An XPath Query Evaluator for Filesystems
EECS 433 Final Project

Stephen Brennan
smb196@case.edu

December 8, 2016

Abstract

XPath is a query language designed for addressing the nodes of an XML file. However, its power in addressing and querying tree-structured data makes it applicable to other types of data. In this project, we describe our implementation of one such application: DPath. DPath is a tool that allows users to search for files with XPath expressions.

1 Introduction

eXtensible Markup Language (XML) is a standardized data format for sharing and storing structured data. It bears a close resemblance to Hypertext Markup Language (HTML), the fundamental markup underlying the web. This is because both languages are based on Structured Generalized Markup Language, a "meta-language" for defining documents that contain "markup" [1]. "Markup" can be thought of as a way to annotate data, creating structure. HTML uses markup to describe how content should be formated. XML uses markup to define logical structure of data. The fundamental unit of structure in XML is a tag:

```
<Student></Student>
```

Tags may have textual attributes associated with them:

```
<Student id="1234" name="Stephen␣Brennan"></Student>
```

Tags can contain textual data as well as other tags:

```
<Student id="1234" name="Stephen␣Brennan">
  <Course id="EECS␣433">Database Systems</Course>
  <Course id="MATH␣303">Number Theory</Course>
  <Course id="EECS␣651">Master&apos;s Thesis</Course>
</Student>
```

XML tags, attributes, and data form a tree structure. XPath is a query language that can be used to query and address each part of the XML tree [2]. It achieves this using *path expressions*. For example, the path expression `//Student/Course` describes all `Course` nodes whose parent is a `Student` node. XPath allows for even more fine-grained queries through the use of predicates, as well as several different *axes*, which express different "familial" relationships among nodes, including ascendant and descendant relations. All of these concepts will be discussed at length in Section 2.

Since XPath is a language for querying the XML tree structure, it could also be applicable to other tree structures. One common tree structure encountered in everyday computing is the file system. Most modern file systems are organized into a tree where internal nodes are directories and leaf nodes are files (or empty directories). Like XML, each file and directory has a name and attributes. However, there are some important differences between file systems that should be recognized.

- Unlike XML, a fully qualified file path uniquely identifies a file, whereas a fully qualified tag path in XML may identify a whole set of nodes. This is because there may be several sibling tags with the same name in XML.
- File systems may contain *links* which point to other parts of the system, or even to a parent, creating cycles. For the purpose of this paper, we will ignore these links. In this project, we use the `lstat()` system call, which treats links as files and does not follow them.

1.1 Problem Definition

For this project, our goal was to implement a tool which evaluates XPath queries on the filesystem. We have achieved this goal. Our contribution is a program which evaluates filesystem queries in a language nearly identical to XPath, called DPath. This tool runs from the command line, accepts a query as its sole argument, and outputs one result per line on standard output. An example query is presented below.

```
> dpath './*[starts-with(name(.), name(..))]'
file:/home/stephen/go/src/github.com/brenns10/dpath/dpath.nn.go
file:/home/stephen/go/src/github.com/brenns10/dpath/dpath.y
file:/home/stephen/go/src/github.com/brenns10/dpath/dpath
file:/home/stephen/go/src/github.com/brenns10/dpath/dpath.nex
```

This query returns files from the DPath source directory which start with the name of their containing directory.

The remainder of this report will be organized as follows. Section 2 will give a detailed description of the XPath query language's syntax and semantics. Section 3 will describe the implementation of DPath. Section 4 will discuss work related to this project, and Section 5 will future improvements to this project.

2 XPath

The XPath language is standardized by the World Wide Web Consortium (W3C), and it is embedded within the XQuery and XSLT languages, which are also W3C standards [2]. In the research and development of DPath, we focused on the 2.0 version of the XPath standard. There are several parts to the XPath standard: syntax[2], semantics[3], data model, type system[4], operators, and built-in function library[5].

2.1 Syntax

XPath is a context-free grammar consisting of a few types of lexical tokens: literals, names, and symbols. Numeric literals can be expressed in integer, decimal, and scientific notations, while string literals are enclosed in single or double quotes. The only character escaped within a string literal is the delimiter itself. When the string delimiter occurs within the string, it is repeated twice. For example, the string literal `'Madam I''m Adam'` is equivalent to the string `Madam I'm Adam`.

