YAXi

Development of an XPath Interpreter

E3-208

Jens Frøkjær
Martin L. Kristiansen
Dan Malthesen
Rene Suurland
Palle B. Hansen
Ivan V. S. Larsen
Tom Oddershede

May 2004

The University of Aalborg
Department of Computer Science

Aalborg Universitet
Department of Computer Science, Frederik Bajers Vej 7E, DK 9220 Aalborg Øst

Title:

YAXi – Development of an XPath Interpereter

Project period:

Dat1, Feb. 4th - May 28th, 2004

Project group:

E3-208

Group members:

Jens Frøkjær
Palle B. Hansen
Martin L. Kristiansen
Ivan V. S. Larsen
Dan Malthesen
Tom Oddershede
Rene Suurland

Supervisor:

Albrecht Schmidt

Copies: 9

Report page count: 76

Appendics page count: 10

Total page count: 86

Abstract:

This report describes the steps taken to develop an XPath interpreter: YAXi. YAXi aims to implement the full XPath 1.0 language standard.
The main goal of the project is to produce a working XPath interpreter. The "Programming Languages and Compilers" course is used for this project.
The report comprises three main parts: I) Introduction, II) Design, III) Implementation, Test, and Conclusion.
The Introduction contains thoughts on different technologies needed for building a working XPath interpreter.
The Design specifies a general framework of the implementation as well as an introduction to different parser technologies.
The last part consists of the four chapters: Implementation, Test, Study Report, and Conclusion. The Implementation chapter describes the important parts of the implementation process. The Test chapter outlines how we performed various tests to verify that YAXi in fact does implement all features of the XPath 1.0 standard. The Study Report evaluates the educational process of the project and finally the the conclusion of the report.

Preface

This report is the result of a DAT2 semester at the university of Aalborg. The project uses the course "Programming Languages and Compilers" as PE-course. This project is study-oriented with no commercial or economic interests involved, thus the main object of the project is to develop a functional program.

Reference resources are marked with [*number*]. The corresponding *number* can be found in the Bibliography in the back of the report.

The web-site "www.cs.auc.dk/∼fr0/" contains: Source code, tests, JavaDoc, a copy of the report, and a copy of the compiled interpreter.

We would like to thank our supervisor, Albrecht Schmidt, for assistance during the project period.

Aalborg, May 2004

Jens Frøkjær

Palle B. Hansen

Martin L. Kristiansen

Ivan V. S. Larsen

Dan Malthesen

Tom Oddershede

Rene Suurland

Contents

1

Introduction

1 Initial Thoughts

1.1 Purpose

The purpose of this DAT2 project is to understand and use compiler technologies. We have chosen to implement the language XPath [8] which is an already existing language defined by the World Wide Web Consortium (W3C) [12].

1.2 System Definition

The system developed should be able to perform XPath queries on any given XML-file. The result of a query will be presented in several different ways chosen by the user, e.g. in XML format or as a graphical tree-representation of the input XML-file. In either cases it is supposed to represent the output in a fashioned manner.

1.3 Problem Domain

With the development of Web technologies, the need for storing data keeps increasing. One of todays popular standards is the XML language [14], which structures and separates data inside a plain text document. As XML documents can contain large amounts of data, the need for selecting specific parts of these documents is apparent. An example of an application could be a

task which requires selecting specific local news from an XML document containing news from all over the world. This introduces a need for a language able to perform selections on such large amounts of structured data.

The main task of this project is to implement an interpreter from scratch which can process and select specific parts of an XML document. W3C, an organization which creates standards that define various languages for use on the Internet, has published a standard for a language called XPath [8]. XPath is a query language designed for selecting data contained within XML documents. During this project an interpreter, called YAXi, will be implemented. This interpreter will conform to the requirements specified in the XPath standard developed by W3C.

2 XML

XML is a language for containing and managing information. It is a family of technologies that can do everything from formatting documents to filtering data. XML builds on the philosophy that information handling should be useful as well as flexible by refining it to its purest and most structured form while still maintaining high readability.

The purpose of this project is to implement an XPath interpreter and therefore this chapter will only look at how XPath represents XML documents internally and which considerations were made before the design process was started.

This chapter is mainly based on Learning XML [23], Java API for XML Processing [19], Document Object Model [11], and About SAX [6].

2.1 Data Model

The following section will primarily be based on Section 5 in the XPath Standard [8], *Data Model*, and the Section *Nodes* in Perfect XML [22].

The XPath language sees an XML document as a tree. All elements contained in the XML document are represented in the tree as nodes. The structure of the tree is actually a lot like that of a file system, where the nodes in the tree represent the files and folders in the file system hierarchy. XPath uses a syntax similar to the path-like addressing of a file system, which makes a tree structure an ideal choice for the data model of XPath. The tree structure in XPath has seven different node types, which are listed below.

Document element (Root node) The document element is also called the root node. The document element contains the whole XML document, except for the comments and processing instructions that may be contained outside the start tag of the XML document. A document node can be found in Line 2 of Listing 2.1.

Element The element node has the special property that it can contain all node types except the root node. An element node can be either a leaf or branching point in the XML tree. Line 2 in Listing 2.1 contains an element node.

Attribute An attribute node is a leaf node. The node may be seen as part of an element node because it contains information about the element, but an attribute is in fact a separate node. An attribute node can be found at Line 2 in Listing 2.1.

Text A text node is a leaf node. One element no may contain zero or more text nodes. The text node may be separated by comments, processing instructions and element nodes. A text node can be found at Line 4 in Listing 2.1.

Comment The comment node does not have any significant effect on the contents of the document, it is just provided in order for the user to add comments to the document. A comment node can be found at Line 7 in Listing 2.1.

Processing instruction Like a comment, this node is added to the DOM tree for sake of completeness. A processing instruction is used in XML as a way of keeping processing specific information. A typical processing instruction can be found at Line 9 in Listing 2.1. In this case the processing instruction tells a program where to find a stylesheet.

Namespace A namespace may look like an attribute, but it is not an attribute because it has an effect on the node and its descendants. If a namespace is declared in an element, the children of the element inherits the namespace from its parent. The scope of a namespace is hierarchical.

In an XML document the first line is called a document prolog and is not considered as one of the seven node types. An example of a document prolog can be found at Line 1 in Listing 2.1. The document prolog holds information about things like type, text encoding, and instructions to the XML processors. It is, however, not included in the tree.

The tree that XPath works on has the following structure:

- The root and the element nodes contain a list of their children.
- Element, text, comment, and processing instruction can be children of other nodes.
- Attribute and namespace nodes are not children of other nodes, but they accept an element node as their parent.

In other words, attribute and namespace nodes are contained inside their parent node and provide information about their parent node.

```
1  <?xml version="1.0"?>
2  <group name="E3-208" alias="d204a">
3      <member name= "Palle B. Hansen" shoesize="44">
4          <phone>12324665</phone>
5          <phone>23233321</phone>
6      </member>
7      <!-- dat2 project -->
8      <project>
9          <?xml-stylesheet type="text/xml" href="limited.xsl"?>
10         <homepage xmlns="http://www.cs.auc.dk/d204a" />
11         <title>XPath Interpreter</title>
12         <supervisor name="Albert" />
13         <url>http://www.cs.auc.dk/d204a</url>
14         <email>d204@cs.auc.dk</email>
15     </project>
16 </group>
```

Listing 2.1: Example of an XML document

The following list, maps node types to line numbers of the above example.

Node type	Line
The document prolog	1
Document element	2
Processing instruction	9
Element	2, 3, 4, 8, 10, 11, 12, 13, 14, and 15
Attribute	2 and 3
Text*	4, 5, 11, 12, 13, and 14
Comment	7
Namespace	9

* Excluding text nodes, which contains only whitespaces.

The following paragraph will describe how the XML document in Figure 2.1 is mapped into a Document Object Model tree by describing how the mapping is done on the first occurrence of all of the various node types.

The first thing that is added to the DOM tree is the root node. The root node is a pointer to the first element node in the DOM tree. The next thing that is added to the DOM tree is one element node and two attribute nodes. The element node gets the value `group`. The first attribute is named `name` and gets the value `Palle B. Hansen`, while the second attribute gets the name `shoesize` and the value `44`. At line four an element node and text node is added. The text node gets the value `12324665` and the element node the value `phone`. At line seven a comment node is added, which gets the value `dat2 project`. At line eleven a processing instruction is added to the DOM tree with the value `http://www.cs.auc.dk/d204a`.

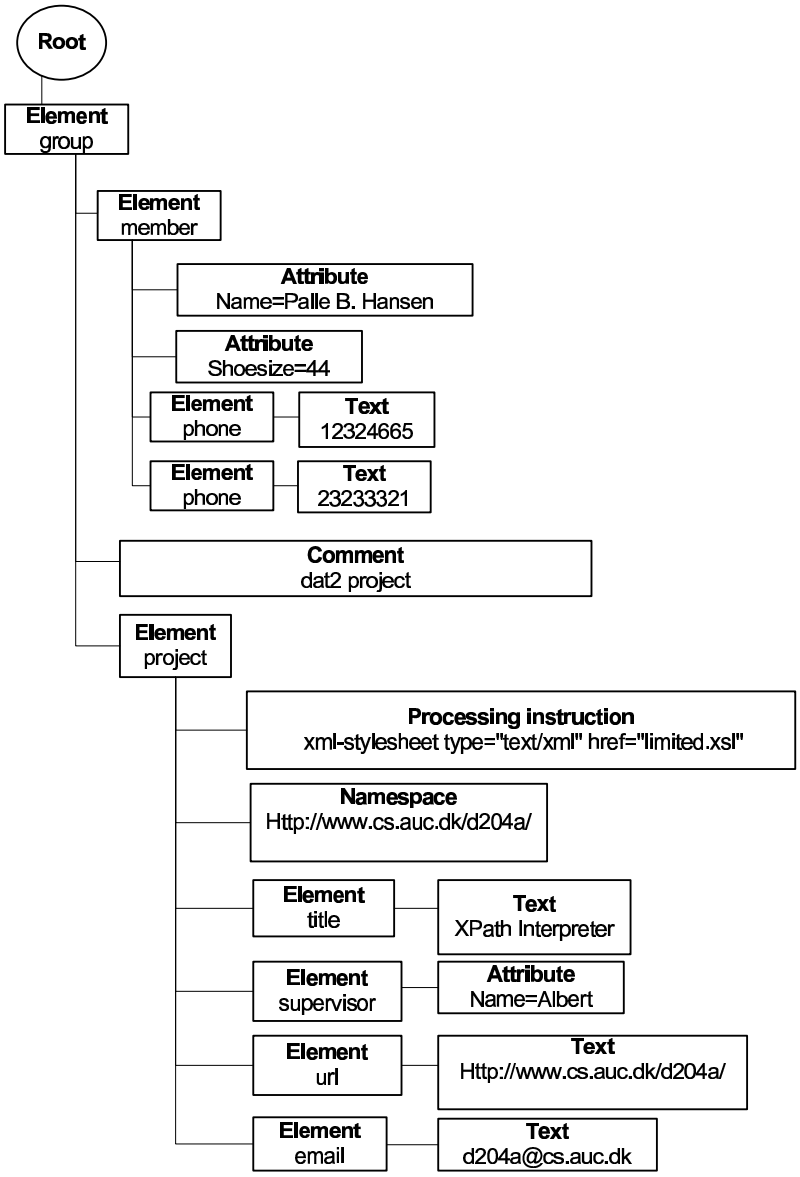

Figure 2.1: An XML document viewed as a DOM tree

2.2 API for XML

We had to find an XML parser API that provided us with a data type suitable for our processing needs. We decided to use the Java API for XML Processing (JAXP). JAXP provides the two industry standard APIs Document Object Model (DOM) and Simple API for XML (SAX), each of which provide their own mechanism for parsing XML documents.

2.2.1 SAX

SAX is short for Simple API for XML [6]. As the name implies, this is an API for parsing XML data. SAX is an event-based parser [5] that will not build a tree, but reacts upon events happening in the data input stream. The principles used by SAX are very similar to those of a lexer [4]. SAX divides an input string into tokens using a grammar that can handle XML events. SAX could easily be compared to a finite automaton [25] designed to recognize a regular language, or in SAX's case, the XML grammar. To use SAX along with XPath, additional code that builds a tree would have to be written because XPath needs random access to the tree.

