 3
- EcmaScript requires LL(1) with backtracking
- Java requires backtracking
- JSON
- REx
- SPARQL
- XML
- XQuery 1.0
- XQuery 3.0

## Syntax diagrams

# Step 2: Modify it by hand



## REx Parser Generator

For running REx, an input file is required and command line options can be specified. Select input file, enter command line options, and click "Generate" to run REx. Click "configure" for command line configuration.

**Input file:** Choose File No file chosen

**Command line:**

Generate

## Sample grammars

- CSS Selectors Level 3
- EcmaScript requires LL(1) with backtracking
- Java requires backtracking
- JSON
- REx
- SPARQL
- XML
- XQuery 1.0
- XQuery 3.0

## Syntax diagrams

# Delete what we don’t need

```
XQuery   ::= Module EOF

Module   ::= VersionDecl? ( LibraryModule | MainModule )

VersionDecl

         ::= 'xquery' 'version' StringLiteral ( 'encoding'
StringLiteral )? Separator

MainModule

         ::= Prolog QueryBody

LibraryModule

         ::= ModuleDecl Prolog

/* ……etc……*/
```

# Delete what we don’t need

```
~~XQuery        ::= Module EOF~~

~~Module        ::= VersionDecl? ( LibraryModule | MainModule )~~

~~VersionDecl~~

~~              ::= 'xquery' 'version' StringLiteral ( 'encoding'~~
~~StringLiteral )? Separator~~

~~MainModule~~

~~              ::= Prolog QueryBody~~

~~LibraryModule~~

~~              ::= ModuleDecl Prolog~~

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

Basically, delete everything but Expr and its descendants

# Add the top-level structure

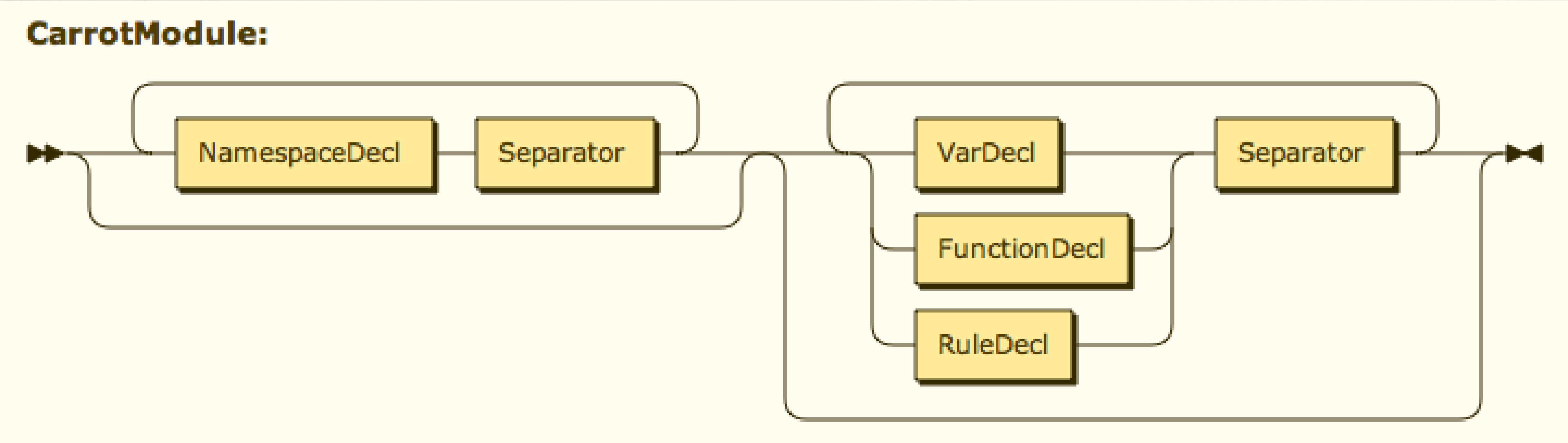
```
CarrotModule ::=
   (NamespaceDecl Separator)*
   ((VarDecl | FunctionDecl | RuleDecl) Separator)*

Separator ::= ';'
```

# Add the top-level structure

```
CarrotModule ::=
  (NamespaceDecl Separator)*
  ((VarDecl | FunctionDecl | RuleDecl) Separator)*

