XML Technologies
Parsing XML & XQuery

Christian Grün
Database & Information Systems Group
Universität Konstanz

Introduction

State of the art

- we have learned about XML, XPath and XQuery from a *user perspective*

Next steps

- we will now look at how XML and XQuery input is *processed*
- in the art of *compiler construction*, typical processing phases are: *parsing* (analysis), *compilation* (optimization), *evaluation, serialization*
- please note that the used terminology *varies a lot*

✎ What does *processing* in the context of a) XML and b) XQuery means?

Introduction

Compilation

Making human-readable input better processable for a CPU

Classical steps

1. lexical analysis
2. syntax analysis
3. semantic analysis
4. optimization
5. code generation

XML processing

– scanning
– parsing

XPath/XQuery processing

1. static analysis
   – scanning
   – parsing
   – compilation
2. dynamic evaluation

Static Analysis

Introduction

- on low level, XML and query inputs are simple *streams of characters*
- characters have to *interpreted* and *transformed* into a representation that can be understood and evaluated by the processing code

Single steps

- *scanning*: turns characters into atomic units (so-called *lexical tokens*)
- *parsing*: checks tokens against a *grammar*, creates an *Abstract Syntax Tree*
- *compilation*: performs validity and type checks, *optimizes* the tree

☞ first two steps can be combined (see e.g. JavaCC)

Static Analysis: Scanning

Lexical tokens

(Some of the) characters handled by an XML scanner:

Token	Examples	Description
L_BR	<	left bracket for opening a tag
R_BR	>	right bracket
L_BR_CLOSE	</	closing left bracket
NAME	html div bgcolor	element or attribute name
EQ	=	equal sign
QUOTE	"	quote
ATT_VAL	#FFFFFF right	attribute value
SPACE	_ \t \r \n	delimiters (white spaces)
TEXT	Databases	element content

Static Analysis: Scanning

Character stream

```
<html>
  <head>
    <title>XML</title>
  </head>
  <body bgcolor="#FFFFFF">
    <h1>Databases</h1>
    <div align="right">
      <b>Assignments</b>
      <ul>
        <li>Exercise 1</li>
        <li>Exercise 2</li>
      </ul>
    </div>
  </body>
</html>
```

Token representation

```
L_BR NAME R_BR
  L_BR NAME SPACE R_BR
    L_BR NAME R_BR TEXT L_BR_CLOSE NAME R_BR
  L_BR_CLOSE NAME R_BR
  L_BR NAME SPACE NAME EQ QUOTE ATT_VAL QUOTE R_BR
    L_BR NAME R_BR TEXT L_BR_CLOSE NAME R_BR
    L_BR NAME SPACE NAME EQ QUOTE ATT_VAL QUOTE R_BR
      L_BR NAME R_BR TEXT L_BR NAME R_BR
      L_BR NAME R_BR
        L_BR NAME R_BR TEXT L_BR_CLOSE NAME R_BR
        L_BR NAME R_BR TEXT L_BR_CLOSE NAME R_BR
      L_BR_CLOSE NAME R_BR
    L_BR_CLOSE NAME R_BR
  L_BR_CLOSE NAME R_BR
L_BR_CLOSE NAME R_BR
```

Static Analysis: Scanning

Simple scanner implementation

- read input character by character
- interpret character and return found token
- raise error if the input is unexpected
- return END token if all characters have been parsed

☞ works fine... for single characters

```
global input-stream input
if input.more():
  char ch ← input.next()
  if ch = '<':
    return L_BR
  else if ch = '>':
    return R_BR
  else if ch = '=':
    return EQ
  ...
  else:
    raise error
return END
```

Static Analysis: Scanning

Open issues

- many tokens comprise *more than one character*
  Example: `<xml id="id0">今日は</xml>`
- character sequences can be *ambiguous*:
  1. `</x>` could be split up as `L_BR TEXT R_BR` or `L_BR_CLOSING NAME R_BR`
  2. a space can be part of an element definition or a text node (`SPACE` vs `TEXT`)

Solutions

1. use *greedy* scanning: check for tokens with maximum length first
2. introduce *states*: interprete input differently, depending on context

Static Analysis: Scanning

Regular expressions

The syntax of tokens can be defined via regular expressions:

Constructor	Examples	Description
a	"a"	single character
[0123]	"0", "1", "2" or "3"	class of characters
[^0123]	everything but "0", "1", ...	exclusion of characters
abc|cba	"abc" or "cba"	alternation
a?	"", "a"	zero or one character
a*	"", "a", "aa", "aaa", ...	zero or more characters
a+	"a", "aa", "aaa", ...	one or more characters
(ab|ba)*	"", "ab", "ba", "abba", ...	grouping

Static Analysis: Scanning

Lexical tokens: States and Constructors

State	Token	Constructor	Description	
CONTENT	TEXT	[^<]+	content	
CONTENT	L_BR_CLOSE	</	closing tag	→ ELEMENT state
CONTENT	L_BR	<	opening tag	→ ELEMENT state
ELEMENT	NAME	[A-Za-z][A-Za-z0-9]+	element/attribute name	
ELEMENT	SPACE	[\t\r\n]+	white spaces	
ELEMENT	EQ	=	equal sign	
ELEMENT	R_BR	>	ending tag	→ CONTENT state
ELEMENT	CLOSE_R_BR	/>	empty tag	→ CONTENT state
ELEMENT	QUOTE	"	double quote	→ QUOTED state
QUOTED	ATT_VAL	[^"]+	attribute value	

Initial state: CONTENT

Static Analysis: Scanning

Simple scanner implementation: greedy scanning

```
global string input
global state state
global int s, e

s = e
e = length(input)
while --e > 0:
  string content ← substr(input,s,e)
  if state = CONTENT:
    if content matches '[^<]+':
      return TEXT and content
    if content matches '</':
      state ← ELEMENT
      return L_BR_CLOSE
    ...
  if state = ELEMENT:
    if content matches '[A-Za-z][A-Za-z0-9]*':
      return NAME and content
    if content matches '"':
      state ← QUOTED
      return QUOTE
    ...
  if state = QUOTED:
    if content matches '[^"]+':
      return ATT_VAL and content
    if content matches '"':
      state ← ELEMENT
      return QUOTE

otherwise, raise error
```

Static Analysis: Scanning

Advantages

- token rules are *simple to define* and *easy to implement*

Drawback

- greedy approach is *expensive*, because token can stretch over multiple lines (worst case: up to end of file)

Alternative

- *non-greedy* scanning: search for the shortest (yet complete) token
- only scan more characters if token is *ambiguous*

Static Analysis: Scanning

Type 1: simple, unique tokens

- no overlap with other tokens of the same state:
  EQ (=), QUOTE ("), R_BR (>), CLOSE_R_BR (/>)

Type 2: ambiguous tokens

- scan more characters to uniquely determine token type:
  L_BR_CLOSE (</), L_BR (<)

Type 3: tokens with varying character contents

- scan all characters that match the token type:
  TEXT, NAME, ATT_VAL, SPACE

Static Analysis: Scanning

Type 1-3: examples

Type 1

```
# state: ELEMENT
if ch = '=':
  return EQ
if ch = '/':
  ch ← next char
  if ch = '>':
    state ← CONTENT
    return CLOSE_R_BR
  else:
    raise error
```

Type 2

```
# state: CONTENT
if ch = '<':
  state = ELEMENT
  ch ← next char
  if ch = '/':
    return L_BR_CLOSE
  else:
    jump one char back
    return L_BR
```

Type 3

```
# state: CONTENT
if ch != '<':
  string content ← ''
  while ch != '<':
    append ch to content
    ch ← next char
  jump one char back
  return TEXT and content
```

Static Analysis: Parsing

Grammars (time for some more TLAs...)

- a parser *retrieves* the tokens and matches them against a *grammar*
- most parsers are based on a *context free grammar* (CFG)

✎ What does "context free" mean?

- an *(E)BNF* grammar defines production rules: `lhs ::= rhs`
- examples:
```
element   ::= startTag content endTag
startTag  ::= L_BR NAME R_BR
```
- *extra-grammatical constraints* cannot be specified by the grammar itself

✎ Do you remember what Well-Formedness Constraints are?