A simple example shows how SAX parses XML data.

```
<?xml version="1.0"?><doc><para>Hello, world!</para></doc>
```

An event-based interface will break the structure of this document down into a series of linear events, such as these:

```
start document
start element: doc
start element: para
characters: Hello, world!
end element: para
end element: doc
end document
```

Big-Oh notation for SAX depends on the implementation. It is only the current state of the parser that matters along with the states saved in memory. This gives us a general space and time complexity of $O(n)$, where n is the length of the document.

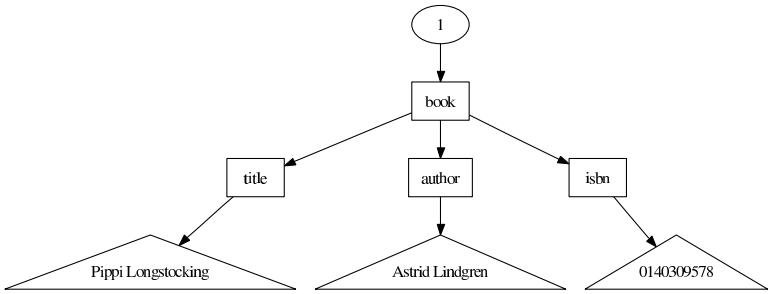

Figure 2.2: DOM tree

2.2.2 DOM

DOM is short for Document Object Model. It defines an interface that provides dynamic access to documents and allows for further processing and incorporation of the ability to access all nodes in the document randomly at any time. The DOM is a tree structure where each node represents an element in the XML document. The API provides, among other things, methods for traversing and changing the tree. The interface is divided into two parts: a core and an HTML [7] part.

Basically DOM works like a parser and builds a DOM tree that reflects the XML structure. It builds a tree in a similar way a parser builds an abstract syntax tree. For this project the DOM API in the JAXP package is used to build the DOM tree.

Time complexity should also be considered. When creating a DOM tree it seems obvious that the time used for building the tree is $O(n)$ and it will require $O(n)$ to store the tree in memory. Traversal of the tree will also require $O(n)$. According to an article [2] on XML.com it is actually $O(n^2)$ when using the method `getItem()` for traversal. A DOM tree works like a linked list, which means it is not possible to access the list at a random location, which forces a complete traversal of the list when searching. Should this function be used in a loop with n elements there will be $0.5(n^2 - n)$ references, or $O(n^2)$. This means that the DOM interface will be slow and require lots of memory when used on large documents.

2.2.3 DOM or SAX

It can be argued that SAX is a speed-wise efficient implementation compared to using an implementation based on DOM. This is because SAX reads the XML document's structure only once as an I/O stream instead of building a

representation of the XML document in memory. This however is not very useful in this project unless the parse information is stored in memory. For example, when SAX transmits a chunk of character data, there is no way to access the parent of the current element because it is not stored in memory. One does not even know whether the character data you receive is the entire contents of a continuous string or just a fragment, you have to wait for the following SAX events to establish that. This means that SAX is also much more efficient memory-wise than DOM, because the DOM has to build a tree in memory. This could cause a problem when DOM parses large XML files. Suppose a large table is stored in XML format, and one just wanted to count the records in the table. SAX would do the job efficiently. On the other hand if one would want to access the records multiple times to sort the table using some algorithm, a DOM tree would be very useful, assuming one has enough memory for the entire table. In many cases, however, one needs to process one record at a time in a loop. In a situation like that it does not make sense to build a tree containing the entire XML document, so the programmer could successfully use SAX, where you only have the most recent element accessible in memory. XPath queries will often require elements to be accessed in random order instead of just sequential order. Because the DOM builds a tree that can be traversed in any order you like, it is apparent that using the DOM approach would fit this project perfectly. Furthermore we estimate that the speed and memory issues with the DOM API will not cause a problem, as our interpreter will probably not be used with extremely large XML documents.

3 Introduction to XPath

The XPath language is a standard developed by W3C to be a subset of the standards XPointer [10] and XSLT [9]. XPath, or the XML Path Language, was developed because both of the aforementioned standards needed a way to select or point to parts of an XML document. XSLT needs XPath for selecting elements to apply templates to and XPointer for referencing between different parts of a document. Even though XPath's main usage is selecting elements from an XML document, it can also be used to evaluate string, numerical, and boolean expressions.

In order for XPath to work on an XML document, the document is conveniently converted into a tree of nodes, which allows XPath to work on the logical structure that XML documents provide. These nodes have different types which will be covered later in this section. To get an idea of how XPath views an arbitrary XML document in a tree structure, take a look at Figure 2.1.

When selecting elements in a document, XPath provides, via so-called axes, a path-like (e.g., `/a/b/d`) access to those elements–hence the name XPath. Expressions in XPath will evaluate to an object of the basic types shown in Figure 3.1.

This brings us to the most interesting and important part of an XPath expression: the Location Path.

This chapter is based on the XPath Standard [8].

3.1 Location Path

The XPath language is designed for selecting parts of an XML document. XPath uses the DOM [11] standard as data model. To understand how an expression is evaluated, one has to understand the concept of a *context*. The context in which an expression is evaluated consists of the following:

- The context node. A node from a given node-set.
- Context position. The position in the current node-set.
- Context size. The size of the current node-set.
- A set of variable bindings.
- A function library. See Section 4, "Core Function Library" in the XPath Standard.
- A namespace in the scope of an expression.

An XPath expression is a series of location steps which are separated by slashes (`/`). The location path selects one or more nodes relative to the context node. The location steps are evaluated one at a time from left to right. The node-set returned after each step is used as the context for the next step. The result of a location path is a node-set of zero or more nodes.

There are two types of location paths, an absolute and a relative. The absolute location path starts with a slash (`/`) and the relative does not. The absolute location path selects nodes from the root of the tree, whereas the relative location path selects nodes relative to the context node. A location step in XPath consists of these three parts: an *axis*, a *node test* and zero or more *predicates*. The axis specifies the path to follow in the document structure used by XPath, which is more or less the same as used by the DOM standard. The node test is used to select only nodes with a certain name or type. And finally the predicates work as a filter on the nodes selected by the node test.

3.2 Expressions in XPath

The purpose of this section is to describe what legal XPath expressions are and how such expressions are evaluated. Also covered in this section is how

the various operators in XPath work and how they are used with the four basic types in XPath.

The primary syntactic production rule in XPath is an expression. An expression is represented in the XPath grammar as the production rule `expr` in the XPath standard. When an XPath expression is evaluated the result must be of these types:

- Node-set (an unordered collection of nodes without duplicates)
- Boolean (true or false)
- Number (a floating-point number)
- String (a sequence of characters)

Figure 3.1: The four basic types allowed as result values of an XPath expression

The example in the following section will use the `PathExpr` production rule in the XPath grammar, as it's starting point. Figure 3.2 shows the derivation sequence through the XPath grammar from the 14th production rule to the first production rule, `LocationPath`. All the production rules can be found in Appendix B.

`Expr` → `OrExpr` → `AndExpr` → `EqualityExpr` → `RelationalExpr` → `AdditiveExpr` → `MultiplicativeExpr` → `UnaryExpr` → `UnionExpr` → `PathExpr` → `LocationPath`

Figure 3.2: The derivation sequence through the grammar from `Expr` to `LocationPath`.

Under normal circumstances, XPath is used for selecting nodes in an XML document with the use of the `LocationPath` rule. XPath can however also be used for string manipulation and evaluation of boolean and arithmetic expressions. The string manipulations allowed by XPath are performed by the `FunctionCall` production rule.

To examine if XPath supports the expressions shown in Figure 3.3, one could take a look at the `FilterExpr` production rule which becomes the `PrimaryExpr` production rule which finally becomes the `FunctionCall` production rule. The return type of function calls depend on the function called, but must be one of the types from the types mentioned in Figure 3.1.

This means that XPath allows a `FunctionCall` as an expression. As said earlier, XPath also evaluates boolean and arithmetic expressions.

- `substring("Hello", 2)` yields the string `ello`
- `number("34.2")` yields the float 34.2
- `boolean(/AAA)` yields true if the node-set is not empty and false otherwise

Figure 3.3: Expressions that merely contain function calls

3.3 Operators in XPath

XPath provides a series of operators. How they work, and on which types they can be used will be explained in the following sections.

The operators in XPath are left associative, and the precedence rule is as follows, where highest number has highest precedence:

1. `or`
2. `and`
3. `=` and `!=`
4. `<=`, `<`, `>`, and `>=`
5. `+`, `-`
6. `mod`, `div`, `*`

3.3.1 Node-set Operators

A node-set is the result of a location path expression and it contains zero or more nodes. Figure 3.2 shows that the location path is just an expression by showing the production derivation sequence for the location path from the primary expression, `Expr`, to the location path expression, `LocationPath`.

The node-set is the only one of the basic types that can be combined by using the `|` operator. The `UnionExpr` production rule states that the `|` operator should work with all the types in the XPath language, but the standard says:

The `|` operator computes the union of its operands, which must be node-sets.

Figure 3.2 shows that the step before a location path is a `PathExpr`. The `PathExpr` can become a `FilterExpr` which can then become a `PrimaryExpr` which can then become either a `VariableReference, '(' Expr ')'`, `Literal`, `Number`, or `FunctionCall`. That means that we can, according to the syntax, join e.g., numbers with node-sets. Since there are no rules for joining types other than a node-set, joining only applies to node-sets according to the XPath Standard [8].

3.3.2 Boolean Operators

XPath offers two boolean operators: `and` and `or`.

The `or` operator returns *true* if at least one of its operands is *true*, otherwise it returns *false*.

The `and` operator returns *true* if and only if both of its operands are *true*, otherwise it returns *false*.

3.3.3 Equality Operators

If a string, number, or boolean value has to be compared to a node-set, the node-set will have to be converted into the same type as that of the object it is being compared to. This is because non-node-set objects can not be converted into a node-set, while the same is not true the other way around. The following section tells when the use of the equal operator would return `true`. If the not equal operator had been used on the same comparisons the result would have been false. The six rules tell how the objects are converted. But since a node-set often must be converted more than once before they are compared, they are described separately from the other objects.

Node-set = String	The result will be true if a node in the node-set has a string value that equals the string value of the string.
Node-set = Number	The result will be true if a node in the node-set has a string value that, converted with the the number function, equals the numerical value of the number.
Node-set = Node-set	The comparison between two node-sets will be true if both node-sets contain a node with the same string value.

If none of the objects are node-sets the equality comparisons must be done in the following ways.

If one of the objects is a boolean	The object that is not a boolean will be converted with the boolean functions from the function core.
If one of the objects is a number	The object that is not a number will be converted into a number using the number functions from the function core.
If one of the objects is a string	See Node-set = String.
Boolean - Boolean	The result would be true if both object are true or false.
String - String	Two strings are equal if and only if they consist of the same sequence of UCS characters.
Number - Number	Numbers are compared for equality according to IEEE 754 [21].

3.3.4 Relational Operators

Relational operators work similarly to equality operators, see Section 3.3.3.

3.3.5 Numeric Operators

The operators used on numbers are `+`, `-`, `mod`, and `div`. When used, the operands will be converted into numbers with the number functions from the function core before the expression is evaluated.

3.4 Location Step

A location step consists of three parts: an `AxisSpecifier`, a `NodeTest`, and zero or more `Predicate`s.