The XPath syntax for expressing identifiers is the QName, which stands for a "qualified name" [6]. A qualified name is a pair consisting of a "namespace name," which is a URI, and a "local name". The namespace URI is bound to a small prefix which is then used in the QName. For instance, the following prefixes are used throughout the XPath specification[2]:

- `xs = http://www.w3.org/2001/XMLSchema`
- `fn = http://www.w3.org/2005/xpath-functions`
- `err = http://www.w3.org/2005/xqt-errors`

With these prefixes, a QName like `xs:integer` would refer to the local name `integer` in the XML Schema namespace, while `fn:count` would refer to the local name `count` within the XPath Functions namespace. QNames may be represented without a namespace name, in which case the name is assumed to come from the default namespace. The local part of a QName must start with a letter or underscore (or a selection of other Unicode code points). The remainder of the name may additionally contain hyphens, periods, or digits, but no whitespace.

Whitespace is not significant in XPath, except in a few cases where it separates tokens [2]. For example, whitespace is required to separate a QName from a numeric literal. Therefore, an expression such as `Student/Course` is equivalent to `Student / Course`, but `Student123` is not equivalent to `Student 123` (which is not syntactically valid).

The XPath grammar is familiar to those who have implemented an expression language. It resembles a simple expression grammar, with different non-terminals representing each "level" of operator precedence. The slash operator, used to create path expressions, has higher precedence than all of the arithmetic operators. Most of the details of the grammar are not relevant to our discussion, but may be found in EBNF form in the XPath Specification [2].

2.2 Data Model

XPath is built on some fundamental concepts that should be well understood in order to implement it. First, the data model. XPath's data model is made up of Items and Sequences [4]. An Item is defined as any single value, which could be an XML element, or an atomic value such as a number or string. The types of these items are represented as QNames. There are several types defined in the XPath specification, most of which are not relevant to our discussion.

A Sequence contains zero or more Items, which may be of heterogeneous types [4]. Sequences are flat, meaning they may not contain other sequences. There is no distinction between an Item and a Sequence containing a single Item (commonly referred to as a singleton). Therefore, every XPath expression returns a Sequence.

XPath represents XML documents with a tree of nodes (each of which is an Item) that must be constructed prior to evaluating a path expression. This parsed document tree includes nearly every part of the document. Elements are nodes, but so are attributes, comments, and text enclosed within an element [4].

2.3 Semantics

When an XPath expression is evaluated, several pieces of information are known by the engine and kept as "context." The pieces of information that are known only during evaluation of an expression are called the "dynamic context", and are critical for understanding the semantics of path expressions [2]. In particular, the dynamic context includes a "context item", which is an Item (as defined in the data model) that is currently being processed. The context also includes the "context position", which contains the index of the context item within its containing sequence.

With an understanding of the context item, we can begin to describe the semantics of a Path expression. A path expression can be represented as a sequence of *steps*. For instance, the path `Student/Course` has two steps, `Student` and `Course`. A step is made up of two parts: an *axis* and a *test*. The axis can be thought of as the "direction" in which to take a step, relative to the context object. The test selects certain types or names of nodes from the axis. The axis and test are represented together with the syntax `axis::test`. When the axis is not provided, it is assumed to be the child axis. Therefore, the `Student` step will yield a sequence of nodes that are children of the context item and have name `Student`. The subsequent `Child` step is applied to each item yielded by the previous step. That is, for every item in the output of the previous step, the context item is set to this item, and the `Child` step is evaluated, yielding nodes which are children of the `Student` node which also have the name `Child`. The result of a path expression is a Sequence that contains every result of the last step of the path.

2.3.1 Axes

Several axes exist on which to write paths. The most common is the `child` axis, which we have seen. The next is the `parent` axis, which returns the node containing the context item. Another common axis is the `descendant` axis, which contains every item which is a child of the context item, or a child of its children, etc. It can be thought of as the "transitive closure" of the `child` axis. Similarly, the `ancestor` axis can be thought of as the "transitive closure" of the `parent` axis, yielding the parent and every subsequent parent of the context item. Finally, since the context item is not included in the `descendant` or `ancestor` axes, there are additionally the `descendant-or-self` and `ancestor-or-self` axes, which include the context item. XPath additionally defines a `self` axis as well as `following`, `following-sibling`, `preceding`, and `preceding-sibling`, which are less relevant to our upcoming discussion on DPath. Finally, the `attribute` axis yields attributes of the context element. [2]

One important notion XPath has is "document order." That is, nodes can be ordered according to when they occur in the document. All of the axes are defined to return items with an ordering related to the document order. Forward Axes (such as `child`, `descendant`, and `following`) return items in document order, while Reverse Axes (such as `ancestor` and `preceding`) return items in the opposite order they occur in the document.

2.3.2 Tests

There are two types of tests in XPath step expressions: a "kind" test and a "name" test. The "kind" test can filter items from an axis to a particular type of item such as an element, a text node, or a comment.