Static Analysis: Parsing

Recursive-descent parser

- in a *recursive-descent* parser, production rules are mapped to *functions*:
  - the function is named like the left-hand side of the rule (`lhs`)
  - the function contains calls to the symbols of the right-hand side (`rhs`)
- tokens (*terminals*, in upper case) are "consumed":

```
element() → void:
  startTag()
  content()
  endTag()

# element ::=
  startTag content endTag
```

```
# startTag ::= L_BR NAME R_BR
startTag() → void:
  consume(L_BR)
  consume(NAME)
  consume(R_BR)
```

```
consume(token token) → void:
  # retrieve next token
  if token != scanner.next():
    raise error
```

Static Analysis: Parsing

Mini-XML grammar

For the sake of simplicity, our grammar uses no wildcards:

```
document      → element
element       → startTag content endTag | emptyTag
startTag      → L_BR NAME attributeList space R_BR
attributeList → ε | SPACE attribute attributeList
attribute     → NAME space EQ space QUOTE ATT_VAL QUOTE
endTag        → L_BR_CLOSE NAME space R_BR
emptyTag      → L_BR NAME attributeList space CLOSE_R_BR
content       → ε | TEXT content | element content
space         → ε | SPACE
```

✎ Remember: What's the difference between `space` and `SPACE`?

Static Analysis: Parsing

Grammar rewritings

- if we have alternations in a grammar (|), which rule is the one to follow?

☞ check next token (perform *one look-ahead*):

Rule:

```
space      → ε | SPACE
```

Rules using space:

```
startTag   → … space R_BR
attribute  → … space EQ …
endTag     → … space CLOSE_R_BR
emptyTag   → … space R_BR
```

```
void space():
  switch scanner.getToken():
    case SPACE:
      consume(SPACE)
      break
    case R_BR,EQ,CLOSE_R_BR,R_BR:
      break
    default:
      raise error
```

Static Analysis: Parsing

Look-ahead: challenges

- the next token must be *uniquely assignable* to one production rule
- otherwise, a conflict occurs (✎ can you spot it?):

```
element   →  startTag content endTag | emptyTag
startTag  →  L_BR NAME attributeList space R_BR
emptyTag  →  L_BR NAME attributeList space CLOSE_R_BR
```

Solution

- rewrite grammar to be LL(1) conform:
  (L)eft-to-right parsing, (L)eftmost derivation, (1) symbol look-ahead

☞ The good thing is: most X... grammars are already expressed in LL(1).

Static Analysis: Parsing

Examples for unambiguous look-ahead rules

```
OLD:   element        →  startTag content endTag | emptyTag
       startTag       →  L_BR NAME attributeList space R_BR
       emptyTag       →  L_BR NAME attributeList space CLOSE_R_BR

NEW:   element        →  element1 element2
       element1       →  L_BR NAME attributeList space
       element2       →  R_BR content endTag | CLOSE_R_BR

OLD:   attributeList  →  ε | SPACE attribute attributeList

NEW:   attributeList1 →  ε | SPACE attributeList2
       attributeList2 →  ε | attribute attributeList1
```

Static Analysis: Parsing

Simplified LL(1) XML grammar

```
document        →  element
element         →  element1 element2
element1        →  L_BR NAME attributeList1
element2        →  R_BR content endTag | CLOSE_R_BR
attributeList1  →  ε | SPACE attributeList2
attributeList2  →  ε | attribute attributeList1
attribute       →  NAME space EQ space QUOTE ATT_VAL QUOTE
endTag          →  L_BR_CLOSE NAME space R_BR
content         →  ε | TEXT content | element content
space           →  ε | SPACE
```

Static Analysis: Parsing

What about XPath/XQuery?

LL(1) grammar for XQuery is much more complex:

- XQuery 1.0 grammar (excluding Full Text/Update) contains ~160 rules
- scanner has to differ between *28 different states*
- however, the *basic approach* for parsing XML or XQuery is similar

```
[1]  Module ::= VersionDecl? (LibraryModule | MainModule)
[2]  VersionDecl ::= "xquery" "version" StringLiteral ("encoding" StringLiteral)? Separator
[3]  MainModule ::= Prolog QueryBody
...
[30] QueryBody ::= Expr
[31] Expr ::= ExprSingle ("," ExprSingle)*
[32] ExprSingle ::= FLWORExpr | QuantifiedExpr | TypeswitchExpr | IfExpr | OrExpr
[33] FLWORExpr ::= (ForClause | LetClause)+ WhereClause? OrderByClause? "return" ExprSingle
[34] ForClause ::= "for" "$" VarName TypeDeclaration? PositionalVar? "in" ExprSingle ("," "$"
       VarName TypeDeclaration? PositionalVar? "in" ExprSingle)*
[35] PositionalVar ::= "at" "$" VarName
[36] LetClause ::= "let" "$" VarName TypeDeclaration? ":=" ExprSingle ("," "$" VarName
       TypeDeclaration? ":=" ExprSingle)*
[37] WhereClause ::= "where" ExprSingle
[38] OrderByClause ::= (("order" "by") | ("stable" "order" "by")) OrderSpecList
[39] OrderSpecList ::= OrderSpec ("," OrderSpec)*
[40] OrderSpec ::= ExprSingle OrderModifier
[41] OrderModifier ::= ("ascending" | "descending")? ("empty" ("greatest" | "least"))?
       ("collation" URILiteral)?
[42] QuantifiedExpr ::= ("some" | "every") "$" VarName TypeDeclaration? "in" ExprSingle ("," 
       "$" VarName TypeDeclaration? "in" ExprSingle)* "satisfies" ExprSingle
[43] TypeswitchExpr ::= "typeswitch" "(" Expr ")" CaseClause+ "default" ("$" VarName)?
       "return" ExprSingle
[44] CaseClause ::= "case" ("$" VarName "as")? SequenceType "return" ExprSingle
[45] IfExpr ::= "if" "(" Expr ")" "then" ExprSingle "else" ExprSingle
[46] OrExpr ::= AndExpr ( "or" AndExpr )*
[47] AndExpr ::= ComparisonExpr ( "and" ComparisonExpr )*
[48] ComparisonExpr ::= RangeExpr ( (ValueComp | GeneralComp | NodeComp) RangeExpr )?
[49] RangeExpr ::= AdditiveExpr ( "to" AdditiveExpr )?
[50] AdditiveExpr ::= MultiplicativeExpr ( ("+" | "-") MultiplicativeExpr )*
[51] MultiplicativeExpr ::= UnionExpr ( ("*" | "div" | "idiv" | "mod") UnionExpr )*
[52] UnionExpr ::= IntersectExceptExpr ( ("union" | "|") IntersectExceptExpr )*
[53] IntersectExceptExpr ::= InstanceofExpr ( ("intersect" | "except") InstanceofExpr )*
[54] InstanceofExpr ::= TreatExpr ( "instance" "of" SequenceType )?
...
```

Static Analysis: Parsing

AST: abstract syntax tree

- tokens are converted to a *tree representation*
- the AST contains all info required for evaluating the expression
- synonyms:
  - *expression tree* (code perspective)
  - *query plan* (database perspective)