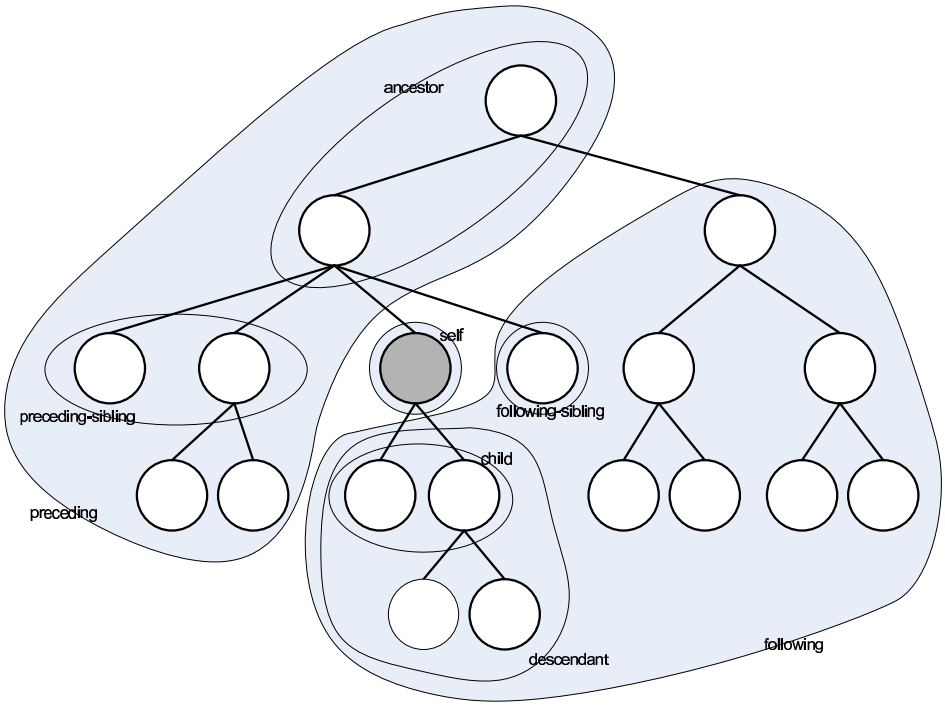

Figure 3.4: A map that show the effect of the various axes

3.4.1 `AxisSpecifier`

The axis allows for forwards and backwards traversal in a document. XPath contains a number of axes which is illustrated in Figure 3.4. The grey node is the current node and the circled sets illustrate which nodes will be selected given the labeled axis.

There are axes which do not appear on Figure 3.4, namely the axes *descendant-or-self* and *ancestor-or-self*. These axes are the union of, respectively, *descendant* and *self* and *ancestor* and *self*.

3.4.2 `NodeTest`

A node test is used to test if a node or a node characteristic is contained in an axis. The test is performed on the context node. Figure 2.1 is used in the following example:

Expression 1 `/descendant-or-self::phone`

The result of Expression 1 would return the two nodes `<phone>1234565</phone>` and `<phone>23233321</phone>`, as these element nodes match the node test.

3.4.3 Predicate

A predicate is an expression that filters nodes from the node-set selected by the `AxisSpecifier`. Predicates are written after the `AxisSpecifier` enclosed by square brackets (`[ ]`). Every expression in a predicate is evaluated to a boolean for each node in the node-set.

The following Expression is based on the XML document shown in Listing 2.1.

Expression 2 `/desdendant-or-self::phone[position()=2]`

Expression 2 will return the node `<phone>23233321</phone>`, as this node is the second element matching the node test of the axisspecifier. As a predicate always evaluates to a boolean, the result of Expression 2 would be the same, if the predicate was written like `[2]`.

3.5 Abbreviated Syntax

An XPath implementation must provide the user with abbreviated syntax that allows for simplified XPath queries.

There are three abbreviations. The most important one specifies that `child::` can be left out in an expression. This means that the expression `AAA` is abbreviated syntax for the expression `child::AAA`.

Another abbreviation is `//`, which is abbreviated syntax for `/descendant-or-self::node()/`. Therefore a query like `//AAA` would be translated into `/descendant-or-self::node()/child::AAA`.

The last abbreviation is the `@`, which is abbreviated for `attribute::`, so `@AAA` would be translated into `attribute::AAA`.

II

Design

4 Design Criteria

This chapter describes which parts of XPath we have chosen to implement in our interpreter. As a basic assumption we want to implement the whole XPath 1.0 language.

This chapter is based on the XPath standard [8].

4.1 Compiler or Interpreter

In order to determine whether our XPath implementation should be built as a compiler or an interpreter, we need to summarize what the difference between the two are.

4.1.1 Compiler

A compiler is a program that translates a source program written in a high-level language into a lower level language for a computer architecture. This can then be executed without using the compiler again.

4.1.2 Interpreter

An interpreter reads a source program written in a high-level language as well as some data for this program. The interpreter then runs the program against the data in order to produce the output.

4.1.3 What to Choose?

The question to be answered in this section is whether to build an interpreter or a compiler for implementing the XPath language. There are advantages and disadvantages with both solutions which must be considered before choosing which technology to use.

One of the advantages of using a compiler would be that a query could be compiled and saved for later execution in, for example, a virtual machine. This would be useful if the same query had to be executed many times. A disadvantage would be that if a query is to be executed only once, the execution time would be greater than that of an interpreter.

If most queries are to be executed only once, it would make no sense to compile the queries. Therefore an interpreter would be the best choice for the implementation.

Because of this an interpreter was chosen to be built for implementing the XPath language.

4.2 The `rand()` Extension

One thing we feel is missing in the XPath language, is some form of randomization. Therefore we have decided to implement an extension to the XPath function libary–a function we have named `rand()`.

A function that provides randomization can be utilized in many situations. An example of where such a function would be useful, is when using XPath to implement banner-cycles on web sites or when selecting random quotes.

The `rand()` function has the following syntax:

syntax	*description*
`rand()`	If no arguments are given, `rand()` will return a random integer between 1 and the length of the context node-set.
`rand(n)`	If one argument is given, `rand(n)` will return a random integer between 1 and n (where $1 \leq n \leq$ |node-set|)
`rand(n,m)`	If two arguments are given, `rand(n,m)` will return a random integer between n and m (where $1 \leq n \leq$ |node-set| and $1 \leq m \leq$ |node-set|)

4.2.1 Examples of Use

The primary use of the random function must be in a predicate like in Expression 3. The result of the example is a node-set containing one random element with the name `QUOTE` from the context node-set.

Expression 3 `//QUOTE[position()=rand()]`

When the `rand()` function is called in a predicate, it is only randomized once–not once for every node in the node-set. If this was not the case, functions like `position()` would return a completely unexpected result, because `position()` is calculated for each node. If `rand()` returned a different number for each node, the result could likely be an empty node-set because in this case `position()` might never equal a random number. Thus, only one random number is generated and therefore it is guaranteed that a node is returned.

The `rand()` function can also be used as an argument in any other function. Expression 4 shows how the `rand()` function is used as an argument in the substring function.

Expression 4 `substring("abcdefghkl", rand(7,9))`

The expression will return one of the following results:

- In case `rand(7,9)` evaluates to 7, the result yielded is the string `ghkl`
- In case `rand(7,9)` evaluates to 8, the result yielded is the string `hkl`
- And finally, in case `rand(7,9)` evaluates to 9, the result yielded is the string `kl`

4.3 The Output

The syntax of the output of an XPath expression is not specified in the XPath standard, so it is up to the implementation how the output should be formatted. As the input format is XML, it would make sense that the output also would be in some XML structure. Our implementation does it in the following way:

The XML output starts with the tag `<nodeset>` and ends with the tag `</nodeset>`. Inside these tags all result nodes will appear. If a node from

the result set is an element, the node will be enclosed by the tags `<Element>` and `</Element>`. Listing 4.1 shows how the output of Expression 5 would be formatted when used with the XML document in Listing 2.1

Expression 5 `//member`

```
<nodeset>
  <Element><member name= "Palle B. Hansen" shoesize="44">
        <phone>12324665</phone>
        <phone>23233321</phone>
    </member>
  </Element>
</nodeset>
```

Listing 4.1: An example of output from the interpreter, containing only one node

Note that the example in Listing 4.1 contains only one node. All other nodes which appear in the output are children of the selected node, and therefore part of the selected node.

If a node from the result set is an attribute, the tags surrounding the node would be `<Attribute>` and `</Attribute>`, and if the node is a text node the tags surrounding the selected node would be `<Text>` and `</Text>`.

As part of the graphical user interface, we have also implemented other output formats like DOT, HTML and postscript. The main purpose of these output formats is to be able to easily get an overview of the output. The DOT format draws the DOM tree while coloring the selected nodes red and the rest of the nodes green.

5 Lexers and Parsers

In order to make the interpreter analyze whether an XPath query is valid or not, we use a lexer and a parser. A language is usually specified in a vocabulary form, called the grammar file, which is processed into two separate recognizers. The language level recognizer is called the parser and the vocabulary recognizer is called the scanner or lexer. The parser recognizes the grammatical structure in a stream of tokens whereas the lexer recognizes the structure in a stream of characters. There are different ways to construct lexers and parsers. Of course one could write them by hand from scratch, but this is a straight-forward monotonous and robotic process which can be very error-prone. Therefore there exist programs designed for automating this process. The one we have chosen is called SableCC, which we will elaborate on in Section 5.2.

5.1 LL(1) and LALR(1)

There are two main groups of parsers: LL(1) and LALR(1). There is no strict equivalence between LL(1) and LALR(1), as there are LL(1) grammars that are not LALR(1) compatible and there are LALR(1) grammars that are not LL(1) compatible. One of the common features of LL(1) and LALR(1) parsers is that they look ahead one token in order to determine the next production. Figure 5.1 shows the parsing of the expression `/A`. The `(.)` indicates where the parser is currently at, meaning that it is the token following the `(.)` that is currently being inspected.

`(.)/A`	The parser is now at the start of the query and looking ahead one token. This is recognized as a `slash` (`/`). Now the parser finds a production matching this `slash`, which is the `AbsoluteLocationPath`.
`/(.)A`	We are now looking at the character `A`. This is parsed as a `Letter`. Now the parser can finish the production `AbsoluteLocationPath`, since the `Letter` will eventually derive to a `RelativeLocationPath`, which ends the derivation sequence.
`/A(.)`	Now we are at the end of the query and the whole production is finished, and therefore the query is parsed successfully.

Figure 5.1: Parsing the query `/A`

5.1.1 LL(1)

LL(1) parsers are top-down parsers. One of the disadvantages with LL(1) parsers is that they can not parse left-recursive grammars, which means that the grammar must be changed in order to make it LL(1) parsable. Actually in some cases it is impossible to make an LL(1) grammar for a specific language. On the other hand it is easier to write an LL(1) parser than for example an LALR(1) parser, because an LL(1) parser does not have to handle, for example, left-recursion.

5.1.2 LALR(1)

Where LL(1) parsers are top-down, LALR(1) parsers are buttom-up, where the parser tries to construct the parse tree from the buttom upwards. One of the advantages with LALR(1) parsers is that thay can handle left-recursive grammars, as they are buttom-up. The parser tracks the matched tokens on the righthand side. It may not know at once which production to choose, so it tracks a set of possible matching productions to find the correct one. A disadvantage with LALR(1) parser is that it is very hard to construct from hand. But luckily, as mentioned before, there are programs like SableCC that automate this process.

5.2 Compiler Compilers

A compiler compiler is a program designed to automate the process of writing compilers. A compiler compiler is utilized by rewriting a formal grammar into

a syntax that the compiler compiler in question understands and the feeding it to the compiler compiler.

There are several possibilities involved when choosing a compiler compiler. Several different combinations are available out there depending on programming language. The combination of Lex and YACC [18] allows a programmer to write a complete single-pass compiler simply by writing two specifications: one for Lex and one for YACC. The Java versions of these programs are called JLex [3] and JavaCUP [24]. Some advantages of using this combination would be:

- JLex DFA based lexers are usually faster than hand written lexers.
- The languages that can be recognized are LALR(1) compatible which means that one does not need to rewrite the grammar, as LALR(1) handles left recursion.
- Both JLex and JavaCUP are available in source code form.

And some drawbacks:

- JLex and JavaCUP have not been specifically designed to work together, so it is the programmer's job to build the links between the code generated by both tools.
- The lack of support for ASTs renders JavaCUP ill suited for multiple-pass compilers.

There are of course other compiler compilers out there. Another popular one is JavaCC, which is based on LL(k) rather than LALR(1). The whole input is given in just one file. JavaCC is based on LL(k) it is top-down, which means that all left-recursion must be eliminated.