The syntax of “kind” tests looks like this: `Student/comment()`, an expression which selects comment nodes whose parent is the `Student` item. The “name” test can filter items from an axis to those with a particular name. If any name is allowable, using an asterisk will return everything from an axis. We have already seen the syntax for these name tests.

2.3.3 Shorthand

A few shorthand notations are introduced in XPath to simplify commonly used queries. We have already seen that the child axis is default, and so it need not be qualified. Another shorthand is that `@` is short for `attribute::`, and so a step like `@size` would select the “size” attribute of an element. The syntax `..` is short for `parent::node()`, which selects the parent of the context item. Finally, the syntax `//` is replaced by `/descendant-or-self::node()/`.

Finally, a path may be rooted or relative. A rooted path starts with a leading slash, and indicates that the context item should be set to the root element of the current document before evaluating the path. Otherwise, a relative path uses the existing context item when evaluating the first step.

2.3.4 Predicates

Any expression returning a sequence, including full path expressions, or steps within a path, may be filtered by one or more predicates. Each predicate is an expression following a sequence, enclosed in square brackets. The predicate is evaluated once for each item in the sequence. Each time, the context item is set to an item from the sequence. If the result of the predicate is a singleton numeric, then it is compared against the “context position”. If it is equal, the item is yielded by the predicate, otherwise, it is skipped. If the result of the predicate is not a singleton numeric, it is coerced to a Boolean. If the Boolean is true, the item is yielded by the predicate, otherwise, it is skipped.

2.3.5 Arithmetic

XPath includes several numeric types, and as a result it supports most common arithmetic operations, including addition, subtraction, multiplication, division, integer division, and the remainder operation. It also supports comparison operators and boolean operators for complex conditions.

2.3.6 Comparisons

Since the result of every XPath expression is a sequence, comparison operations have some interesting semantics. There are two sets of comparison operators in XPath. The first set are “Value Comparison” operators: `eq, ne, le, lt, ge, gt`. They accept only singletons as their operands, and so they compare exactly one value against exactly one other value. They will raise an error if used with anything other than singletons.

The second set of comparison operators is the “General Comparison” operators: `=, !=, <=, <, >=, >`. These take any sequence of operands and return true if there exists any pair of items from the sequences that satisfy the comparison conditions. Thus, they may be thought of as “existential comparisons.”

2.3.7 Functions

Finally, XPath includes a library of built-in functions with utility operations that can be applied to strings, numbers, booleans, and nodes. There are several relevant functions, which will be discussed in the following sections.

3 DPath Implementation

With this understanding of XPath, we will now describe the implementation of DPath, our tool and language for querying files.

3.1 Deviations From XPath

While the purpose of this project was to “use XPath to query file systems,” an implementation of XPath which is 100% conforming to the specification was not a stated goal. Fully implementing a W3C standard such as XPath represents an undertaking much too large for a single semester project. In addition, an implementation which completely adheres to the XPath standard would likely be too inflexible to be useful for file system querying. So, we focused on striking a balance between the competing goals of conformance, utility, and ease of implementation. As a result, we made some changes which compromised conformance for ease of implementation, especially when they did not impact the utility of DPath. Similarly, we compromised conformance for the benefit of utility even at the cost of more implementation efforts. However, we strove to preserve as much of the language as possible, so that using DPath would feel the same as using XPath. Following is a list of deviations DPath makes from XPath.

- We discarded QNames in favor of simple identifiers (with the same syntactic definition as the local part of a QName). QNames have no real place in a file system, because no XML is actually involved.
- We dramatically simplified the number of types. XPath includes several numeric types in various sizes, as well as date and time types, and any user-defined type from XML Schema. DPath simply includes a 64-bit signed integer type, a double precision floating point type, strings, booleans, and files.
- We implemented only a subset of the built-in functions (listed later in this section).
- We did not implement the `self`, `preceding`, `preceding-sibling`, `following`, and `following-sibling` axes. These do not have obvious places in directory querying.
- Since file systems have no equivalent to document ordering, we discarded the distinction between forward and reverse axes, and do not provide any guarantee on what order the axes yield items. As a further result of this, without deterministic item ordering, there is no reason to use predicates for numeric indexing. Therefore, all predicate expressions are cast to booleans rather than the semantics described in Section 2.3.4.
- Since not all file names conform to the limitations of the QName local identifier syntax, we introduced a shorthand syntax, `#"literal"` which can be used wherever a name test is expected. For example, `#".git"` can be used to refer to the `.git` directory, since `.git` is not a valid QName local identifier.
- The only kind tests are `file()` and `dir()`.
- Some lesser-known parts of XPath were not implemented. We judged that the utility they provided to DPath was not worth their implementation effort. These parts include: for-return expressions, if-conditionals, quantifiers, and type casting expressions, and the union and intersection operators.