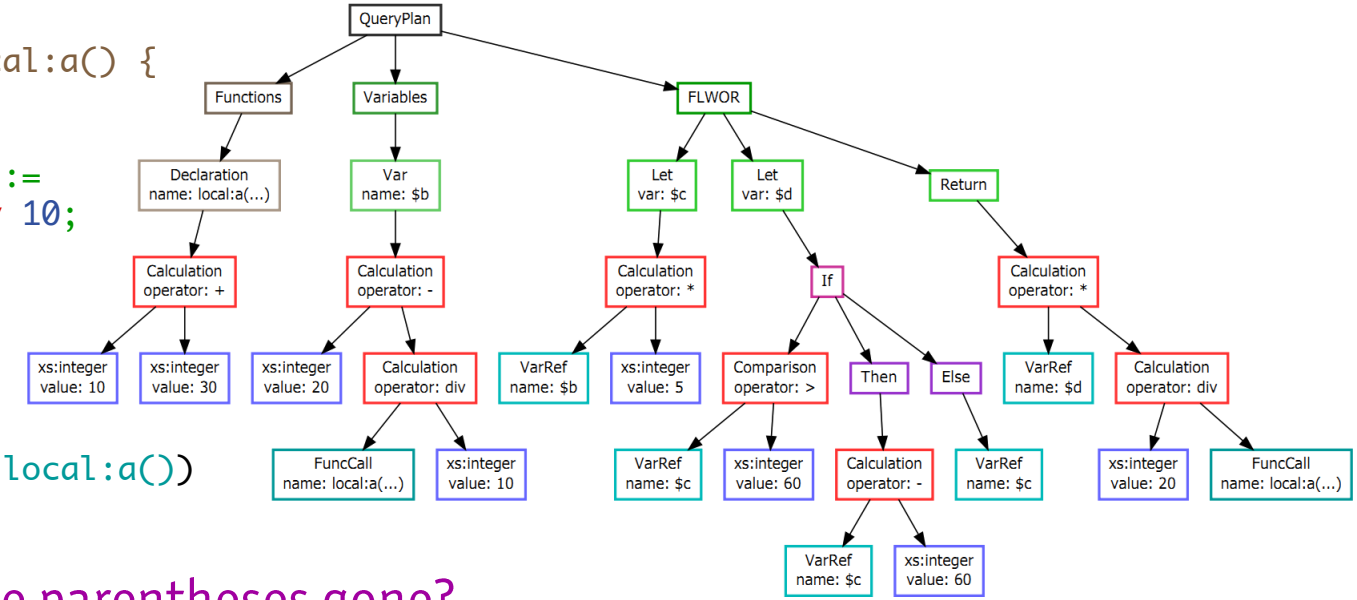
✎ Can XML structures also be represented in ASTs?

```
[1]  Module ::= VersionDecl? (LibraryModule | MainModule)
[2]  VersionDecl ::= "xquery" "version" StringLiteral ("encoding" StringLiteral)? Separator
[3]  MainModule ::= Prolog QueryBody
...
[30] QueryBody ::= Expr
[31] Expr ::= ExprSingle ("," ExprSingle)*
[32] ExprSingle ::= FLWORExpr | QuantifiedExpr | TypeswitchExpr | IfExpr | OrExpr
[33] FLWORExpr ::= (ForClause | LetClause)+ WhereClause? OrderByClause? "return" ExprSingle
[34] ForClause ::= "for" "$" VarName TypeDeclaration? PositionalVar? "in" ExprSingle ("," "$"
       VarName TypeDeclaration? PositionalVar? "in" ExprSingle)*
[35] PositionalVar ::= "at" "$" VarName
[36] LetClause ::= "let" "$" VarName TypeDeclaration? ":=" ExprSingle ("," "$" VarName
       TypeDeclaration? ":=" ExprSingle)*
[37] WhereClause ::= "where" ExprSingle
[38] OrderByClause ::= (("order" "by") | ("stable" "order" "by")) OrderSpecList
[39] OrderSpecList ::= OrderSpec ("," OrderSpec)*
[40] OrderSpec ::= ExprSingle OrderModifier
[41] OrderModifier ::= ("ascending" | "descending")? ("empty" ("greatest" | "least"))?
       ("collation" URILiteral)?
[42] QuantifiedExpr ::= ("some" | "every") "$" VarName TypeDeclaration? "in" ExprSingle ("," 
       "$" VarName TypeDeclaration? "in" ExprSingle)* "satisfies" ExprSingle
[43] TypeswitchExpr ::= "typeswitch" "(" Expr ")" CaseClause+ "default" ("$" VarName)?
       "return" ExprSingle
[44] CaseClause ::= "case" ("$" VarName "as")? SequenceType "return" ExprSingle
[45] IfExpr ::= "if" "(" Expr ")" "then" ExprSingle "else" ExprSingle
[46] OrExpr ::= AndExpr ( "or" AndExpr )*
[47] AndExpr ::= ComparisonExpr ( "and" ComparisonExpr )*
[48] ComparisonExpr ::= RangeExpr ( (ValueComp | GeneralComp | NodeComp) RangeExpr )?
[49] RangeExpr ::= AdditiveExpr ( "to" AdditiveExpr )?
[50] AdditiveExpr ::= MultiplicativeExpr ( ("+" | "-") MultiplicativeExpr )*
[51] MultiplicativeExpr ::= UnionExpr ( ("*" | "div" | "idiv" | "mod") UnionExpr )*
[52] UnionExpr ::= IntersectExceptExpr ( ("union" | "|") IntersectExceptExpr )*
[53] IntersectExceptExpr ::= InstanceofExpr ( ("intersect" | "except") InstanceofExpr )*
[54] InstanceofExpr ::= TreatExpr ( "instance" "of" SequenceType )?
...
```

Static Analysis: Parsing

Abstract syntax tree: BaseX

```
declare function local:a() {
  10 + 30
};
declare variable $b :=
  20 - local:a() div 10;

let $c := $b * 5
let $d :=
  if($c > 60)
  then $c - 60
  else $c
return $d * (20 div local:a())
```



- Where have the parentheses gone?