It is here SableCC comes into the picture. SableCC is a powerful tool, that has many of the advantages of the above mentioned compiler compilers. SableCC generates a lexer, a parser, and an abstract syntax tree. This will be elaborated on in Section 5.3.

5.3 SableCC

SableCC is a powerful tool that generates both a lexer and a parser. Furthermore it generates an AST and a special tree-walker class for this AST.

One of the advantages of using SableCC is that one only has to write one grammar file for both the lexer and parser. This way it is easier to debug errors. Compiler compilers like JLex and JavaCUP create a lexer and a parser which are not designed to work together, whereas SableCC generates a lexer and a parser which are fully interoperable.

As SableCC is a buttom-up parser, it accepts LALR(1) languages. This means that we do not need to rewrite our grammar in order to eliminate left-recursion, as LALR(1) can handle left-resursion.

Another advantage with SableCC is that there is a clear separation of user generated code and machine generated code. This is handy when the code fails during development, because errors are easier to pin-point to either the user generated code or the code generated by the compiler compiler.

```
1  node.Start tree;
2  try
3  {
4    Parser p = new Parser(
5      new Lexer(
6      new PushbackReader(
7      new CharArrayReader(query.toCharArray()),10000)));
8
9    tree = p.parse();
10 }
11 catch (Exception e}
12 { //This block is called if the lexer or parser fails}
13
14 try
15 {
16   ASTAnalysisAdapter ast = new ASTAnalysisAdapter(ns);
17   tree.apply(ast)
18 }
19 catch (Exception e)
20 { //This block is called is some usercode fails}
```

Listing 5.1: How the lexer and parser is called from user-code

In the block at Line 11 in Listing 5.1, all errors from the lexer and parser are caught. This makes it easy to debug bugs in the grammar, as the whole lexer and parser is generated without any user code.

As mentioned before, SableCC generates a special tree-walker class for the AST. This class is called `DepthFirstAdapter`. In order to use this feature, one must construct a new class extending the `DepthFirstAdapter`. When applying this (Line 17 in Listing 5.1), a method corresponding to every node in the AST is called using the visitor pattern, which will be discussed in Section 6.3.

For this project SableCC was found the most suitable due to its many options

and functionality. The fact that SableCC generates a parser and a lexer which are fully compatible was also an important factor in this choice.

5.4 Backus-Naur Form

BNF is short for *Bachus-Naur form*, which is a formal mathematical way to describe a language [17]. It is used to define the grammar of a language in a way which ensures that there can be no disagreements or ambiguity as to what is allowed and what is not. A BNF grammar is so unambiguous that one can mechanicly construct a parser for it[25]. A language defined by a BNF grammar is the set of all strings you can produce by the language rules. Rules are called production rules and a rule could look like:

```
symbol := alternative1 | alternative2
```

This syntax means that you can replace the symbol on the left side of the `:=` with either alternatives on the right side. Below is a simple BNF grammar:

```
S   :=  '-' FN | FN
FN  :=  DL | DL '.' DL
DL  :=  D | D DL
D   :=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

The symbols on the left are all abbreviations: `S` is the start symbol, `FN` produces a fractional number, `DL` is a digit list, while `D` is a digit. With this grammar you can write all numbers, including negative and fractional numbers. This language however has some recursion, making it more difficult to understand (`DL := D | D DL`). This can be solved using the Extended BNF. EBNF adds three operators to BNF in order to describe languages:

- ? which means that the symbol (or group of symbols in parenhesis) to the left of the operator is optional (zero or more times).
- * which means that something can be repeated any number of times (and possibly be skipped altogether).
- + which means that something can appear one or more times.

Using these operators the language defined above can be written as:

```
S  :=  '-'?  D+ ('.'  D+)?
D  :=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

This way of writing is less complex, but it must be noted that EBNF is not more powerfull in terms of defining a language, just more convenient.

6 XPath grammar in SableCC

This chapter is mainly based on Étienne Gagnon's master thesis on SableCC [16].

6.1 The Structure

The XPath grammar file, which can be seen in Appendix A, is the result of the massaging of the grammar from the XPath standard. There are four main categories in the XPath grammar file:

Helpers	Specification of strings or regular expressions that can be used to define tokens.
Tokens	Definition of the formal tokens recognized by the lexer.
Ignored tokens	Tokens that the parser will ignore.
Productions	Definition of the productions in the grammar, specified from tokens only.

6.1.1 Helpers

Helpers is a specification of strings or regular expressions. We have defined three helpers:

```
letter = ['A'..'Z'] | ['a'..'z'] | [0x7F .. 0xFF]
```

```
digit = ['0'..'9']
all = [0 .. 0xFF]
```

These are necessary to define some of the tokens, but not needed in the productions. The helpers can be seen as substitutions.This means that when a token contains `digit`, `digit` can be substituted with `['0'..'9']`.

6.1.2 Tokens

Tokens are mainly definitions of small strings. In the grammar file, the most unambiguous tokens are put in the top of the section containing tokens. This has the consequence that tokens like `digit`, `letter`, and `all` are put in the last part of this section.

Tokens are read top-down, meaning that the first token matching the current input is returned.

This is also the reason why `literal`, `number` and `ncname` are placed in the bottom of the token section.

```
literal = ''' ([all - ''']) *  ''' | '"' ([all - '"']) *  '"';
number = digit+ ('.' digit+)? | '.' digit+;
ncname = (letter | '_') (letter | '_' | digit | '.' | '-')*;
```

One thing comes to mind when reading the tokens: `star` and `multistar` are both defined to be `*`. This has the consequence that the latter of the two is never used, but this is solved later in Section 6.2.

6.1.3 Ignored Tokens

The ignored tokens are left out by the parser, meaning that even though a specified token appears in a query, it will not be evaluated. In XPath whitespaces are ignored, causing the expression "`/*`" to return the same result as "`/ *`". This makes the interpreter more flexible when accepting user queries.

6.1.4 Productions

Productions define the relationship between tokens, and therefore they can only be defined using tokens. The first production defined will always be the first evaluated.

```
Productions
  start = expr;
```

In the grammar every XPath expression must start with a the `expr` production rule.

The grammar that SableCC accepts must be a context-free-grammar in Backus-Naur Form [15]. Actually it accepts a variation of the EBNF (Extended Backus-Naur Form), which is the BNF extended with regular expressions (`()`, `+`, `?`, and `*`). The reason why it is a variation of the EBNF, is because that parentheses, for example, cannot be used in productions. For example the production in Listing 6.1 must be massaged in order to make SableCC accept it.

```
FunctionCall ::= FunctionName '(' Argument (',' Argument)* ? ')'
```

Listing 6.1: `FunctionCall` as it is defined in the formal grammar from W3C

The first thing that must be changed when massaging the production in Listing 6.1 is the non-terminals. Every non-terminal in SableCC must be in lowercase, so non-terminals like `FunctionCall` must be changed into `functioncall`. The next thing one must change is the "`::=`" which must be substituted with a "`=`". Now substitution of all terminals with tokens must be done, as productions can only be defined by other productions or tokens. Therefore the definition of "`(`" , "`)`", and "`,`" must be defined as tokens, which we call `paren_l`, `paren_r`, and `comma`. Now the last thing needed to be done before making the grammar work with SableCC, is to eliminate the parentheses because, as mentioned before, one cannot use parentheses in productions. In order to eliminate these parentheses another production must be created. This production must express the same as `(Argument(',' Argument)*)?`, just without parentheses. This production is called `argumentlist` and can either be an `argument` or an `argument comma argumentlist`. Listing 6.2 shows the complete `functioncall` production as it appears in the XPath grammar file for SableCC in Appendix A.

```
functioncall = functionname paren_l argumentlist ? paren_r
argumentlist = {argument} argument
             | {argumentlist} argument comma argumentlist ;
```

Listing 6.2: FunctionCall massaged into SableCC

Note that in Listing 6.2 we have specified two unique names in curly brackets. These are necessary when there are more than one option in a production, and they are used for naming methods and classes in the generated code.

6.2 XPath is not LALR Parsable

As mentioned earlier, the XPath grammar is not LALR(1) parsable. In this section we will describe the problem and how we solved it.

6.2.1 The Problem

An LALR(1) parser looks ahead one token and must then solve everything on the left to proceed. Let us look at the `PredicateExpr` in the following expression:

Expression 6 `//*[/**4]`

`(.)/**4`	Everything has been parsed correctly up until this point. The `/` is the star of an `AbsoluteLocationPath` and the parser can proceed.
`/(.)**4`	The parser is now looking at a `*`. This can be parsed as a multiplication sign or a wildcard for the `AbsoluteLocationPath`

This ambiguity means that XPath is not LALR(1) parsable. Of course we could use an LALR(k) parser, but a problem would arise if we had a `PredicateExpr` like expression 7, where $*^{k+1}$ should be read as $k+1$ `*`. In this case we would encounter exactly the same problem as before. Even though we read k tokens ahead, we will still not be able to determine whether the next token is a multiplication or wildcard sign.

Expression 7 `//*[/*`$^{k+1}$`4]`

6.2.2 The Solution

In order to correct the problem described in Section 6.2.1, there is an approach in Section 3.7 of the XPath standard [8]. It states:

> "If there is a preceding token and the preceding token is not one of `@`, `::`, `(`, `[`, `,` or an `Operator`, then a `*` must be recognized as a `MultiplyOperator` and an `NCName` must be recognized as an `OperatorName`."

The problem has now changed from being parser related to being lexer related. There are now two solutions for changing the way the lexer goes about its business to make the grammar compliant with Section 3.7 of the XPath standard.

First Solution

As in SableCC, the first matching token is always returned. It would be possible to add tokens to the grammar file like:

```
Tokens
    /*
    /
    *
```

This would result in `/*` being read before any `/` and `*`. This should be done for all of the tokens stated above. A large rewrite of the grammar would be required for this solution and the grammar would be less readable, so another solution is preferable.

Second Solution

As this is a lexer-related problem, perhaps the lexer could be tweaked so that the grammar and the parser would not be changed in any way. The rule to implement is pretty simple: *The type of the preceding token determines which token to be returned next.* Listing 6.3 shows the unmodified code before the new rule is applied. The rule can be seen in Java code in Listing 6.4 at Line 562.

```
547 Token new0(int line, int pos) { return new TMod(line, pos); }
548 Token new1(int line, int pos) { return new TSlash(line, pos); }
549 Token new2(int line, int pos) { return new TParenL(line, pos); }
550 Token new3(int line, int pos) { return new TParenR(line, pos); }
551 Token new4(int line, int pos) { return new TBracketL(line, pos); }
552 Token new5(int line, int pos) { return new TBracketR(line, pos); }
553 Token new6(int line, int pos) { return new TAt(line, pos); }
554 Token new7(int line, int pos) { return new TComma(line, pos); }
555 Token new8(int line, int pos) { return new TPipe(line, pos); }
556 Token new9(int line, int pos) { return new TOr(line, pos); }
557 Token new10(int line, int pos) { return new TAnd(line, pos); }
558 Token new11(int line, int pos) { return new TPlus(line, pos); }
559 Token new12(int line, int pos) { return new TMinus(line, pos); }
560 Token new13(int line, int pos) { return new TStar(line, pos); }
561 Token new14(int line, int pos) { return new TMultistar(line, pos)
        ; }
562 The list goes on
```

Listing 6.3: The SableCC generated Lexer.java file

The following solution will only solve the problem for the * part, but the rest is similar.

```
547 private static boolean last=false;
548
549 Token new0(int line, int pos) { last=true; return new TMod(line,
        pos); }
550 Token new1(int line, int pos) { last=true; return new TSlash(line,
        pos); }
551 Token new2(int line, int pos) { last=true; return new TParenL(line
        , pos); }
552 Token new3(int line, int pos) { last=false; return new TParenR(line
        , pos); }
553 Token new4(int line, int pos) { last=true; return new TBracketL(
        line, pos); }
554 Token new5(int line, int pos) { last=false; return new TBracketR(
        line, pos); }
555 Token new6(int line, int pos) { last=true; return new TAt(line, pos
        ); }
556 Token new7(int line, int pos) { last=true; return new TComma(line,
        pos); }
557 Token new8(int line, int pos) { last=true; return new TPipe(line,
        pos); }
558 Token new9(int line, int pos) { last=true; return new TOr(line, pos
        ); }
559 Token new10(int line, int pos) { last=true; return new TAnd(line,
        pos); }
560 Token new11(int line, int pos) { last=true; return new TPlus(line,
        pos); }
561 Token new12(int line, int pos) { last=true; return new TMinus(line
        , pos); }
562 Token new13(int line, int pos) { if (!last) return new14(line,pos)
        ; last=false; return new TStar(line, pos); }
563 Token new14(int line, int pos) { last=true; return new TMultistar(
        line, pos); }
564 The list goes on
```

Listing 6.4: The modified Lexer.java file

Now if the last token is not `mod`, `/` or any other tokens mentioned in the quote in the beginning of Section 6.2.2, a `MultiStar` is returned.