Separator ::= ';'
```

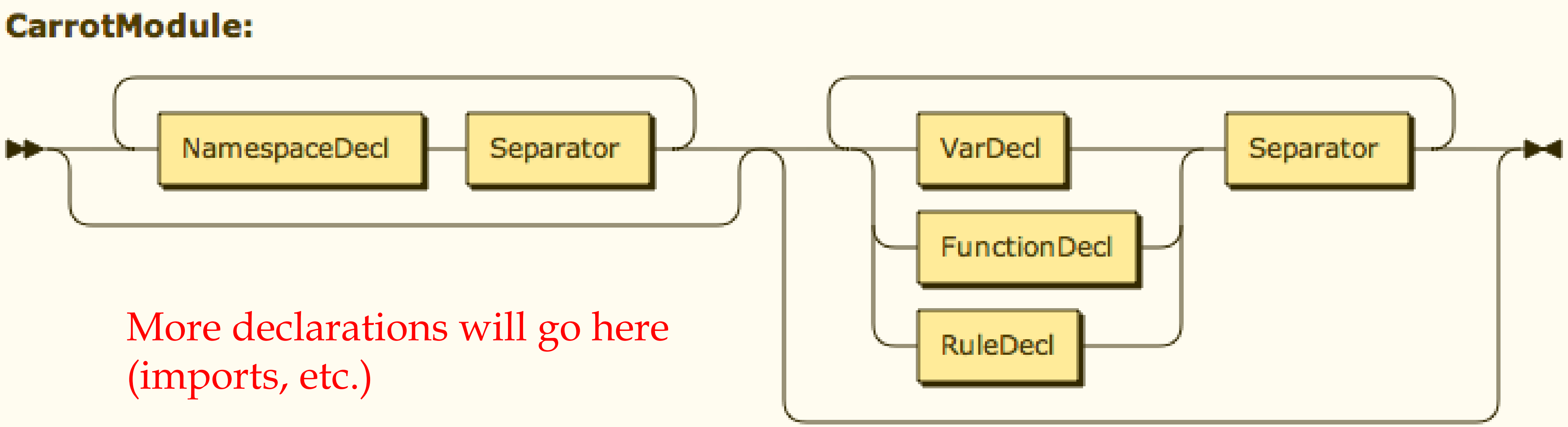


Generated by http://railroad.my28msec.com/rr/ui

# Add the top-level structure

```
CarrotModule ::=
   (NamespaceDecl Separator)*
   ((VarDecl | FunctionDecl | RuleDecl) Separator)*

Separator ::= ';'
```



More declarations will go here (imports, etc.)

Generated by http://railroad.my28msec.com/rr/ui

# Add variable definitions

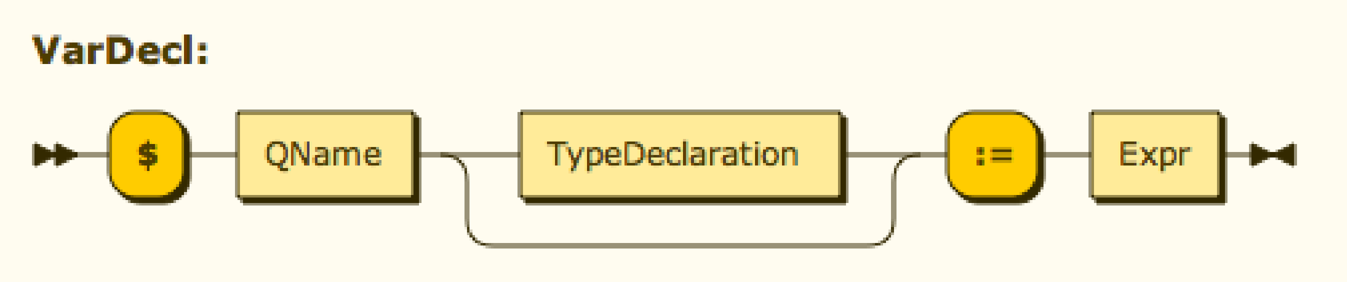
- Similar to XQuery, but without “declare variable”:

```
VarDecl ::= '$' QName TypeDeclaration? ':=' Expr
```

# Add variable definitions

- Similar to XQuery, but without “declare variable”:

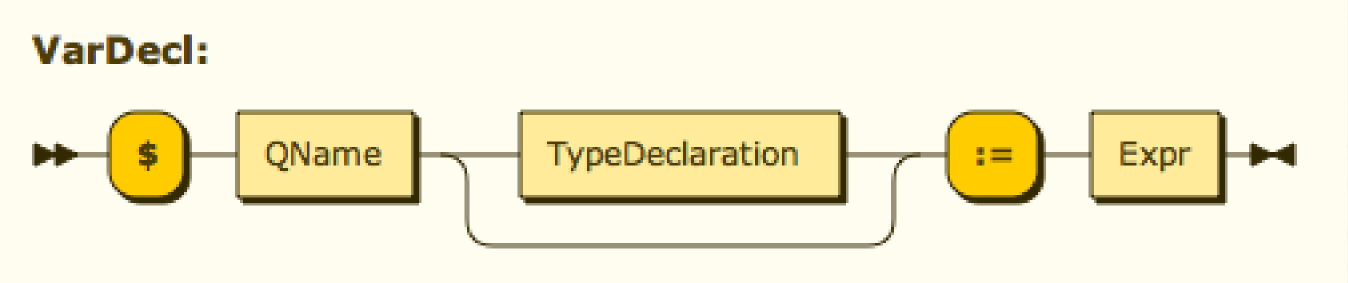
```
VarDecl ::= '$' QName TypeDeclaration? ':=' Expr
```



# Add variable definitions

- Similar to XQuery, but without "declare variable":

```
VarDecl ::= '$' QName TypeDeclaration? ':=' Expr
```

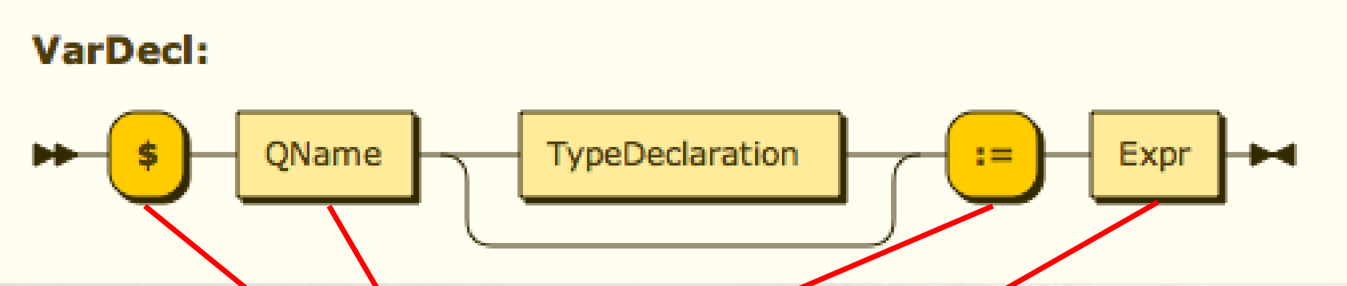


- For example: `$foo := "a string";`

# Add variable definitions

- Similar to XQuery, but without “declare variable”:

```
VarDecl ::= '$' QName TypeDeclaration? ':=' Expr
```



- For example: `$foo := "a string";`

# Add variable definitions

- Similar to XQuery, but without “declare variable”:

```
VarDecl ::= '$' QName TypeDeclaration? ':=' Expr
```

- Example: `$foo := "a string";`

- Eventual parse result:

```
<VarDecl>
  <TOKEN>$</TOKEN>
  <QName>foo</Qname>
  <TOKEN>:=</TOKEN>
  <Expr>
    <StringLiteral>"a string"</StringLiteral>
  </Expr>
</VarDecl>
```

# Add variable definitions

- Similar to XQuery, but without "declare variable":

```
VarDecl ::= '$' QName TypeDeclaration? ':=' Expr
```

- Example: `$foo := "a string";`
- Eventual parse result:

```
<VarDecl>
  <TOKEN>$</TOKEN>
  <QName>foo</Qname>
  <TOKEN>:=</TOKEN>
  <Expr>
    <StringLiteral>"a string"</StringLiteral>
  </Expr>
</VarDecl>
```

# Add variable definitions

- Similar to XQuery, but without “declare variable”:

```
VarDecl ::= '$' QName TypeDeclaration? ':=' Expr
```

- Example: `$foo := "a string";`

- Eventual parse result:

```
<VarDecl>
  <TOKEN>$</TOKEN>
  <QName>foo</QName>
  <TOKEN>:=</TOKEN>
  <Expr>
    <StringLiteral>"a string"</StringLiteral>
  </Expr>
</VarDecl>
```

# Add variable definitions

- Similar to XQuery, but without “declare variable”:

  `VarDecl ::= '$' QName TypeDeclaration? ':='` **`Expr`**

- Example: `$foo := "a string";`

- Eventual parse result:

```
<VarDecl>
  <TOKEN>$</TOKEN>
  <QName>foo</QName>
  <TOKEN>:=</TOKEN>
  <Expr>
    <StringLiteral>"a string"</StringLiteral>
  </Expr>
</VarDecl>
```

# Extend the expression grammar

- To the XQuery “Expr” grammar rule, add these three:

1. ruleset invocations — `^mode(nodes)`
2. shallow `copy{…}` constructors
3. text node literals — `` `my text node` ``

# Add copy constructors

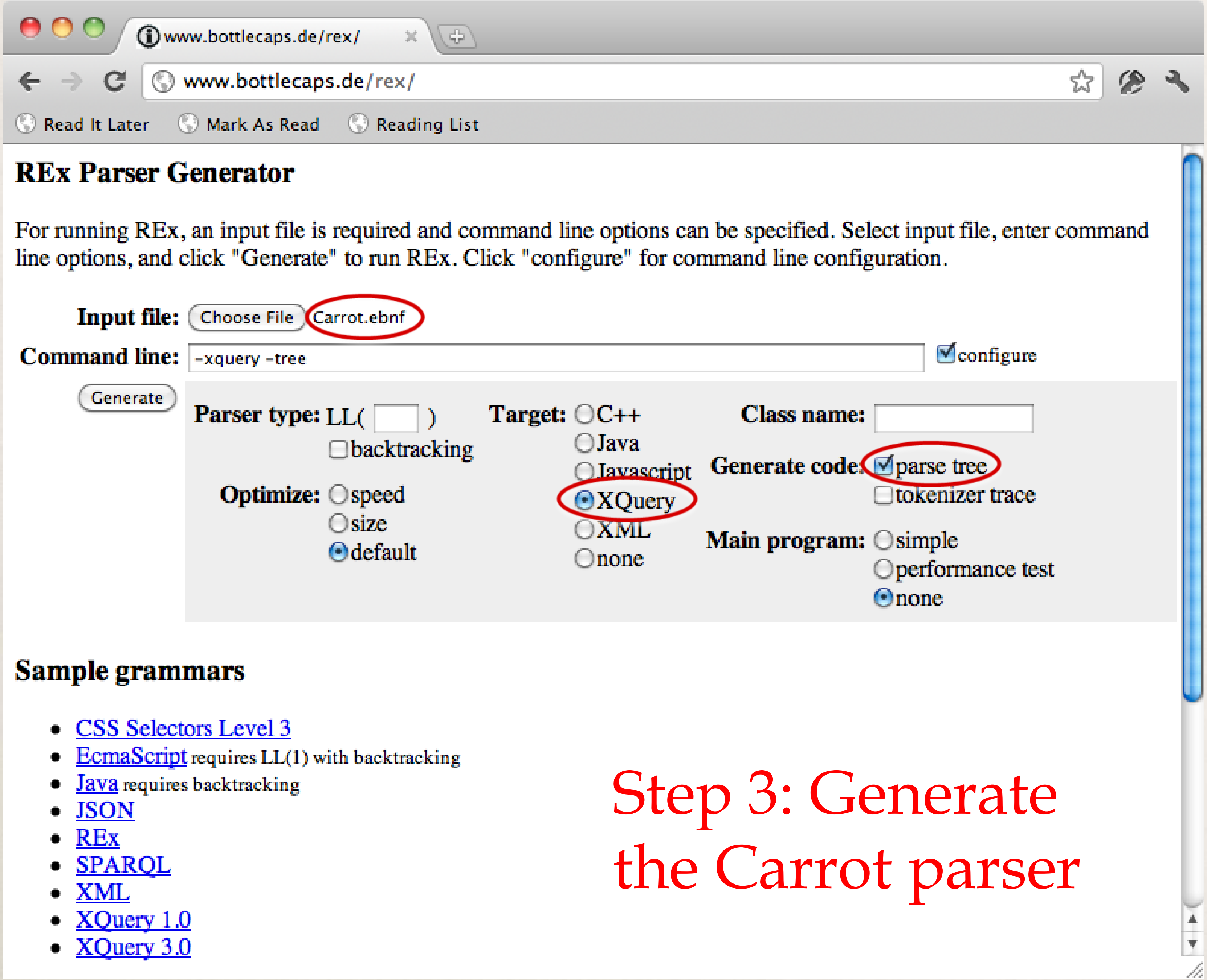
```
CompCopyConstructor ::= 'copy' '{' Expr '}'
```

# Add copy constructors