Despite this rather daunting list of omissions and deviations, the resulting DPath language retains the necessary flavor and power of XPath for querying directories, and was sufficiently simple to be implemented by a single person in a single semester.

3.2 Parsing

DPath is implemented in the Go programming language. This is a relatively new language which was released open-source by Google in November of 2009 [7].

The scanner is implemented using Nex [8], a Go tool inspired and heavily based on the earlier Unix tool Lex [9]. This tool takes as input a source file (`dpath.nex`) containing several regular expressions, each one defining a token. It produces as output a lexical analyzer in Go (`dpath.nn.go`) which is capable of parsing a stream of these tokens and returning them sequentially.

The parser is implemented using Go's built-in implementation of the Yacc tool [10]. This tool takes as input a source file (`dpath.y`) containing a grammar, where each production specifies an action to be taken, in the form of Go code. It produces as output an LR(1) parser in Go (`y.go`).

The generated lexer and parser are accessed from the rest of the code via the `Parse()` function, which takes an input stream and returns either a parse tree or an error.

The returned parse tree is an "interface" which supports printing (for diagnostics) and evaluation. The `ParseTree` interface has an implementation for many different productions in the grammar. For instance, there is an implementation of this interface called `BinopTree`. This implementation represents an element that is a binary operator. It contains the operator itself as a string, as well as the left and right hand sides of the operator as pointers to other `ParseTrees`.

Code evaluation is therefore distributed across the `Evaluate()` methods from each implementation of `ParseTree`, which may all be found in `tree.go`.

3.3 Data Model

Internally, the DPath data model mirrors XPath's specification. An `Item` interface is defined in `item.go`. `Item` has an implementation for each of DPath's types. Its implementations must support comparison as well as the basic operators.

Another important part of the data model is the `Axis` interface. This interface requires two methods: `GetByName()`, which helps implement name filters in step expressions, and `Iterate()`, which returns a `Sequence` of nodes related to the current context object through this axis. The `Iterate()` method implements most of the axis operations that don't involve name tests. `Axis` has implementations for the child, parent, descendant, ancestor, descendant or self, and ancestor or self axes. These implementations may all be found in `axis.go`.

However, the most important interface of all is the `Sequence` interface. Defined in `sequence.go`, it supports two major operations. `Value()` returns the current value "pointed at" by the sequence. `Next()` attempts to advance the sequence forward by one, returning true when there are more items, or false when none remain in the sequence. There are several implementations of the `Sequence` interface, ranging from simple lists to complex operations. Most path queries are evaluated by composing sequences, so it is very important to understand them. Therefore, a list of sequences implementations in DPath is presented below.

- `WrapperSequence` is the simplest sequence type. It contains a list of items (computed ahead of time), and returns them one by one as `Next()` is called. This sequence can be used with empty lists and singleton lists, and so it is used by much of the arithmetic evaluation code.
- `RangeSequence` is a numeric sequence, which takes an initial and final value. Each time `Next()` is called, the current value is incremented. When the current value reaches the final one, the sequence stops yielding items. This is used to implement XPath numeric ranges (section 3.3.1 of [2])
- `ExpressionFilter` is a complex filter which takes as input a "source" sequence and a list of parsed predicate expressions. The `ExpressionFilter` *wraps* its source sequence. Each time `Next()` is called, it calls `Next()` on the underlying sequence to get an item. It sets this as the context item and evaluates each expression. If any of the predicates fail, it repeats this process until it finds an item from its source sequence that satisfies all predicates. Therefore, `ExpressionFilter` filters a source sequence by applying predicates.
- `ConditionFilter` is similar to the `ExpressionFilter`, but instead of filtering by XPath expressions, it filters by an arbitrary Go function. This is useful for some specialized operations.
- `PathSequence` is possibly the most important part of DPath. It implements the heart of the language: a single step from a path expression. To do this, it takes as input a sequence (the output from the previous step) and an expression (the current step). For each item in the sequence, it evaluates the expression and yields items from the resulting sequence.
- `DescendantSequence` implements the task of iterating over the descendant axis. It does this by performing a depth first search of the filesystem starting at the current directory. When `Next()` is called, it pops a directory from the stack, performs the `readdir()` system call, and yields files and directories from the resulting list. As it yields, it pushes children onto the stack, so that once the current directory is yielded, a new one is available to explore on the stack.

3.4 Example Queries

Following are a few sample queries that illustrate how query evaluation works in DPath.