6.3 Visitor Pattern

SableCC uses a slightly altered version of the visitor design pattern.

The visitor design pattern is used when doing the contextual analysis, for checking whether scope and type rules are respected, and for doing the code generation.

```
1 Visitable() {
2     public void apply(Visitor);
3 }
4 public class Visitor {
5     AbstractSyntaxTree AST;
```

```
6
7       AST.getFirstNode().apply(this);
8
9       public void caseE1(Visitable v)
10      {
11          //Execute relevant code
12          v.getFirstChild.apply(this);
13          v.getSecondChild.apply(this);
14      }
15
16      public void caseE2(Visitable v)
17      {
18          //Execute relevant code
19      }
20
21      public void caseE4(Visitable v)
22      {
23          //Execute relevant code
24          v.getFirstChild.apply(this);
25          v.getSecondChild.apply(this);
26      }
27
28      public void caseE4(Visitable v)
29      {
30          //Execute relevant code
31      }
32
33      public void caseE5(Visitable v)
34      {
35          //Execute relevant code
36      }
37  }
38  public class E1 implements Visitable {
39      public void apply(Visitor v)
40      {
41          v.caseE1(this);
42      }
43  }
44  public class E2 implements Visitable {
45      public void apply(Visitor v)
46      {
47          v.caseE2(this);
48      }
49  }
50
51  // Classes E4 through E6 are identical to
52  // E1 and E2 in implementation
```

Listing 6.5: Example of the use of the Visitor Design Pattern

Each class in the AST implements the visitable interface, and therefore the class has the `apply()` method (in other implementations of the visitor pattern, the method is called `accept()`). The `apply()` method is invoked by the visitor class, which traverses the AST. The `apply()` method then calls the appropriate method in the visitor class, with the object `this` as argument. In that way it is possible to separate parsing and code generation by having a visitor class doing each task. This makes it easy to replace the code

generator.

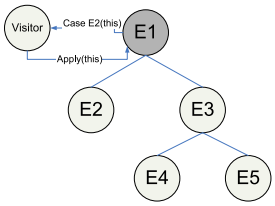

Figure 6.1: Visiting E1

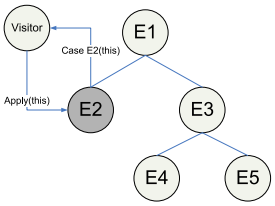

Figure 6.2: Visiting E2

Figure 6.1 and 6.2 illustrate the principle of the visitor design pattern with a simple example. The nodes in the tree all implement the visitable interface, and has an `apply()` method. Visitor calls the `apply()` method on the `E1` object, which calls the `caseE1()` method, with the object `this` as argument. Actually, this can be simplified by using Java's method overloading, and letting all nodes call the same method with `this` as argument. In this case the correct method is called, based on the type of the node object.

7 Framework

This chapter is primarily intended for providing an overview of the framework. It can be used for guidance and help when reading the implementation section.

7.1 Components

This section describes the key components in our project, which are the central class files in the design. These classes combined forms the core of the XPath interpreter.

Figure 7.1 shows a sketch of the project framework, showing the interaction between various classes.

7.1.1 The `ASTAnalysisAdapter` Class

This class is the core of the interpreter. This is the class that traverses the abstract syntax tree using the visitor pattern described in Section 6.3, determining which axes and functions to use. This is also the class that handles all calculations and selection of DOM nodes. Furthermore, the normalization from abbreviated to unabbreviated syntax is done when traversing the abstract syntax tree. All nodes are stored in node-sets which are also kept track of in this class.

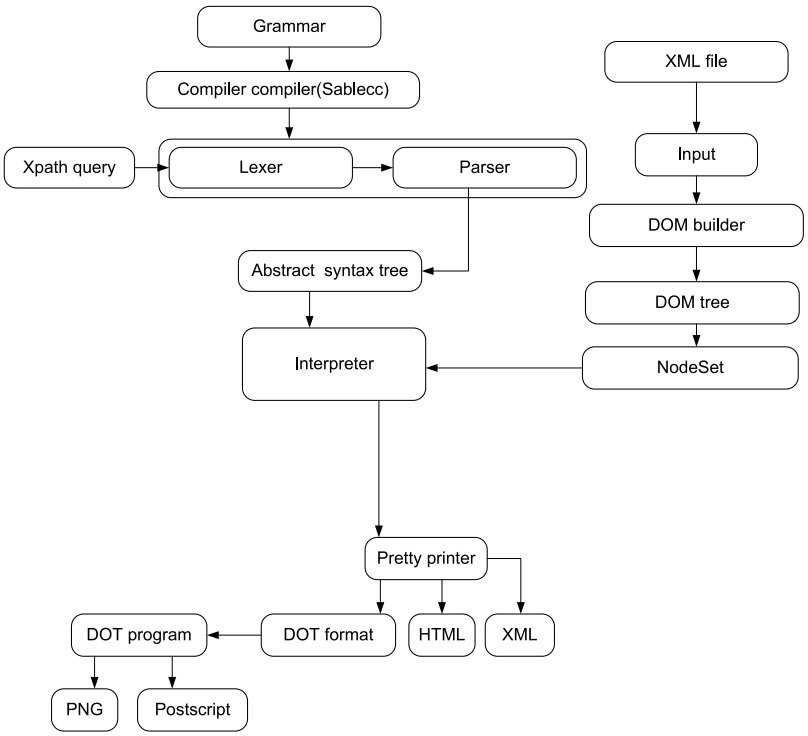

Figure 7.1: The project framework

7.1.2 The `DomTree` Class

This class handles the transformation from an XML document into a DOM tree, which is the tree that most of the other classes in the implementation will work on.

7.1.3 The `NodeSet` Class

This class is a container for DOM nodes. It contains nodes from the DOM tree in document order. It has a built-in iterator for iterating through the nodes in the node-set. This class also handles all functions specified for node-sets.

7.1.4 The `AxisSpecifier` Class

This class is primarily intended for internal use in the `ASTAnalysisAdapter`. It has a method for each axisname in the grammar. The class handles all selection of nodes selected by the axis and node test.

7.1.5 The Function Classes

There is a class for each type in XPath that handles functions related to that specific type. These are `BooleanFunctions`, `NumberFunctions`, and `StringFunctions`. These classes combined represent the most of the function library as it is specified in the XPath standard.

7.1.6 The `XPathInterpreter` Class

This is the front-end of the interpreter. The `XPathInterpreter` class is responsible for taking input, in the form of XPath expressions, from the user, and returning the result.

7.1.7 The `XPathGUI` Class

This class is actually not part of the interpreter, and could easily be left out. It is designed for pretty-printing the result of an XPath query. The class converts the result into different formats like XML, HTML, and DOT. DOT is part of the Graphviz package [1].

7.2 Class Diagram

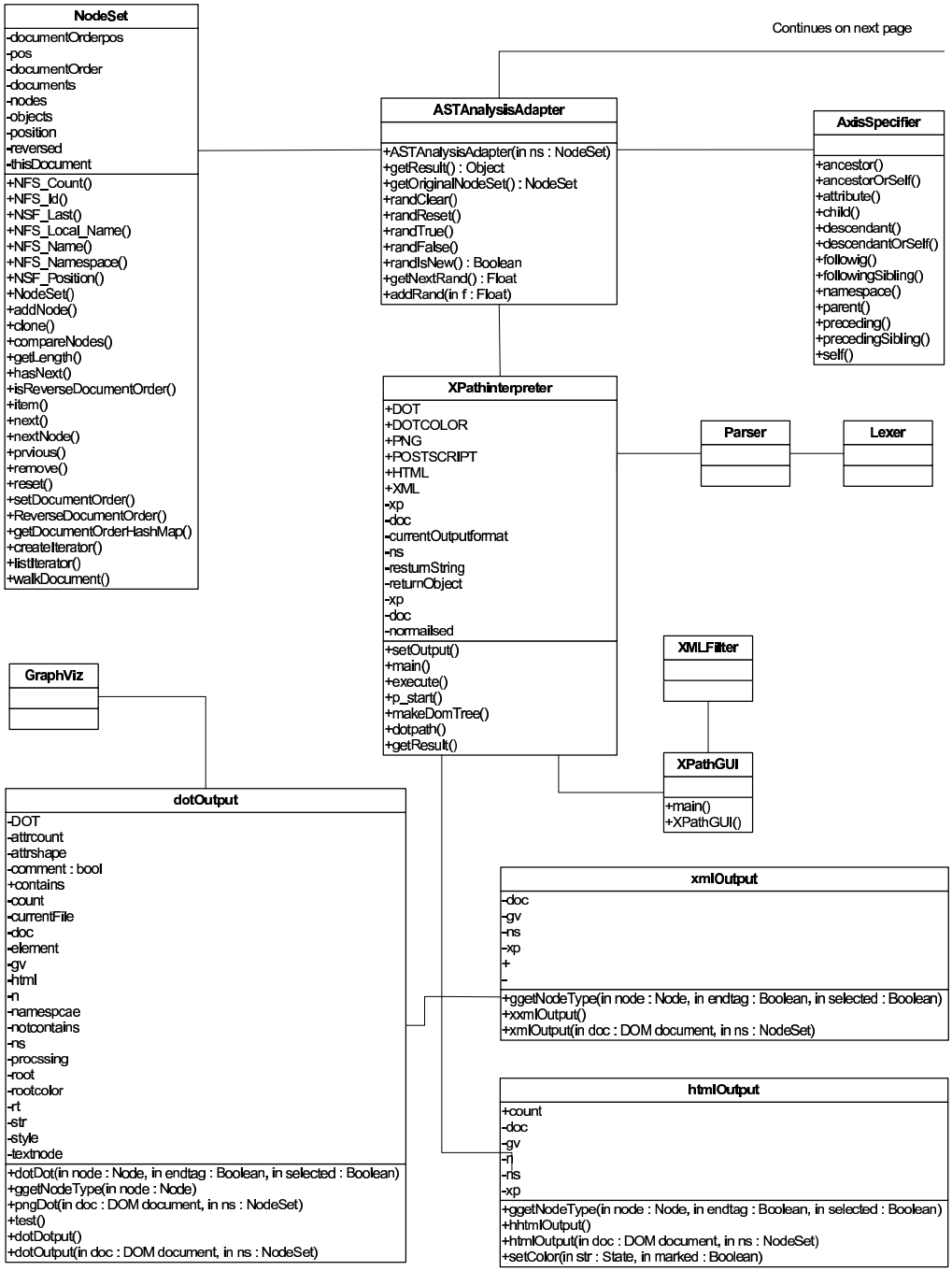

Figure 7.2: The project framework

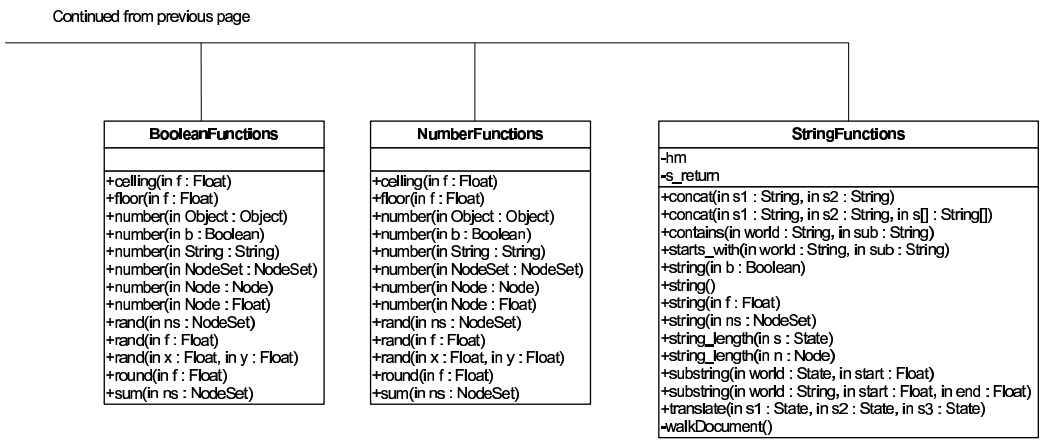

Figure 7.3: The project framework

7.3 Design Specifications

In this section we want to describe the interface of the different classes described in Section 7.1. This helps when programming in large groups, as people will easily pick up on how to use these classes.