```
ComputedConstructor ::= CompDocConstructor
                      | CompElemConstructor
                      | CompAttrConstructor
                      | CompTextConstructor
                      | CompCommentConstructor
                      | CompPIConstructor
```

# Add copy constructors

```
ComputedConstructor ::= CompDocConstructor
                      | CompElemConstructor
                      | CompAttrConstructor
                      | CompTextConstructor
                      | CompCommentConstructor
                      | CompPIConstructor
                      | CompCopyConstructor
```



## REx Parser Generator

For running REx, an input file is required and command line options can be specified. Select input file, enter command line options, and click "Generate" to run REx. Click "configure" for command line configuration.

**Input file:** Choose File Carrot.ebnf

**Command line:** -xquery -tree ☑configure

Generate

**Parser type:** LL( ) □backtracking

**Optimize:** ○speed ○size ●default

**Target:** ○C++ ○Java ○Javascript ●XQuery ○XML ○none

**Class name:**

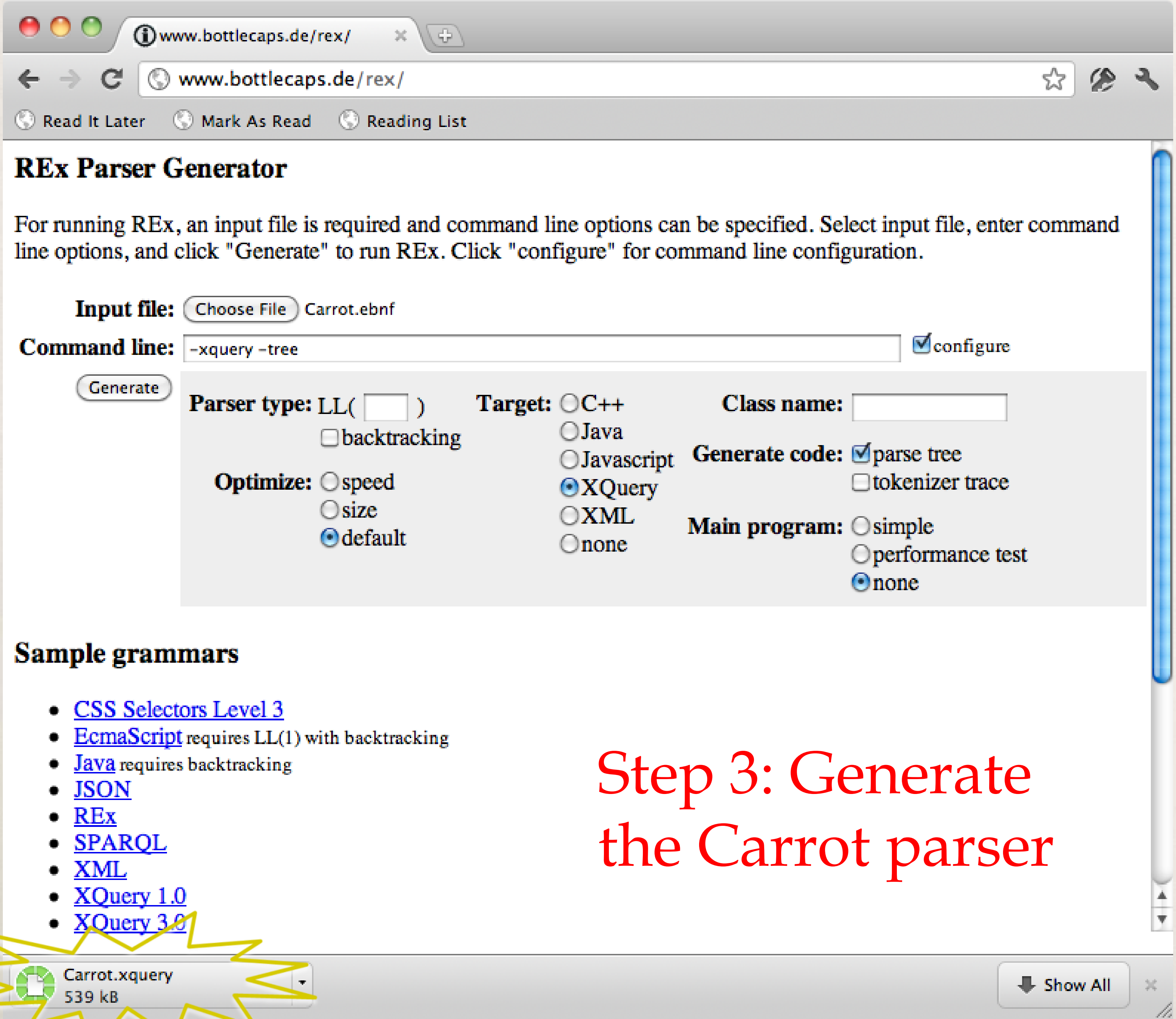
**Generate code:** ☑parse tree □tokenizer trace

**Main program:** ○simple ○performance test ●none

## Sample grammars

- CSS Selectors Level 3
- EcmaScript requires LL(1) with backtracking
- Java requires backtracking
- JSON
- REx
- SPARQL
- XML
- XQuery 1.0
- XQuery 3.0

Step 3: Generate the Carrot parser



# REx Parser Generator

For running REx, an input file is required and command line options can be specified. Select input file, enter command line options, and click "Generate" to run REx. Click "configure" for command line configuration.

**Input file:** Carrot.ebnf

**Command line:** -xquery -tree ☑configure

**Parser type:** LL( ) □backtracking

**Optimize:** ○speed ○size ●default

**Target:** ○C++ ○Java ○Javascript ●XQuery ○XML ○none

**Class name:**

**Generate code:** ☑parse tree □tokenizer trace

**Main program:** ○simple ○performance test ●none

## Sample grammars

- CSS Selectors Level 3
- EcmaScript requires LL(1) with backtracking
- Java requires backtracking
- JSON
- REx
- SPARQL
- XML
- XQuery 1.0
- XQuery 3.0

Carrot.xquery
539 kB

Step 3: Generate the Carrot parser

# Parsing a sample module

Sample module:

```
p:parse-Carrot("
  $names := for $x in //name return lower-case(.);
")
```

# Parsing a sample module

Sample module:

```
p:parse-Carrot("
   $names := for $x in //name return lower-case(.);
")
```

Parsed result:

```
<Carrot><CarrotModule><VarDecl><TOKEN>$</TOKEN><QName><FunctionName><QName>names</QName></FunctionName></QName>
<TOKEN>:=</TOKEN><Expr><ExprSingle><FLWORExpr><ForClause> <TOKEN>for</TOKEN><ForBinding>
<TOKEN>$</TOKEN><VarName><QName><FunctionName><QName>x</QName></FunctionName></QName></VarName>
<TOKEN>in</TOKEN><ExprSingle><OrExpr><AndExpr><ComparisonExpr><RangeExpr><AdditiveExpr>
<MultiplicativeExpr><UnionExpr><IntersectExceptExpr><InstanceofExpr><TreatExpr><CastableExpr><CastExpr><UnaryExpr><ValueExpr><PathExpr>
<TOKEN>//</TOKEN><RelativePathExpr><StepExpr><AxisStep><ForwardStep><AbbrevForwardStep><NodeTest><NameTest><QName><FunctionName>
<QName>names</QName></FunctionName></QName></NameTest></NodeTest></AbbrevForwardStep></ForwardStep><PredicateList/></AxisStep></StepExpr
></RelativePathExpr></PathExpr></ValueExpr></UnaryExpr></CastExpr></CastableExpr></TreatExpr></InstanceofExpr>
</IntersectExceptExpr></UnionExpr></MultiplicativeExpr></AdditiveExpr></RangeExpr></ComparisonExpr></AndExpr></OrExpr></ExprSingle></For
Binding></ForClause> <TOKEN>return</TOKEN><ExprSingle><OrExpr><AndExpr><ComparisonExpr><RangeExpr><AdditiveExpr>
<MultiplicativeExpr><UnionExpr><IntersectExceptExpr><InstanceofExpr><TreatExpr><CastableExpr><CastExpr><UnaryExpr><ValueExpr><PathExpr><
RelativePathExpr><StepExpr><FilterExpr><PrimaryExpr><FunctionCall><FunctionName> <QName>lower-
case</QName></FunctionName><TOKEN>(</TOKEN><ExprSingle><OrExpr><AndExpr><ComparisonExpr><RangeExpr><AdditiveExpr><MultiplicativeExpr><Un
ionExpr><IntersectExceptExpr><InstanceofExpr><TreatExpr><CastableExpr><CastExpr><UnaryExpr><ValueExpr><PathExpr><RelativePathExpr><StepE
xpr><FilterExpr><PrimaryExpr><ContextItemExpr><TOKEN>.</TOKEN></ContextItemExpr></PrimaryExpr><PredicateList/></FilterExpr></StepExpr></
RelativePathExpr></PathExpr></ValueExpr></UnaryExpr></CastExpr></CastableExpr></TreatExpr></InstanceofExpr></IntersectExceptExpr></Union
Expr></MultiplicativeExpr></AdditiveExpr></RangeExpr></ComparisonExpr></AndExpr></OrExpr></ExprSingle><TOKEN>)</TOKEN></FunctionCall></P
rimaryExpr><PredicateList/></FilterExpr></StepExpr></RelativePathExpr></PathExpr></ValueExpr></UnaryExpr></CastExpr></CastableExpr></Tre
atExpr></InstanceofExpr></IntersectExceptExpr></UnionExpr></MultiplicativeExpr></AdditiveExpr></RangeExpr></ComparisonExpr></AndExpr></O
rExpr></ExprSingle></FLWORExpr></ExprSingle></Expr></VarDecl><Separator><TOKEN>;</TOKEN></Separator></CarrotModule><EOF/></Carrot>
```

# Parsing a sample module

Sample module:

```
p:parse-Carrot("
   $names := for $x in //name return lower-case(.);
")
```

Parsed result:

```
<Carrot><CarrotModule><VarDecl><TOKEN>$</TOKEN><QName><FunctionName><QName>names</QName></FunctionName></QName>
<TOKEN>:=</TOKEN><Expr><ExprSingle><FLWORExpr><ForClause> <TOKEN>for</TOKEN><ForBinding>
<TOKEN>$</TOKEN><VarName><QName><FunctionName><QName>x</QName></FunctionName></QName></VarName>
<TOKEN>in</TOKEN><ExprSingle><OrExpr><AndExpr><ComparisonExpr><RangeExpr><AdditiveExpr>
<MultiplicativeExpr><UnionExpr><IntersectExceptExpr><InstanceofExpr><TreatExpr><CastableExpr><CastExpr><UnaryExpr><ValueExpr><PathExpr>
<TOKEN>//</TOKEN><RelativePathExpr><StepExpr><AxisStep><ForwardStep><AbbrevForwardStep><NodeTest><NameTest><QName><FunctionName>
<QName>names</QName></FunctionName></QName></NameTest></NodeTest></AbbrevForwardStep></ForwardStep><PredicateList/></AxisStep></StepExpr></Relati
vePathExpr></PathExpr></ValueExpr></UnaryExpr></CastExpr></CastableExpr></TreatExpr></InstanceofExpr>
</IntersectExceptExpr></UnionExpr></MultiplicativeExpr></AdditiveExpr></RangeExpr></ComparisonExpr></AndExpr></OrExpr></ExprSingle></ForBinding></ForCla
use> <TOKEN>return</TOKEN><ExprSingle><OrExpr><AndExpr><ComparisonExpr><RangeExpr><AdditiveExpr>
<MultiplicativeExpr><UnionExpr><IntersectExceptExpr><InstanceofExpr><TreatExpr><CastableExpr><CastExpr><UnaryExpr><ValueExpr><PathExpr><RelativePathExpr
><StepExpr><FilterExpr><PrimaryExpr><FunctionCall><FunctionName> <QName>lower-
case</QName></FunctionName><TOKEN> (</TOKEN><ExprSingle><OrExpr><AndExpr><ComparisonExpr><RangeExpr><AdditiveExpr><MultiplicativeExpr><UnionExpr><
IntersectExceptExpr><InstanceofExpr><TreatExpr><CastableExpr><CastExpr><UnaryExpr><ValueExpr><PathExpr><RelativePathExpr><StepExpr><FilterExpr><PrimaryE
xpr><ContextItemExpr><TOKEN> . </TOKEN></ContextItemExpr></PrimaryExpr><PredicateList/></FilterExpr></StepExpr></RelativePathExpr></PathExpr></ValueExpr
></UnaryExpr></CastExpr></CastableExpr></TreatExpr></InstanceofExpr></IntersectExceptExpr></UnionExpr></MultiplicativeExpr></AdditiveExpr></RangeExpr></
ComparisonExpr></AndExpr></OrExpr></ExprSingle><TOKEN>) </TOKEN></FunctionCall></PrimaryExpr><PredicateList/></FilterExpr></StepExpr></RelativePathExpr
></PathExpr></ValueExpr></UnaryExpr></CastExpr></CastableExpr></TreatExpr></InstanceofExpr></IntersectExceptExpr></UnionExpr></MultiplicativeExpr></Addi
tiveExpr></RangeExpr></ComparisonExpr></AndExpr></OrExpr></ExprSingle></FLWORExpr></ExprSingle></Expr></VarDecl><Separator><TOKEN> ; </TOKEN></Separator
></CarrotModule><EOF/></Carrot>
```

# Parsing a sample module

Sample module:

```
p:parse-Carrot("
  $names := for $x in //name return lower-case(.);
")
```

Parsed result:

```
                 $                **names**
   :=                          for
   $                    x
   in
   //
   names

     return
                                lower-
case              (
             .
                        )
                                                           ;
```

# Parsing a sample module

Sample module:

```
p:parse-Carrot("
  $names := for $x in //name return lower-case(.);
")
```

Parsed result (string value):

```
  $names := for $x in //name return lower-case(.);
```

# Parsing a sample module

Sample module:

```
p:parse-Carrot("
  $names := for $x in //name return lower-case(.);
")
```

Parsed result (string value):

```
  $names := for $x in //name return lower-case(.);
```

Compilation target:

```
<xsl:stylesheet …>
  <xsl:variable name="names"
                select="for $x in //name return lower-case(.)"/>
</xsl:stylesheet>
```

# Compilation is thus trivial for XPath 2.0 expressions

Just get the string value of the expression parse tree:

```
<xsl:value-of select="Expr"/>
```

# Compilation is thus trivial for XPath 2.0 expressions

Just get the string value of the expression parse tree:

```
<xsl:value-of select="Expr"/>
```

But not every XQuery expression is valid XPath 2.0…

The trick is identifying which ones are and which ones aren’t.

# Compiling Carrot

# The task at hand

Convert the XML parse tree into an XSLT stylesheet.

# Three phases of compilation

1. Simplify the parse tree
2. Annotate the expressions
3. Generate the XSLT

# A lot of noise to deal with

Sample module:

```
p:parse-Carrot("
   $names := for $x in //name return lower-case(.);
")
```

Parsed result:

```
<Carrot><CarrotModule><VarDecl><TOKEN>$</TOKEN><QName><FunctionName><QName>names</QName></FunctionName></QName>
<TOKEN>:=</TOKEN><Expr><ExprSingle><FLWORExpr><ForClause> <TOKEN>for</TOKEN><ForBinding>
<TOKEN>$</TOKEN><VarName><QName><FunctionName><QName>x</QName></FunctionName></QName></VarName>
<TOKEN>in</TOKEN><ExprSingle><OrExpr><AndExpr><ComparisonExpr><RangeExpr><AdditiveExpr>
<MultiplicativeExpr><UnionExpr><IntersectExceptExpr><InstanceofExpr><TreatExpr><CastableExpr><CastExpr><UnaryExpr><ValueExpr><PathExpr>
<TOKEN>// </TOKEN><RelativePathExpr><StepExpr><AxisStep><ForwardStep><AbbrevForwardStep><NodeTest><NameTest><QName><FunctionName>
<QName>names</QName></FunctionName></QName></NameTest></NodeTest></AbbrevForwardStep></ForwardStep><PredicateList/></AxisStep></StepExpr></Relati
vePathExpr></PathExpr></ValueExpr></UnaryExpr></CastExpr></CastableExpr></TreatExpr></InstanceofExpr>
</IntersectExceptExpr></UnionExpr></MultiplicativeExpr></AdditiveExpr></RangeExpr></ComparisonExpr></AndExpr></OrExpr></ExprSingle></ForBinding></ForCla
use> <TOKEN>return</TOKEN><ExprSingle><OrExpr><AndExpr><ComparisonExpr><RangeExpr><AdditiveExpr>
<MultiplicativeExpr><UnionExpr><IntersectExceptExpr><InstanceofExpr><TreatExpr><CastableExpr><CastExpr><UnaryExpr><ValueExpr><PathExpr><RelativePathExpr
><StepExpr><FilterExpr><PrimaryExpr><FunctionCall><FunctionName> <QName>lower-
case</QName></FunctionName><TOKEN> (</TOKEN><ExprSingle><OrExpr><AndExpr><ComparisonExpr><RangeExpr><AdditiveExpr><MultiplicativeExpr><UnionExpr><
IntersectExceptExpr><InstanceofExpr><TreatExpr><CastableExpr><CastExpr><UnaryExpr><ValueExpr><PathExpr><RelativePathExpr><StepExpr><FilterExpr><PrimaryE
xpr><ContextItemExpr><TOKEN> . </TOKEN></ContextItemExpr></PrimaryExpr><PredicateList/></FilterExpr></StepExpr></RelativePathExpr></PathExpr></ValueExpr
></UnaryExpr></CastExpr></CastableExpr></TreatExpr></InstanceofExpr></IntersectExceptExpr></UnionExpr></MultiplicativeExpr></AdditiveExpr></RangeExpr></
ComparisonExpr></AndExpr></OrExpr></ExprSingle><TOKEN>) </TOKEN></FunctionCall></PrimaryExpr><PredicateList/></FilterExpr></StepExpr></RelativePathExpr