3.4.1 A Simple Example

First, let us consider the following query:

```
(1 to 10)[. mod 2 eq 0]
```

This query involves no files or axes. Instead, it involves a little-known feature of XPath called "range expressions," which are numeric expressions that produce lists of numerics. `(1 to 10)` produces a list of the integers 1 through 10, while the predicate afterward filters it to those which are divisible by 2. Thus, the output of this expression should be the sequence `2, 4, 6, 8, 10`.

First, the parser produces the parse tree shown in Figure 1.

Next, `Evaluate()` is called on the `FilteredSequenceTree`. This function first evaluates `BinopTree` on the left hand side, which returns a `RangeSequence`. The `RangeSequence` dynamically creates a list

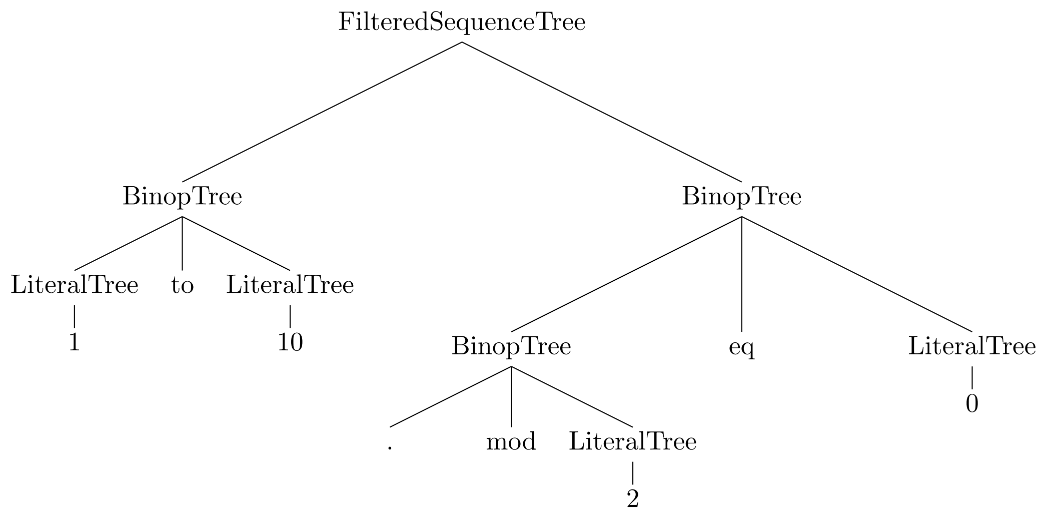

Figure 1: Parse tree for `(1 to 10)[. mod 2 eq 0]`

of numbers from 1 to 10, computing the next one each time `Next()` is called. This `RangeSequence` is returned back to the `FilteredSequenceTree`. From there, an `ExpressionFilter` is created. This is another sequence type, which takes a source sequence (our range) and a list of `ParseTrees` which are predicates. For each item in the source sequence, it sets the context object to that item, evaluates each predicate, and returns the item if all predicates are true.

So, an `ExpressionFilter` is returned from the `Evaluate()` function. At this point, the main function begins calling `Next()` on the returned sequence and printing out the results. This causes the `ExpressionFilter` to call `Next()` on the `RangeSequence`, and then filter the results until it finds an item satisfying all predicates, which it returns. The first such item is the integer 2, then 4, 6, 8, and 10, at which point the `RangeSequence` runs out.

One important thing to note about this entire process is that not very much computation occurs when `Evaluate()` is called. Instead, sequences are constructed that dynamically produce and filter items each time `Next()` is called. This is important, because implementations which load an entire sequence of items into memory would fail miserably on large queries, like those involving `descendant-or-self` [1].

3.4.2 A Filesystem Query

Next, let us consider the following query.

```
.//*
```

This query returns all files that are descendants of the current directory. Figure 2 shows the effective parse tree.

In order to evaluate this query, an entire system of sequences is created and combined. The tree in Figure 3 represents the structure of the system of sequences.

We see that the first sequence created (at the lower left) is a `WrapperSequence`, which wraps a list of exactly one item, the context item. In this case, the context item is the current directory. This item is passed into the parent `PathSequence`, which evaluates the implicit `descendant-or-self::*` expression on that context item. This calls `Iterate()` on the `DescendantOrSelfAxis`. Since the `descendant-or-self` axis is a combination of the current context item and its descendants, it simply concatenates a sequence containing the current context item with the `DescendantSequence`.

Finally, this sequence is passed to the outer `PathSequence`, which evaluates the `*` step for each item returned by it. This involves calling `Iterate()` on the default axis, `ChildAxis`. This simply reads the contents of the directory and returns a `WrapperSequence` of children of the context item (or an empty sequence when the context item is a file).