7.3.1 `ASTAnalysisAdapter`

The `ASTAnalysisAdapter` class extends the tree-walker, `DepthFirstAdapter`, generated by SableCC. This class must be given as an argument in the `apply()` method in order to start the traversel of the AST (see Listing 5.1). This class takes in a `NodeSet` object containing the `Document` node (the root). When the traversal finishes, the result can be obtained with the `getResult()` method. If an error occurs while traversing the tree, an exception is thrown– e.g., `InvalidFunctionNameException`, `InvalidParameterException`, and `ParameterMismatchException`.

7.3.2 `XPathInterpreter`

This class initiates the whole interpretation process. It has a public method called `execute()` which takes in the XPath expression and the XML document file as arguments. This method returns the output according to what has been specified in the `setOutput()` method, e.g. XML, HTML. The

`execute()` method is only used when having other classes have to use the `XPathInterpreter` class. The `XPathInterpreter` class can also be used as an independent program, as it has a `Main()` method. Therefore it can be used in the command line by typing `java XPathInterpreter "expression" "file"`. The output is written to standard out in XML format.

7.3.3 `NodeSet`

When instantiating a `NodeSet` object, a DOM tree, in the form of a `Document` object, must be given as an argument. This class is utilized for determining document order in the document. When adding a node, the `addNode()` method is used. The class has its own built-in iterator, which includes the methods `nextNode()`, `previousNode()`, and `getLength()`. Furthermore the node-set functions, which are specified in the XPath standard, are implemented in methods with the prefix "NSF_".

7.3.4 `AxisSpecifier`

The methods of the `AxisSpecifier` class have simple interface. There is one method for each axis with the same name as the axisname. All the methods have the same syntax: The arguments are a node-set and a nodetest and the return type is a `NodeSet` object. This `NodeSet` will contain the nodes which have been selected by the axisspecifier.

III

Implementation, Test, and Conclusion

8 Implementation

Our implementation of the XPath interpreter has been concentrated around the implementation of the following classes: `NodeSet`, `AxisSpecifier`, and `ASTAnalysisAdapter`. These classes are of primary interest because they contain the core of the implementation, whereas the remaining classes consist of smaller "helper-functions" needed for full compliance to the XPath standard. In the section below we will describe the core classes to clarify our intent with regards to how the implementation was done.

8.1 `NodeSet`

There is an order of all nodes in a document called document order. This is defined as the order in which the first character of each node contained within the XML representation of the XML document appears. This means that the root node will be the first node. Elements appear before their children. Attributes and namespaces of an element appear before the children of the element.

It states clearly in the XPath standard that a node-set is an unordered collection of nodes without duplicates. But it is also defined that axes like preceding and following must return all nodes, that are respectively before and after the context node in document order. In order to maintain this document order we have expanded the `NodeSet` class so that the stored nodes will be kept in document order.

To implement this we have chosen to keep track of the of the document order at all times in the `NodeSet` class. This has the consequence that when a copy of our `NodeSet` class is instantiated, a DOM tree must be given as argument.

In the `NodeSet` constructor the whole DOM tree is traversed recursively and the nodes are stored in a hash map. Every node is stored in the hash map using the `Node` object as the key, and an integer representing the number of the node in document order as the value. Listing 8.1 shows the general idea of a method that is called from the constructor that uses a depth-first traversal to index the whole DOM tree.

In order to ensure that there are no duplicate nodes in the node-set, we have implemented a test in the `addNode()` method that scans the current node-set and determines whether an identical node is already in the node-set or not. If an identical node is found, the new node is discarded.

```
1  private void walkDocument (Node node, HashMap hm)
2  {
3    documentOrderPos++;
4    hm.put (node, new Integer (documentOrderPos));
5
6    Node child = node.getFirstChild ();
7
8    if (child != null)
9    {
10     walkDocument (child, hm);
11     while ((child = child.getNextSibling ()) != null)
12     {
13       walkDocument (child, hm);
14     }
15   }
16 }
```

Listing 8.1: This shows the general idea of how we traverse the tree to determine document order

The `walkDocument()` method is called from the `NodeSet` constructor with the document node and the hash map as arguments. The field `documentOrderPos` is a static variable that keeps track of the document order. `documentOrderPos` will increase every time the method is called, immediately after the node is put into the hash map (Line 3-4). When instantiated, the `NodeSet` object will have a complete index of the document order. When adding a node to the node-set, the node is added to the linked list of the `NodeSet` object. This linked list keeps the actual nodes in document order. It is necessary to iterate through the list to find the correct position for the node.

The `NodeSet` class also implements reversed document order, which is needed by some axisspecifiers. An example of this is the ancestor axis that must return the nodes in reversed document order. Therefore the methods `setDocumentOrder()` and `setReversedDocumentOrder()` has been implemented. When a node-set is returned from the method `ancestor()` in the `Axisspecifier` class, it is set to reversed document order. Actually the

nodes are still placed in document order in the linked list, it just reads the node from the end of the linked list and backwards.

8.2 AxisSpecifier

When traversing the abstract syntax tree, a new node-set must be generated every time an axisspecifier is encountered. In order to automate this process we have constructed a class handling all these axisspecifiers, namely the `AxisSpecifier` class. For each axisname there is a static method which takes in a `NodeSet` object and a `String` object representing the nodetest as arguments. The return types of the methods are `NodeSet` objects. These `NodeSet` objects contain all the nodes that are the result of the selection.

When calling one of the methods, the argument `NodeSet` is iterated and corresponding to the axixspecifier and node test a new node-set is selected.

Some of the axisspecifiers can be defined by other axisspecifiers. We have made use of this when implementing axisspecifiers like descendant. Listing 8.2 shows how we implemented the descendant axis using child.

```
1  public static NodeSet descendant(NodeSet nodes, String nodetest)
2  {
3    NodeSet ns = new NodeSet(nodes.getDocument());
4    NodeSet children = new NodeSet(nodes.getDocument());
5    Node node;
6
7    nodes.setDocumentOrder();
8    children = child(nodes,"node()");
9    ns.join(children);
10   while(children.getLength()>0)
11   {
12     children = child(children,"node()");
13     ns.join(children);
14   }
15   ns=nodeSetTest(ns,nodetest);
16   ns.setDocumentOrder();
17   return ns;
18 }
```

Listing 8.2: This shows how we implemented the descendant axis

When the method is called, a new node-set is selected containing all the child-nodes of the current node in the node-set (Line 8). After this is done, the descendant method is called with the new node-set as an argument (Line 12). This way the method iterates recursively through all the descendants of a context node-set.

The descendant axisspecifier is used in the implementation of other axisspecifiers such as descendant-or-self and following.

8.3 ASTAnalysisAdapter

A tree-walker is constructed by SableCC to analyse the Abstract Syntax Tree. We use this analysis to create an XPath Object tree by adding information for every *important* node in the AST. A node is considered important if it contains information that impacts the calculation an expression. We later use this information to evaluate the XPath expression and determine the result. This result is only calculated on request, when the `getResult()` method is called.

Our XPath Object, or `XObject` for short, consists of a reference to its parent and to each of its children. Furthermore the `XObject` contains an attribute that represents the *important* node, whose information is needed. The most important feature in the `XObject` is the `eval()` method. This method makes it possible to recursively evaluate the XPath expression, which is represented by the `XObject`.

By letting the tree-walker, in this case called a switch, traverse the abstract syntax tree and build an XPath Object tree, we enable the program to evaluate predicates and expressions more than once. Opposed to calculating the result while traversing the AST, this method of derivation allows for more freedom in constructing XPath expressions. An example is the `position()` function which returns the current position in the context node-set.

Expression 8 `/descendant-or-self/child::AAA[position() = 1]`

If Expression 8 is to be evaluated correctly, the predicate will have to be evaluated for each node in the node-set corresponding to the node-set returned by the axes, which is seen in Expression 9.

Expression 9 `/descendant-or-self/child::AAA`

By building an `XObject` with just enough information to evaluate the expression, we can dynamically calculate the predicate for each node. In the example above the predicate `[position() = 1]` will evaluate to true only for the node with the specific position of 1, thus only this node is added to the final result.

Every analysis involves a number of stacks to keep track of the results.

- `axisStack`
- `nodeSetStack`

- `exprStack`
- `xStack`

The `nodeSetStack` is used to keep track of each node-set generated and manipulated. AxisSpecifiers and node tests used to determine these node-sets are stored in the `axisStack`. The `exprStack` holds objects of the `ExprObject` type, which are containers for the root node of an XPath Object tree. The most important stack is the `xStack`. This is the primary storage, and where every computation result is stored along the recursive evaluation.

8.4 Documentation

In order to make it easier for people using the different components of the interpreter, documentation has been made for methods and classes. As java is the choice of compiler it comes with a tool for automating the process of making documentation. This tool is called Javadoc[20] and it generates HTML-based documentation from the java source files.

8.5 Unimplemented features

As the implementation of DOM that is used for the interpreter is DOM level 1[11], there is a node-set operation that is not implemented. The method in question is `id()`, which was not introduced until DOM level 2, and therefore not available. For this reason the method has been left out of the implementation of YAXi, as we decided that implementing this feature would be too time consuming.

9 Test

Writing tests is a very important yet often misunderstood task. Tests assists the developers in verifying bugs and/or patches. Writing the perfect test cases for specific uses is extremely vital, but this is often a very time consuming task and sometimes quite difficult. It may seem like a lot of work, but writing tests will actually save development time.

9.1 Test Strategy

The purpose of this section is to describe how the tests of our interpreter were performed, and to argue for the testprocedure. The tests of YAXi were divided into three parts. The description of the tests can be found in Section 9.5. The tests were performed with a tool we developed, which is described later in this section.

Testing the XPath interpreter involves testing if a given expression evaluates as described in the XPath standard.

For testing purposes we developed a graphical user interface and a command line-based tool.

The graphical tool is used for testing one XPath expression at a time, and the result can be presented in six different ways:

XML Shows the part of the original XML document, which has been selected with the XPath expression, as a new XML document.

DOT-report Shows the generated DOT-file.

```
1<1                              &  false
concat("titi","toto")            &  tititoto
normalize-space("wer   den ")    &  wer den
```

Figure 9.1: Example of an input file

PNG Shows the whole XML document as a tree with the selected parts emphasized.

DOT-test Shows the whole XML document as a tree with the selected parts emphasized with colors.

PostScript Generates a PostScript-file for use in this report. The XML document is shown as a tree with the selected parts emphasized. The file is generated in grayscale.

HTML Shows the whole XML document, with the selected parts marked with the color red.

The command line tool is practical for testing several XPath expressions at a time. It reads from a file which contains the expressions to test in the XPath interpreter along with the results which the expression should return. The file has a format which makes it easy to include in this report. An example of such a file, with simple expressions, can be seen in figure 9.1.

The testing will be done using the method illustrated in figure 9.2. A line is read from the input file, and the expression is then evaluated. The result from the XPath interpreter is compared to the expected result. When all expressions in the input file have been evaluated, the result is printed to the screen as shown in figure 9.3. Each expression is marked with either "Correct" or "Incorrect".