Static Analysis: Parsing

XQuery parsing

```
# IfExpr ::= "if" "(" Expr ")" "then"
#   ExprSingle "else" ExprSingle
if() → expr:
  # no IF found: return
  if !consume(IF): return NULL

  # build expression
  check-for(OPENING_PAR)
  expr if ← exprSingle()
  check-for(CLOSING_PAR)
  check-for(THEN)
  expr then ← exprSingle()
  check-for(ELSE)
  expr else ← exprSingle()
  return new if(if, then, else)
```

```
# AdditiveExpr ::= MultiplicativeExpr
#   (("+"|"-") MultiplicativeExpr)*
additiveExpr() → expr:
  # parse first operand
  expr op1 ← multiplicativeExpr()
  if op1 = NULL: return NULL

  # check for addition
  if consume(PLUS):
    expr op2 ← multiplicativeExpr()
    if op2 = NULL: raise error
    return new addition(op1, op2)

  # check for subtraction
  if consume(MINUS):
    expr op2 ← multiplicativeExpr()
    if op2 = NULL: raise error
    return new subtraction(op1, op2)

  # otherwise, return first operand
  return e1
```

Static Analysis: Parsing

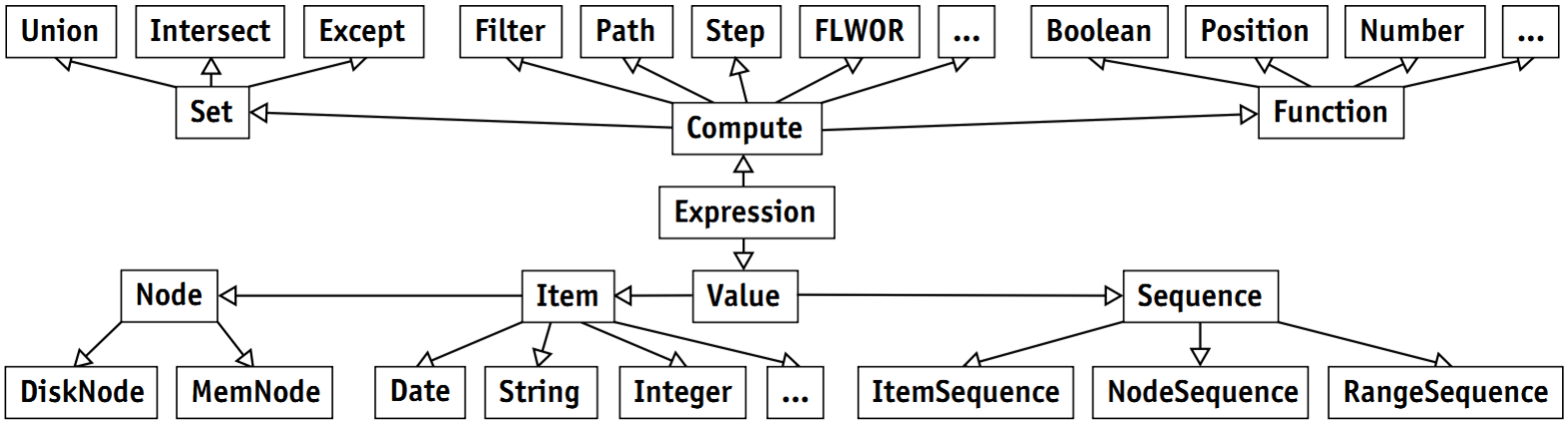
XQuery expressions

```
# EXPRESSION (SUPER CLASS)
expr:
  abstract evaluate() → value

# CONDITIONAL EXPRESSION: IF
if extends expr:
  # arguments
  expr if, then, else
  # evaluate expression
  evaluate() → value:
    if if.evaluate():
      return then.evaluate()
    else:
      return else.evaluate()
```

```
# ARITHMETIC EXPRESSION: ADDITION
addition extends expr:
  # operands
  expr op1, op2
  # evaluate expression
  evaluate() → value:
    return op1.evaluate() + op2.evaluate()

# ARITHMETIC EXPRESSION: SUBTRACTION
subtraction extends expr:
  # operands
  expr op1, op2
  # evaluate expression
  evaluate() → value:
    return op1.evaluate() - op2.evaluate()
```

Static Analysis: Parsing

XQuery expressions



- diagram shows some of the XQuery *expressions* in BaseX
- *Values* result from query evaluation, but can also be located in the AST

Static Analysis

Scanning, parsing: what comes next?

- our input character stream has been transformed to *tokens*
- tokens have been converted to an *abstract syntax tree*
- in the upcoming *compilation* phase, we will:
  - perform *type checks* to find errors and enable more optimizations
  - *statically optimize* the tree to speed up execution
- the *decorated* and *optimized* tree can then be *evaluated*
- results of query evalution can be *serialized*,
  or *passed on* to any other processing step