[1] There is one case where an entire Sequence must be loaded into memory: general comparison operators. Every pair of items must be tested, so the left hand side is loaded into memory before iterating one by one over the right hand side. This is illustrated by the performance difference for the expressions `(1 to 1000000) = 1` versus `1 = (1 to 1000000)`. This would also be necessary for the union and intersection operators, should they be implemented.

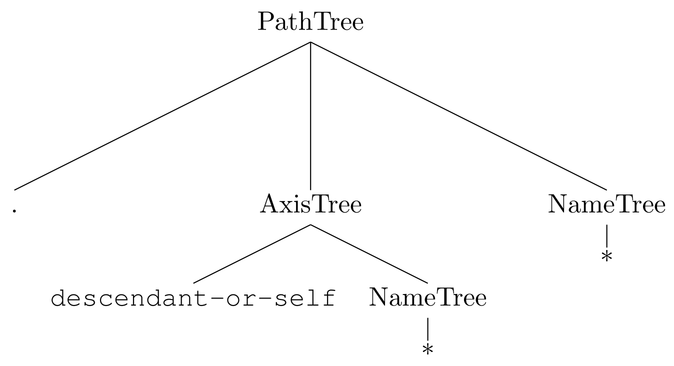

Figure 2: Effective parse tree of expression `.//*`

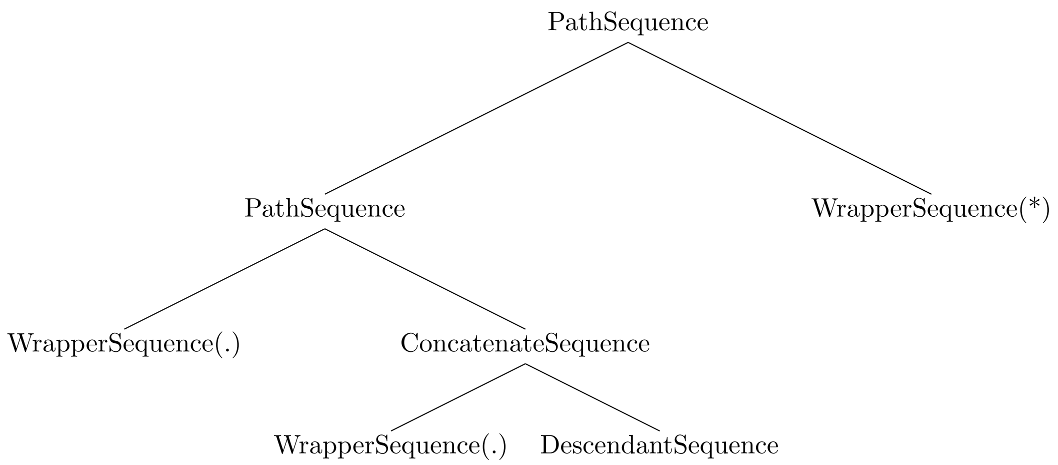

Figure 3: Sequence diagram of expression `.//*`

It's interesting to note that the output of the first two steps is all descendants of the current directory, *including the current directory.* But when this sequence is passed into the last step, all the files are filtered (since they have no children), and each directory is iterated over again! The only observable effect of this is that the current directory is not included in the final output, whereas it would have been included if the query were simply the first two steps (i.e. `./descendant-or-self::*`). In fact, it makes sense that a path containing the step `descendant-or-self::*` followed by `child::*` could be simplified to `descendant::*`, and a simple optimizer could certainly take advantage of this fact. However, DPath includes no query optimization, always using the naïve approach that exactly mirrors the semantics of a query.

3.5 Built-in Functions and Attributes

DPath has XPath's powerful path syntax for referencing items in a tree structure using their parent, child, ancestor, and descendant relationships. However, this would not be useful without the ability to apply more conditions on these relationships. As such, DPath implements a substantial subset of XPath's built-in functions, especially focusing on string operations. This section provides a brief listing of these functions. More documentation is available in the `SYNTAX.md` file included in the DPath source directory.

- `string()`, `concat()`, and `substring()` for converting and concatenating, and taking substrings of strings.
- `string-length()` for counting characters in strings.
- `starts-with()`, `ends-with()`, and `contains()` for checking conditions on strings.
- `matches()` for matching strings against regular expressions.

- `count()` for finding the length of sequences, along with `empty()` and `exists()` for checking special cases of these lengths.
- `not()` for inverting booleans and `boolean()` for converting anything to a boolean.
- `path()` for returning a file's full path, and `name()` for returning its base name.
- `round()` for rounding a number

In addition, the XPath attribute axis has an analog in DPath. The DPath attribute axis contains only one item, the size of the context item. For instance, `y.go/@size` returns the integer 21474, the size of the file in bytes.