The expressions in the input file have been chosen to make sure that the the whole XPath interpreter is tested and insures that the interpreter accepts any well-formed XPath expression and that it returns the correct result. Some of the test data may look redundant, e.g. $1 > 0$ and $0 < 1$, which should both evaluate to true. Logically the two comparisons are equal, but both possibilities have to be tested because they are not evaluated in the same way in the XPath interpreter. Therefore a lot of expressions that look the same have to be tested.

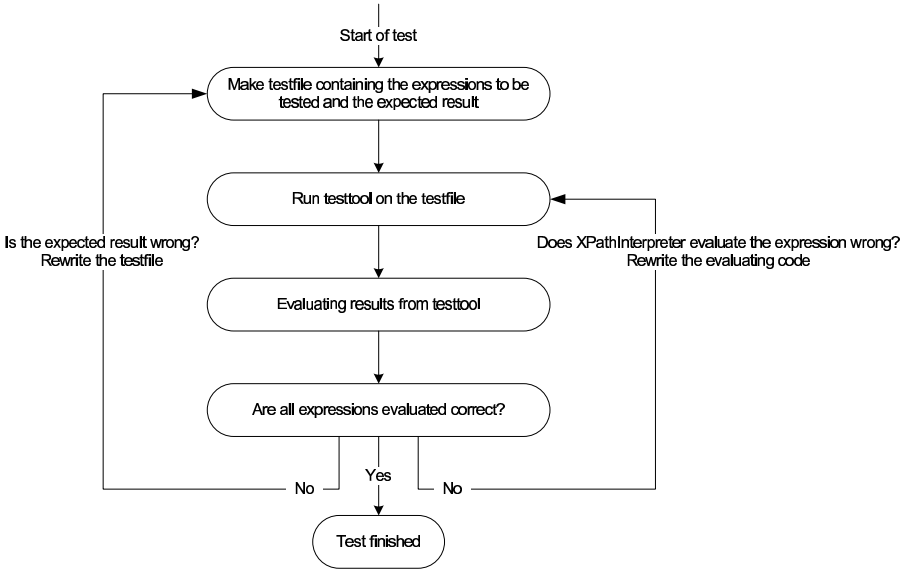

Figure 9.2: Testing cycle

Input to test tool
`//*/namespace::*[position()=1]`
Output from test tool
Correct

Figure 9.3: The input and output of the test tool used in the test phase

9.2 Test Examples

Our test consists of several hundred expressions and their expected results written in a file. All test results can be seen on the homepage at "www.cs.auc.dk/∼fr0/". To insure that the node-set functions `last()` and `position()` perform as described in the XPath standard, a series of expressions containing those functions were used to cross-check the functions.

In the following example all whitespaces have been removed from the XML output of YAXi.

Input
```
count(//E9/descendant-or-self::*)
```
Output
```
<number>5.0</number>
```

Figure 9.4: An expression and its result

With the knowledge of the number of nodes in the node-set, the nodeset `position()` function can be used to select the fifth node in the node-set.

Input
```
//E9/descendant-or-self::*[position()=5]
```
Output
```
<nodeset><Element><E12/></Element></nodeset>
```

Figure 9.5: An expression and its result

Now the idea is to use the node-set function `last()` to select the last node in the node-set, and then compare it to the XML output. If the outputs of Figure 9.5 and Figure 9.6 are the same, then the `last()` function on the node-set works as expected.

Input
```
//E9/descendant-or-self::*[last()]
```
Output
```
<nodeset><Element><E12/></Element></nodeset>
```

Figure 9.6: An expression and its result

9.3 Test of Design Criteria

The main design criteria was to implement the whole XPath standard. To test this, we had to develop test material which represented as much as possible of the XPath standard. We performed extensive testing by executing several hundreds of expressions and evaluating the results according to the described test strategy. The test of YAXi has been divided into three parts. Tests and their results can be found the homepage specified earlier in this chapter.

The first test Simple calculations and function calls which can be performed without the use of a node-set are tested.

The second test A location step without predicates is tested.

The third test This test adds the predicate to the location path. The test has the purpose to ensure that the combining of function calls, operators and different parts of a location step works.

The reason for dividing the test of YAXi into three parts was to increase the understanding of a given error found during the tests. The reason for picking this procedure was to ensure that a mistake in a function call or one of the operators would not affect the result of later testing of the location path.

9.4 Evaluation of Test

Testing is an important part of our project, considering we have decided to implement the whole XPath standard. Therefore it is essential to prepare a set of tests which test all parts of an XPath interpreter.

During the test period we focused on expressions which did not yield the expected result. Each time we reconsidered the expected result, and found that sometimes we had expected an incorrect result. Other tests returned the expected result, and found the expression giving an incorrect result. We used these observations to correct the errors in the code that caused the interpreter to fail. After changing the code, the tests were carried out again. All tests were carried out in each iteration in the testing cycle, to ensure we did not make changes to the code, which would result in incorrect results.

Using the test method as described, we have ensured that our XPath interpreter generates correct results according to the XPath specification. We have also shown that we have met the demands of the design criteria.

9.5 Test Conclusion

During the tests the errors listed below were found. These errors have now been corrected. The tests and results can be found on the homepage specified at the beginning of this chapter.

- Following-sibling axis executes an infinite loop.
- The boolean function `lang()` accepts a node and a string as arguments, it should only have taken a string.
- The backwards axis did not work because of an error in the nodeset.
- The equality comparison between strings always returned true
- Negative numbers were handled as positive numbers. $1 + -2$ would evaluate to 3

First test More than 250 tests have been performed. All tests met the expected results. This proves that YAXi is able to evaluate simple calculations and function calls.

Second test Each of the axes have been tested twice, each with a different starting point. Similar to the axis, the node test have also been tested. Both the axes and the node test work as expected.

third test This tests if YAXi is able to handle one or more predicates. In the expressions both function calls and calculations were mixed to ensure that YAXi was able to deal with all kinds of expressions in the predicates. 20 tests were performed, with as many as four predicates, and YAXi performed as expected.

10 Study Report

10.1 Study Report

The purpose of this section of the report is to reflect upon the work methods used during the project period. This part of the report is not to be considdered scientific material, but rather a guide to the cencor and supervisor as to how the project has come to be.

10.2 Analysis

At the first stage of our project we spent a relatively long time evaluating technologies and reading the XPath standard. As the focus of this project is to implement an already known standard, we saw the understanding and discussion of unclear parts of the standard as a very important part of this stage, hence the relative large amounts of time allocated to this task. When choosing a compiler compiler we had to determine which technology we wanted to make use of, i.e. if we wanted to use an LL(1) or an LALR(1) parser. In the end we decided that SableCC was the best choice. At this stage, we also had to discuss the different APIs for XML parsing and which programming language to choose for implementing the interpreter.

10.3 Design

The first major design issue was massaging the grammar. This proved to be a slow process due to very lacking documentation on SableCC. Because the

grammar is such an important part of the project, the whole group wanted as much insight in this process as possible, so everyone participated in this process. It proved to be an even more complicated process than we had assumed, as the grammar turned out not to be LALR(1) parsable, which had to be solved (see Section 6.2).

At this stage we also had to define the framework of our implementation, which was rather easy, as the XPath standard specifies many of the components of an implementation. At the end of the design stage we had worked out a document specifying the interfaces of all the components.

10.4 Implementation

As the interface of all components had been specified in the design stage of the project, the implementation part of most of the components was relatively straight-forward.

The programming process itself was split into subtasks:

NodeSet The task was to implement the node-set as it is described in the XPath standard, including all node-set functions

ASTAnalysisAdapter The task was to implement the tree-walker that walks the abstract syntax tree, performing the calculations and keeping track of the node-sets.

Axisspecifier The task was to implement the axisspecifiers as they are specified in the XPath standard.

Function libary The task was to implement the whole function library including the booleanfunctions, numberfunctions and stringfunctions.

GUI/front end Implementation of the front end of the interpreter. This task also include implementing a GUI for the interpreter.

As our group consists of seven people, the possibility of making parallel programming processes is present. Therefore we split our group into tree small subgroups which take care of their own subtask.

This approach has the advantage that there quickly is progress in the implementation progress, which means that the components can be tested together rather early, and eventual errors in the framework can be detected in the beginning of this stage. Another advantage with this approach is that members

of the subgroup can supply each other when problems occur. One of the disadvantages with this approach is that it is only the subgroup responsible for a component who has insight in this component. In order to reduce this problem, we used the built-in feature in java called Javadoc in order to document the functionality and interface of the different components.

10.5 Testing

As the focus is to implement a known standard, a test process is important to determine whether all features of XPath has been implemented correctly.

We performed white and black box testing during the development, because we used a hard coding approach during this project. Everything uploaded to our CVS repository must compile without errors. We did this to make sure people could count on work from group members to be correct and usable.

Some of the testing was done while developing, most of it being white box tests. We made a lot of effort ensuring the correctness of the results of the internal functions. Once the program was near completion we began blackbox testing with querries we knew the results of. This should give us an idea about the correctness of the program.

10.6 Conclusion

Looking back at the project as a whole, we conclude that it has progressed satisfactorily, even though some stages of the project development took a lot longer than initially scheduled. Especially the massaging of the grammar turned out to be more time demanding than expected. The implementation also demanded a lot of attention, but as we split the group into subgroups this process was completed relatively painlessly. We found this split into subgroups to be a very useful style of programming, so we continued to use this throughout the whole writing process.

11 Conclusion

11.1 Further development

Even though the whole XPath standard has been implement, there are still many things that could be done in order to improve on the implementation. One of the things that one could improve on is the execution time. For example the task of maintaining document order in a node-set is rather expensive. Actually, the time complexity in the worst case when adding n nodes to the node-set is $O(n!)$. It could have been implemented using for example, a quick sort, which has the time complexity of $O(n * log(n))$. This would be a big improvement on the execution time. As execution time has not been the main focus of this project, there are many other places where a rewrite of the code could successfully improve on the execution speed.

Looking at further development, this project could be the first step towards implementing languages like XSLT[9] or XQuery[13], which both extend the functionality of the XPath language.

11.2 Conclusion

As we have shown throughout this project, we have developed a clear understanding of building a useful interpreter as well as an understanding of web technologies like XML, DOM, and XPath.

We have managed to implement a full grown XPath 1.0 interpreter, with the only exception beeing the function `id()`, that is not supported by the DOM level 1 API, which is used in YAXi.

The development of the interpreter was aided by using the tool SableCC, which has increased the flexibility of the development of the interpreter. The formal XPath grammar did require some massaging to transform it into the EBNF required by SableCC, including a small rewrite of the lexer, while still maintaining a fully compatible language syntax.

There has been added a small, but yet powerful, improvement to the XPath function core: `rand()`, which enables non-deterministic selection of nodes.

We have used the test chapter to show, that we have built a fully working implementation of the XPath language. Perhaps speed could be improved on, but in many real-life applications, like RSS-feeds, speed is no problem, because of other overheads.

In overall we can conclude that we have managed to develop an XPath interpreter, YAXi, that conforms to the requirements specified by both the XPath standard as well as our own requirements.

Bibliography

[1] AT&T. Graphviz. http://www.research.att.com/sw/tools/graphviz/, 2003.

[2] Ben Berck. Creating efficient msxml applications. http://www.xml.com/pub/a/2002/03/06/efficient.html, 2002.

[3] Elliot Joel Berk and C. Scott Ananian. Jlex: A lexical analyzer generator for java(tm). http://www.cs.princeton.edu/ appel/modern/java/jLex/, 2003.

[4] David A Watt & Deryck F Brown. Programming lanuage processors in java, page 118-124, 2000.

[5] David A Watt & Deryck F Brown. Programming lanuage processors in java, page 83-93, 2000.

[6] David Brownell. About sax. http://sax.sourceforge.net/.

[7] World Wide Web consortium. Html 4.01 specification. http://www.w3.org/TR/html401/, 1999.

[8] World Wide Web consortium. Xml path language (xpath) version 1.0. http://www.w3.org/TR/xpath, 1999.

[9] World Wide Web consortium. Xsl transformations (xslt) version 1.0. http://www.w3.org/TR/xslt, 1999.

