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:

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

```python
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*: interpret input differently, depending on context
Static Analysis: Scanning

Regular expressions

The syntax of tokens can be defined via regular expressions:

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Examples</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>&quot;a&quot;</td>
<td>single character</td>
</tr>
<tr>
<td>[0123]</td>
<td>&quot;0&quot;, &quot;1&quot;, &quot;2&quot; or &quot;3&quot;</td>
<td>class of characters</td>
</tr>
<tr>
<td>[^0123]</td>
<td>everything but &quot;0&quot;, &quot;1&quot;, ...</td>
<td>exclusion of characters</td>
</tr>
<tr>
<td>abc</td>
<td>cba</td>
<td>&quot;abc&quot; or &quot;cba&quot;</td>
</tr>
<tr>
<td>a?</td>
<td>&quot;&quot;, &quot;a&quot;</td>
<td>zero or one character</td>
</tr>
<tr>
<td>a*</td>
<td>&quot;&quot;, &quot;a&quot;, &quot;aa&quot;, &quot;aaa&quot;, ...</td>
<td>zero or more characters</td>
</tr>
<tr>
<td>a+</td>
<td>&quot;a&quot;, &quot;aa&quot;, &quot;aaa&quot;, ...</td>
<td>one or more characters</td>
</tr>
<tr>
<td>(ab</td>
<td>ba)*</td>
<td>&quot;&quot;, &quot;ab&quot;, &quot;ba&quot;, &quot;abba&quot;, ...</td>
</tr>
</tbody>
</table>
## Static Analysis: Scanning

### Lexical tokens: States and Constructors

<table>
<thead>
<tr>
<th>State</th>
<th>Token</th>
<th>Constructor</th>
<th>Description</th>
<th>Transition to State</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTENT</td>
<td>TEXT</td>
<td>[^&lt;]+</td>
<td>content</td>
<td>CONTENT</td>
</tr>
<tr>
<td>CONTENT</td>
<td>L_BR_CLOSE</td>
<td>&lt;/</td>
<td>closing tag</td>
<td>ELEMENT</td>
</tr>
<tr>
<td>CONTENT</td>
<td>L_BR</td>
<td>&lt;</td>
<td>opening tag</td>
<td>ELEMENT</td>
</tr>
<tr>
<td>ELEMENT</td>
<td>NAME</td>
<td>[A-Za-z][A-Za-z0-9]+</td>
<td>element/attribute name</td>
<td>CONTENT</td>
</tr>
<tr>
<td>ELEMENT</td>
<td>SPACE</td>
<td>[ \t\r\n]+</td>
<td>white spaces</td>
<td>CONTENT</td>
</tr>
<tr>
<td>ELEMENT</td>
<td>EQ</td>
<td>=</td>
<td>equal sign</td>
<td>CONTENT</td>
</tr>
<tr>
<td>ELEMENT</td>
<td>R_BR</td>
<td>&gt;</td>
<td>ending tag</td>
<td>CONTENT</td>
</tr>
<tr>
<td>ELEMENT</td>
<td>CLOSE_R_BR</td>
<td>/&gt;</td>
<td>empty tag</td>
<td>CONTENT</td>
</tr>
<tr>
<td>ELEMENT</td>
<td>QUOTE</td>
<td>&quot;</td>
<td>double quote attribute value</td>
<td>QUOTED</td>
</tr>
<tr>
<td>QUOTED</td>
<td>ATT_VAL</td>
<td>[^&quot;]+</td>
<td>attribute value</td>
<td>CONTENT</td>
</tr>
</tbody>
</table>

Initial state: **CONTENT**
Static Analysis: Scanning

Simple scanner implementation: greedy scanning

```plaintext
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

<table>
<thead>
<tr>
<th>Type 1</th>
<th>Type 2</th>
<th>Type 3</th>
</tr>
</thead>
</table>
| # state: ELEMENT  
if ch = '=':  
    return EQ  
if ch = '/':  
    ch ← next char  
if ch = '>':  
    state ← CONTENT  
    return CLOSE_R_BR  
else:  
    raise error | # state: CONTENT  
if ch = '<':  
    state = ELEMENT  
    ch ← next char  
if ch = '/':  
    return L_BR_CLOSE  
else:  
    jump one char back  
    return L_BR | # 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: \( \text{lhs} ::= \text{rhs} \)
- Examples: \( \text{element} ::= \text{startTag} \text{ content} \text{ endTag} \)
- \( \text{startTag} ::= \text{L\_BR} \text{ NAME} \text{ R\_BR} \)
- Extra-grammatical constraints cannot be specified by the grammar itself.