4 Related Work

4.1 XPath Query Evaluation

The current state of XPath query evaluation resembles that of regular expression evaluation. Regular expressions have a well-established theory and require only time linear in the length of their input in order to decide whether an input string is matched. However, most popular standard implementations of regular expressions (such as those present in Java, Python, Perl, PHP, Ruby, and others, but not Go, and therefore not DPath) use greedy "backtracking" algorithms that have a worst-case exponential runtime [11]. These backtracking algorithms are based on a semantic understanding of regular expressions, rather than the theory of finite automata.

XPath queries do not have quite the same body of well-established theory, but Gottlob *et al.* have demonstrated that the XPath query evaluation problem can be solved in low-degree polynomial time [12]. This polynomial is in both the size of the input document and the query. Additionally, they have given several subsets of XPath which have lower complexity, and algorithms which can evaluate queries from these subsets efficiently [13, 12].

Despite these findings, Gottlob *et al.* demonstrated that many mainstream XPath implementations have runtime that is exponential in query size, even two years after their initial findings [12]. They conjecture that this is because most implementations are motivated by the semantics described in the XPath specification, which leads implementors to an exponential-time algorithm. This is the same approach we have taken in DPath, and thus our implementation also has a time complexity that is exponential in the worst case. This can easily be demonstrated with the following sequence of DPath queries, each of which runs progressively slower (adapted from the experiments in [12]):

```
> dpath '*'
> dpath '*[count(../*) > 1]'
> dpath '*[count(../*[count(../*) > 1]) > 1]
> dpath '*[count(../*[count(../*[count(../*) > 1]) > 1]) > 1]
```

Unfortunately, efficient algorithms for XPath (such as those proposed by Gottlob *et al.*) generally expect an indexed list of nodes available [12]. In the filesystem, nodes are discovered by the `readdir()` system call, which costs a system call and potentially a disk seek. As a result, the more efficient algorithms are not easily used without a system for creating and maintaining filesystem indices. Additionally, for queries which involve searching the whole file system for a descendant (e.g. `//document.tex`), DPath will inherently be slow because it cannot use an index to limit the search, and will instead traverse the whole filesystem.

4.2 Filesystem Querying

A discussion of file system querying would not be complete without mentioning the GNU Findutils [14]. This is a collection of file search programs (`find`, `locate`, `xargs`, and `updatedb`) which are distributed as standard on many Linux and other Unix-based computers. The `find` program serves a similar purpose to DPath, but instead of evaluating XPath queries, it uses a system of command-line flags to specify search criteria.

GNU `find` is capable of searching based on conditions on the file path and size (both of which DPath supports), as well as modes, owner, permissions, and access times (which DPath does not support). It allows the user to "prune" the search tree by specifying a max recursion depth, or specifying folders not to descend into. DPath is capable of these restrictions, but they are not evaluated efficiently, since they must be expressed as predicates on the sequence of items which is already being traversed.

One critical feature that GNU Findutils has is the capability to create a filesystem index and query that, rather than searching the file system each time a query runs. This functionality is mostly used for fast name searches, since file names change much less frequently than other metadata. This feature is important because, during file system searches, the limiting factor is frequently not the speed of the query evaluator, but rather the number of times the disk is accessed for file system calls such as `readdir()` and `stat()`. File system indices can reduce or eliminate these system calls, but require regular updates (similar to database indices). Unfortunately, the filesystem index must be manually updated by the user [14]. Unless the user is knowledgeable enough about their system to know that (a) indices would be useful, and (b) how to schedule a job for their maintenance, they cannot take advantage of the indexing performance boost.

4.3 XPath Over Filesystem

DPath is not a novel invention. We were able to find at least two other implementations of XPath on the file system on the Internet.

The first implementation we found is a modification of the GNU `find` tool [15]. However, it required some non-trivial modifications to the build process, as well as manual steps, in order to compile correctly. This implementation exposed more file attributes to the system and included a maximum depth option, and it relied on a pre-made XPath library rather than implementing its own.

The second implementation, called FOXPath, is an extension to XPath rather than a separate language unrelated to XML [16]. As such, FOXPath allows users to query filesystems containing XML files in a way that uses their contents. It is based on XPath 3.0, while DPath is based on XPath 2.0. One major limitation to FOXPath is that it introduces a rather complex, context-dependent syntax for escaping file names which are not syntactically valid QNames. It is also based on an existing XML library, unlike DPath.