[10] World Wide Web consortium. Xml pointer language (xpointer). http://www.w3.org/XML/Linking, 2000.

[11] World Wide Web consortium. W3c document object model. http://www.w3.org/DOM, 2002.

[12] World Wide Web Consortium. W3c in 7 points. http://www.w3.org/Consortium/Points/, 2003.

[13] World Wide Web consortium. Xquery 1.0: An xml query language. http://www.w3.org/TR/xquery/, 2003.

[14] World Wide Web Consortium. Extensible markup language (xml) 1.0 (third edition). http://www.w3.org/TR/2004/REC-xml-20040204/, 2004.

[15] International Organization for Standardization (ISO). Iso14977. http://www.cl.cam.ac.uk/ mgk25/iso-14977.pdf, 1996.

[16] Etienne Gagnon. Sablecc, an object-oriented compiler framework. http://www.sablecc.org/thesis/thesis.php, School of Computer Science McGill University, Montreal, 1998.

[17] Lars Marius Garshol. Bnf and ebnf: What are they and how do they work? http://www.garshol.priv.no/download/text/bnf.html, 2003.

[18] Stephen C. Johnson. Yacc: Yet another compiler-compiler. http://www.cs.utexas.edu/users/novak/yaccpaper.htm, 1996.

[19] Sun Microsystems. Java api for xml processing (jaxp) version x.xx. http://java.sun.com/xml/jaxp.

[20] Sun Microsystems. Ieee standard for binary folating-point arithmetic. http://java.sun.com/j2se/javadoc/, 1994-2004.

[21] Institute of Electrical and Electronics Engineers. Ieee standard for binary folating-point arithmetic, 1985.

[22] PerfectXML. Xml path language (xpath). http://www.perfectxml.com/ph/xml-htp_11.pdf, 2002.

[23] Erik T. Ray. *Learning XML*. O'REILLY, 2001.

[24] C. Scott Ananian Scott Hudson, Frank Flannery. Cup parser generator for java. http://www.cs.princeton.edu/ appel/modern/java/CUP/, 1999.

[25] Michael Sipser. Introduction to the theory of computation, 1997.

All URLs are valid as of May 24th.

IV

Appendices

XPath Grammar

```
Helpers
  letter = ['A'..'Z'] | ['a'..'z'] | [0x7F .. 0xFF];
  digit = ['0'..'9'];
  all = [0 .. 0xFF];


Tokens
  mod = 'mod';
  slash = '/';
  paren_l = '(';
  paren_r = ')';
  bracket_l = '[';
  bracket_r = ']';
  at = '@';
  comma = ',';
  pipe = '|';
  or = 'or';
  and = 'and';
  plus = '+';
  minus = '-';
  star = '*';
  multistar = '*';
  div = 'div';
  dollar = '$';
  blank = 13 | 10 | 9 | ' ';

  abbreviatedstep = '.' | '..';

  nodetype = 'comment'
           | 'text'
           | 'node';

  processing_instruction = 'processing-instruction';

  axisname = 'ancestor'
           | 'ancestor-or-self'
           | 'attribute'
           | 'child'
           | 'descendant'
           | 'descendant-or-self'
           | 'following'
```

```
42              | 'following-sibling'
43              | 'namespace'
44              | 'parent'
45              | 'preceding'
46              | 'preceding-sibling'
47              | 'self';
48
49    colon = ':';
50    underscore = '_';
51    doublecolon  = '::';
52    doubleslash  = '//' ;
53    equality = '=' | '!=' ;
54    relation = '<=' | '>=' | '<' | '>' ;
55
56    literal = ''' ([all - '''])*  ''' | '"' ([all - '"'])*  '"';
57    number = digit+ ('.' digit+)? | '.' digit+;
58
59    ncname = (letter | '_') (letter | '_' | digit | '.' | '-')*;
60
61  Ignored Tokens
62      blank;
63  Productions
64    start =  expr;
65
66  //Location Paths
67    locationpath = {relativelocationpath} relativelocationpath
68                 | {absolutelocationpath} absolutelocationpath;
69
70    absolutelocationpath = {abbreviatedabsolutelocationpath}
          abbreviatedabsolutelocationpath
71                         | {relativelocationpath} slash
                               relativelocationpath?;
72
73
74    relativelocationpath = {abbreviatedrelativelocationpathslash}
          abbreviatedrelativelocationpathslash? stepslash* step
75                       | {abbreviatedrelativelocationpath}
                             abbreviatedrelativelocationpath;
76
77    abbreviatedrelativelocationpathslash =
          abbreviatedrelativelocationpath slash;
78
79    stepslash = step slash;
80
81  //Location steps
82     step = {axisspecifier} axisspecifier nodetest predicate*
83          | {abbreviatedstep} abbreviatedstep;
84
85     axisspecifier = {axisname}axisname doublecolon
86                   | {abbreviatedaxisspecifier}
                        abbreviatedaxisspecifier;
87
88  //Axes
89
90     nodetest  = {nametest} nametest
91               | {nodetype} nodetype paren_l paren_r
92               | {processing} processing_instruction paren_l literal
                    ? paren_r ;
93
94     predicate = bracket_l predicateexpr bracket_r;
95
96     predicateexpr = expr;
```

```
97
98  //abbreviations
99
100     abbreviatedabsolutelocationpath = doubleslash
            relativelocationpath;
101
102     abbreviatedrelativelocationpath = relativelocationpath
            doubleslash step;
103
104     abbreviatedaxisspecifier = at?;
105
106     expr = orexpr;
107
108     primaryexpr = {variablereference} variablereference
109                 | {expr} paren_l expr paren_r
110                 | {literal} literal
111                 | {number} number
112                 | {functioncall} functioncall;
113
114     functioncall = functionname paren_l argumentlist? paren_r;
115
116     argumentlist = {argument} argument
117                  | {argumentlist} argument comma argumentlist;
118
119     argument = expr;
120
121     unionexpr = {pathexpr} pathexpr | {unionexpr} unionexpr pipe
            pathexpr;
122
123     pathexpr = {locationpath} locationpath
124              | {filterexpr} filterexpr
125              | {filterexprslashrelativelocationpath} filterexpr
                     slash relativelocationpath
126              | {filterexprdoubleslashrelativelocationpath}
                     filterexpr doubleslash relativelocationpath;
127
128     filterexpr = {primaryexpr} primaryexpr
129                | {filterexpr} filterexpr predicate;
130
131     orexpr = {and} andexpr
132              |{or} orexpr or andexpr;
133
134     andexpr = {equal} equalityexpr
135              | {and} andexpr and equalityexpr;
136
137     equalityexpr = {relationalexpr} relationalexpr
138              | {equalityexpr} equalityexpr equality relationalexpr;
139
140     relationalexpr = {additiveexpr} additiveexpr
141              | {relationalexpr} relationalexpr relation additiveexpr
                     ;
142
143     additiveexpr = {multiplicativeexpr} multiplicativeexpr
144              | {addplusexpr} additiveexpr plus multiplicativeexpr
145              | {addminusexpr} additiveexpr minus multiplicativeexpr;
146
147
148     multiplicativeexpr = {unaryexpr} unaryexpr
149              | {multitimeseexpr} multiplicativeexpr multistar
                     unaryexpr
150              | {multimodexpr} multiplicativeexpr mod unaryexpr
151              | {multidivexpr} multiplicativeexpr div unaryexpr;
```

```
152
153     unaryexpr = minus* unionexpr;
154
155 //ExpressionLexicalStructure
156     nametest = {star} star
157              | {ncname} ncname colon star
158              | {qname} qname;
159
160   functionname = qname;
161
162   qname = ncnamecolon? ncname;
163   ncnamecolon = ncname colon;
164
165   variablereference = dollar qname;
```

B Formal XPath Grammar

Location Path

[1] LocationPath ::= RelationalLocationPath
| AbsoluteLocationPath

[2] AbsoluteLocationPath ::= '/' RelativeLocationPath?
| AbbreviatedAbsoluteLocationPath

[3] RelativeLocationPath ::= Step
| RelativeLocationPath '/' Step
| AbbreviatedRelativeLocationPath

[4] Step ::= AxisSpecifier NodeTest Predicate*
| AbbreviatedStep

[5] AxisSpecifier ::= AxisName '::'
| AbbreviatedAxisSpecifier

Axes

[6] AxisName ::= 'ancestor'
| 'ancestor-or-self'
| 'attribute'
| 'child'
| 'descendant'
| 'descendant-or-self'
| 'following'
| 'following-sibling'
| 'namespace'
| 'parent'
| 'preceding'
| 'preceding-sibling'
| 'self'

NodeTest

[7] NodeTest ::= NameTest
| NodeType '(' ')'
| 'processing-instruction' '(' Literal ')'

Predicates

[8] Predicate ::= '[' PredicateExpr ']'

[9] PredicateExpr ::= Expr

Abbreviated Syntax

[10] AbbreviatedAbsoluteLocationPath ::= '//' RelativeLocationPath

[11] AbbreviatedRelativeLocationPath ::= RelativeLocationPath '//' Step

[12] AbbreviatedStep ::= '.' | '..'

[13] AbbreviatedAxisSpecifier ::= '@'?

Expressions

[14] Expr ::= OrExpr

[15] PrimaryExpr ::= VariableReference
| '(' Expr ')'
| Literal
| Number
| FunctionCall

Function Calls

[16] FunctionCall ::= FunctionName '('(Argument(',' Argument)*)? ')'

[17] Argument ::= Expr

[18] UnionExpr ::= PathExpr
| UnionExpr '|' PathExpr

[19] PathExpr ::= LocationPath
| FilterExpr
| FilterExpr '/' RelativeLocationPath
| FilterExpr '//' RelativeLocationPath

[20] FilterExpr ::= PrimaryExpr
| FilterExpr Predicate

Booleans

[21] OrExpr ::= AndExpr
| OrExpr 'or' AndExpr

[22] AndExpr ::= EqualityExpr
| AndExpr 'and' EqualityExpr

[23] EqualityExpr ::= RelationalExpr
| EqualityExpr '=' RelationalExpr
| EqualityExpr '!=' RelationalExpr

[24] RelationalExpr ::= AdditiveExpr
| RelationalExpr '<' AdditiveExpr
| RelationalExpr '>' AdditiveExpr
| RelationalExpr '<=' AdditiveExpr
| RelationalExpr '>=' AdditiveExpr

Numbers

```
[25]  AdditiveExpr       ::=  MultiplicativeExpr
                            |  AdditiveExpr '+' MultiplicativeExpr
                            |  AdditiveExpr '-' MultiplicativeExpr

[26]  MultiplicativeExpr ::=  UnaryExpr
                            |  MultiplicativeExpr MultiplyOperator UnaryExpr
                            |  MultiplicativeExpr 'div' UnaryExpr
                            |  MultiplicativeExpr 'mod' UnaryExpr

[27]  UnaryExpr          ::=  UnionExpr
                            |  '-' UnaryExpr
```

Strings

```
[28]  ExprToken          ::=  '(' | ')' | '[' | ']'
                            |  '.' | '..' | '@' | ',' | '::'
                            |  NameTest
                            |  NodeType
                            |  Operator
                            |  FunctionName
                            |  AxisName
                            |  Literal
                            |  Number
                            |  VariableReference

[29]  Literal            ::=  '"' [ ^ " ]* '"'
                            |  "'" [ ^ ' ]* "'"

[30]  Number             ::=  Digits ('.' Digits?)?
                            |  '.' Digits

[31]  Digits             ::=  [0-9 ]+
```

[32] Operator ::= OperatorName
| MultiplyOperator
| '/'
| '//'
| '|'
| '+'
| '-'
| '='
| '!='
| '<'
| '<='
| '>'
| '>='

[33] OperatorName ::= 'and' | 'or' | 'mod' | 'div'

[34] MultiplyOperator ::= '*'

[35] FunctionName ::= QName - NodeType

[36] VariableReference ::= '$' QName

[37] NameTest ::= '*'
| NCName ':' '*'
| QName

[38] NodeType ::= 'comment'
| 'text'
| 'processing-instruction'
| 'node'

[39] ExprWhitespace ::= S