Do you remember what Well-Formedness Constraints are?
Recursive-descent parser

- in a \textit{recursive-descent} parser, production rules are mapped to \textit{functions}:
  - the function is named like the left-hand side of the rule (\texttt{lhs})
  - the function contains calls to the symbols of the right-hand side (\texttt{rhs})
- tokens (\textit{terminals}, in upper case) are “consumed”:

```plaintext
# startTag ::= L_BR NAME R_BR

element() \rightarrow \textbf{void}:
  startTag() \rightarrow \textbf{void}:
    consume(L_BR)
    consume(NAME)
    consume(R_BR)

consume(token token) \rightarrow \textbf{void}:
  # retrieve next token
  if token \neq \texttt{scanner.next()}: raise error
```

# element ::= startTag content endTag
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 rewrites

- if we have alternatives in a grammar (I), which rule is the one to follow?
  check next token (perform one look-ahead):

Rule:

\[
\text{space} \rightarrow \varepsilon \mid \text{SPACE}
\]

Rules using `space`:

- `startTag` → `... space R_BR`
- `attribute` → `... space EQ ...
- `endTag` → `... space CLOSE_R_BR`
- `emptyTag` → `... space R_BR`

```java
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

индивидуальн The good thing is: most X... grammars are already expressed in LL(1).
Static Analysis: Parsing

Examples for unambiguous look-ahead rules

OLD:  
\[ \text{element} \rightarrow \text{startTag content endTag | emptyTag} \]
\[ \text{startTag} \rightarrow \text{L_BR NAME attributeList space R_BR} \]
\[ \text{emptyTag} \rightarrow \text{L_BR NAME attributeList space CLOSE_R_BR} \]

NEW:  
\[ \text{element} \rightarrow \text{element1 element2} \]
\[ \text{element1} \rightarrow \text{L_BR NAME attributeList space} \]
\[ \text{element2} \rightarrow \text{R_BR content endTag | CLOSE_R_BR} \]

OLD:  
\[ \text{attributeList} \rightarrow \varepsilon | \text{SPACE attribute attributeList} \]

NEW:  
\[ \text{attributeList1} \rightarrow \varepsilon | \text{SPACE attributeList2} \]
\[ \text{attributeList2} \rightarrow \varepsilon | \text{attribute attributeList1} \]
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
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?

...
[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 ("," "$" VarName TypeDeclaration? "=" ExprSingle)*
[37] WhereClause ::= "where" ExprSingle
[38] OrderByClause ::= (("order" "by") | ("stable" "order" "by"))) OrderSpecList
[39] OrderSpecList ::= OrderSpec ("," OrderSpec)*
[40] OrderSpec ::= ExprSingle OrderModifier
[41] OrderModifier ::= ("ascending" | "descending")? ("empty" ("greatest" | "least"))? ("collation" URLLiteral)?
[42] QuantifiedExpr ::= ("some" | "every") "$" VarName TypeDeclaration? "in" ExprSingle ("," "$" VarName TypeDeclaration? "in" ExprSingle) "satisfies" ExprSingle
[43] TypeswitchExpr ::= "typeswitch" ("(" Expr ")") CaseClause+ "default" ("(" VarName "return" "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)?
...
declare function local:a() {
    10 + 30
};
declare variable $b :=
    20 - local:a() \div 10;
let $c := $b \times 5
let $d :=
    \text{if} ($c > 60)
    \text{then} $c - 60
    \text{else} $c
return $d \times (20 \div local:a())

Where have the parentheses gone?
### Static Analysis: Parsing

#### XQuery parsing

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

# build expression
check-for(OPENER_PAR)
expr if ← exprSingle() # parse first operand
check-for(CLOSER_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 and
  return e1
```

---

Lecture: XML Technologies
Winter 2015/16
Dr. Christian Grün, Dr. Alexander Holupirek. DBIS Group

Universität Konstanz
Static Analysis: Parsing

XQuery expressions

```plaintext
# 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 evaluation can be *serialized*, or *passed on* to any other processing step