5 Future Work

Potentially the greatest potential improvement of DPath would be filesystem indexing. In particular, DPath could make use of the indices created by the `find` utility, and use these to speed up the evaluation of (at least) rooted descendant queries (such as `//file.txt`).

Additionally, it would be beneficial to include more filesystem metadata in the attribute axis of files. However, Go's built-in operating system interface did not provide such details, in an attempt at platform independence. (In fact, DPath is at least compatible with Mac OS and Linux. In theory, Go works on Windows as well, but we have not tested DPath on that system).

Another avenue for improvement is to add query optimization. As mentioned earlier, queries such as `//*` are evaluated inefficiently with our query evaluator. Instead, this query could be translated to an equivalent query such as `/descendant::node()`, which is more efficiently evaluated by DPath. Another opportunity for optimization would be caching filesystem reads and writes so as to avoid unnecessary system calls.

Finally, there are a few portions of the XPath language we did not implement, such as for-return expressions, if-conditionals, quantifiers, type casting expressions, union, and intersection operators. Although these did not seem particularly significant for the utility of a directory query language, implementing them would make DPath more "feature complete" as an XPath-like language.

6 Conclusion

In this project, we have described the implementation of DPath, an implementation of XPath for file system querying. We have found the development to be informative and the end result to be a useful tool. The complete source code to DPath, along with setup instructions and other documentation, may be found at: `https://github.com/brenns10/dpath`

References

[1] D. Barron, "Why use sgml?," *Electronic Publishing*, vol. 2, no. 1, pp. 3–24, 1989.

[2] J. Simeon, A. Berglund, M. Kay, M. Fernandez, S. Boag, J. Robie, and D. Chamberlin, "XML path language (XPath) 2.0 (second edition)," W3C recommendation, W3C, Dec. 2010. `http://www.w3.org/TR/2010/REC-xpath20-20101214/`.

[3] J. Simeon, M. Dyck, A. Malhotra, K. Rose, M. Fernandez, D. Draper, P. Fankhauser, M. Rys, and P. Wadler, "XQuery 1.0 and XPath 2.0 formal semantics (second edition)," W3C recommendation, W3C, Dec. 2010. `http://www.w3.org/TR/2010/REC-xquery-semantics-20101214/`.

[4] M. Fernandez, A. Malhotra, A. Berglund, J. Marsh, M. Nagy, and N. Walsh, "XQuery 1.0 and XPath 2.0 data model (XDM) (second edition)," W3C recommendation, W3C, Dec. 2010. `http://www.w3.org/TR/2010/REC-xpath-datamodel-20101214/`.

[5] N. Walsh, J. Melton, A. Malhotra, and M. Kay, "XQuery 1.0 and XPath 2.0 functions and operators (second edition)," W3C recommendation, W3C, Dec. 2010. `http://www.w3.org/TR/2010/REC-xpath-functions-20101214/`.

[6] T. Bray, D. Hollander, A. Layman, R. Tobin, and H. Thompson, "Namespaces in xml 1.0 (third edition)," W3C recommendation, W3C, Dec. 2009. `https://www.w3.org/TR/2009/REC-xml-names-20091208/`.

[7] R. Griesemer, R. Pike, K. Thompson, I. Taylor, R. Cox, J. Kim, and A. Langley, "Hey! ho! let's go!," *Google Open Source Blog*, nov 2009. `https://opensource.googleblog.com/2009/11/hey-ho-lets-go.html`.

[8] B. Lynn, "Nex," 2011. `http://crypto.stanford.edu/~blynn/nex/`.

[9] M. E. Lesk and E. Schmidt, "Lex: A lexical analyzer generator," 1975.

[10] S. C. Johnson, *Yacc: Yet another compiler-compiler*, vol. 32. Bell Laboratories Murray Hill, NJ, 1975.

[11] R. Cox, "Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby,...)," *URL: `http://swtch.com/rsc/regexp/regexp1.html`*, 2007.

[12] G. Gottlob, C. Koch, and R. Pichler, "Efficient algorithms for processing xpath queries," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 2, pp. 444–491, 2005.

[13] G. Gottlob, C. Koch, and R. Pichler, "The complexity of xpath query evaluation," in *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 179–190, ACM, 2003.

[14] J. Youngman, "Finding files," 2011. `https://www.gnu.org/software/findutils/manual/html_mono/find.html`.

[15] O. Paraschenko, "Find with xpath over file system," 2005. `http://uucode.com/texts/xfind/`.

[16] H.-J. Rennau, "FOXpath - an expression language for selecting files and folders," in *Proceedings of Balisage: The Markup Conference 2016*, Mulberry Technologies, Inc., 2016.