></PathExpr></ValueExpr></UnaryExpr></CastExpr></CastableExpr></TreatExpr></InstanceofExpr></IntersectExceptExpr></UnionExpr></MultiplicativeExpr></Addi
tiveExpr></RangeExpr></ComparisonExpr></AndExpr></OrExpr></ExprSingle></FLWORExpr></ExprSingle></Expr></VarDecl><Separator><TOKEN> ; </TOKEN></Separator
></CarrotModule><EOF/></Carrot>
```

# Phase 1: Simplify the tree (demo)

# Mapping Carrot to XSLT

| In Carrot | Translated to |
|---|---|
| Rule definition | <xsl:template> |
| Variable definition | <xsl:variable> |
| Function definition | <xsl:function> |
| XPath expression | XPath expression |
| non-XPath expression | ??? |

# Example non-XPath mappings

| In Carrot | Translated to |
|---|---|
| Element constructors | Literal result elements |
| FLWOR expressions | <xsl:for-each>, <xsl:variable>, <xsl:if>, <xsl:sort>* |
| text{} | <xsl:value-of> |
| Etc. | |

* However, see http://www.xmlprague.cz/2006/slides06/carlisle/dpc-prague2006-18.html

# Mismatch

- In XQuery and Carrot, only one syntactic context:
  - Expressions
- In XSLT, two syntactic contexts:
  - Sequence constructors
  - Expressions
- Non-XPath expressions need to be converted to sequence constructors
  - Composed using auto-generated helper functions

# Phase 2: Annotate the tree (demo)

# Phase 3: Generate the XSLT (demo)

# Project goals

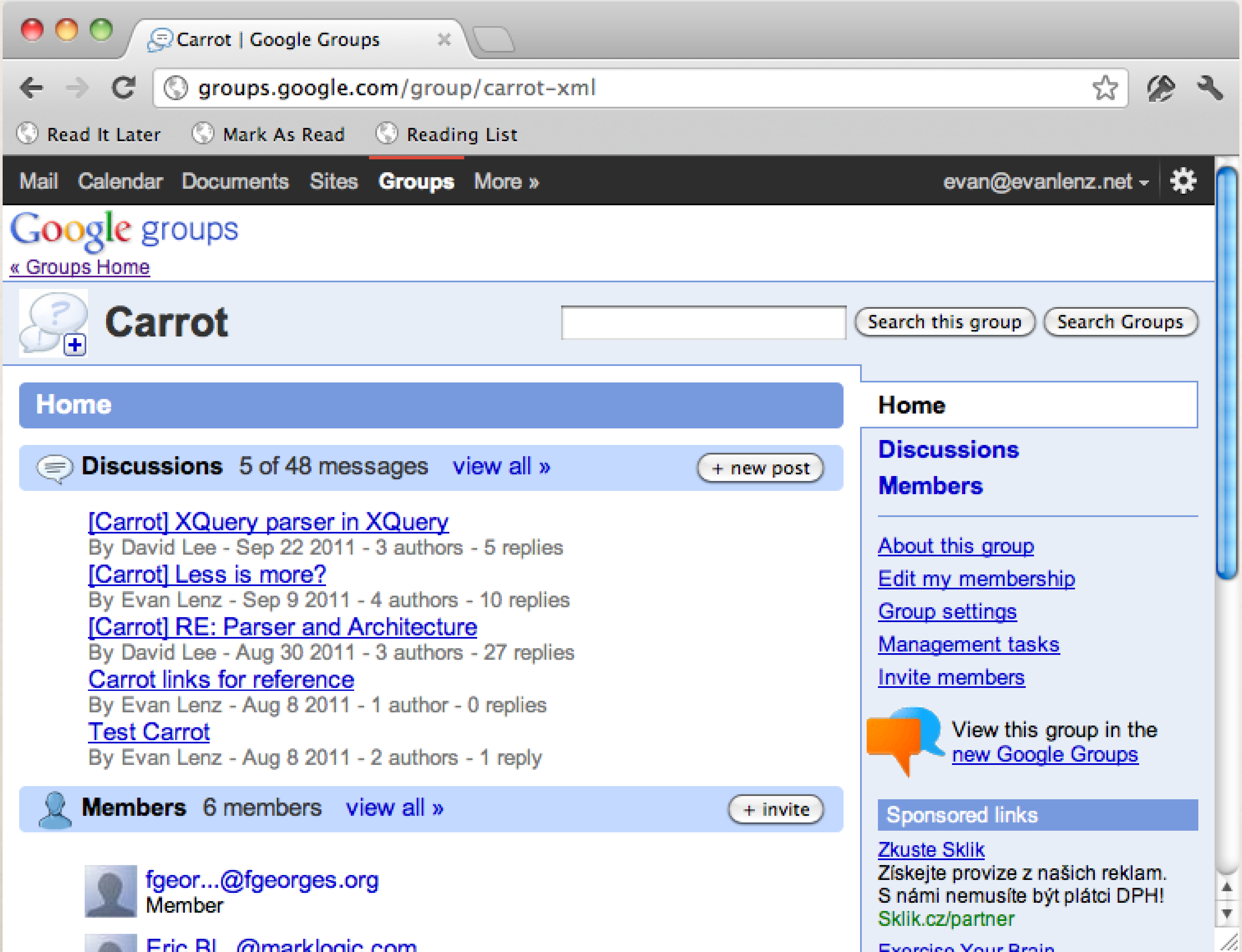
- Create implementations for Saxon and MarkLogic
  - Create a library for MarkLogic
  - Implement caching of compiled stylesheets
- Solicit feedback and participation

# Related projects

- Compilation to XSLT
  - http://monet.nag.co.uk/xq2xml/
    - http://web.archive.org/web/20080926043959/http://monet.nag.co.uk/xq2xml/
  - http://www.xmlprague.cz/2006/slides06/carlisle/dpc-prague2006-01.html
- Compilation to XQuery
  - “Compiling XSLT 2.0 into XQuery 1.0” http://www2005.org/cdrom/docs/p682.pdf

# Get involved

- Join the Google Group
  - http://groups.google.com/group/carrot-xml
- Watch or fork the GitHub project
  - http://github.com/evanlenz/carrot
- Share your ideas
- Cheerlead, prod, or provoke



# Carrot

## Home

**Discussions** 5 of 48 messages view all »

[Carrot] XQuery parser in XQuery
By David Lee - Sep 22 2011 - 3 authors - 5 replies

[Carrot] Less is more?
By Evan Lenz - Sep 9 2011 - 4 authors - 10 replies

[Carrot] RE: Parser and Architecture
By David Lee - Aug 30 2011 - 3 authors - 27 replies

Carrot links for reference
By Evan Lenz - Aug 8 2011 - 1 author - 0 replies

Test Carrot
By Evan Lenz - Aug 8 2011 - 2 authors - 1 reply

**Members** 6 members view all »

fgeor...@fgeorges.org
Member

Eric Bl...@marklogic.com

View this group in the new Google Groups

Questions?