

**Санкт-Петербургский Государственный Университет
Математико-механический факультет**

Кафедра системного программирования

Реализация XPath над S-выражениями

Миленин Евгений Геннадьевич

Содержание

1.	Введение	3
2.	Введение в предметную область.....	6
2.1.	S-выражения.....	6
2.2.	XML Information Set	8
2.3.	XPath	9
2.4.	Виртуализация XML	11
3.	Обзор родственных работ	12
3.1.	Способы навигации по S-выражениям. Pattern Matching.....	12
3.2.	SXML	14
3.3.	SXPath.....	16
3.4.	S-выражения, SXPath и Parent Pointers.....	17
4.	Описание предлагаемого подхода	19
4.1.	Отображение S-выражений на XPath Data Model	19
4.2.	Generative XPath.....	22
4.3.	Технические аспекты реализации XPath над S-выражениями.....	24
4.3.1.	Boxes	25
4.3.2.	Организация сравнения узлов	26
4.3.3.	Реализация функций осей XPath	29
4.4.	Применение библиотеки	30
4.5.	Ограничения библиотеки.....	31
4.6.	Примеры использования	32
4.7.	Оценка производительности	35
5.	Заключение.....	38
6.	Список литературы.....	39
7.	Приложение.....	41

1. Введение

Главной синтаксической конструкцией языков семейства Lisp являются списки, так называемые S-выражения. Они являются основой для представления данных и самого программного кода Lisp языков. При помощи списков можно моделировать любые типы данных, в том числе деревья и даже графы. Очень часто списки удобно рассматривать именно как деревья, как форма представления иерархических данных. Во всех языках семейства Lisp существуют примитивы для работы с S-выражениями – взять голову, получить хвост списка, а также базовые операции для работы со списками, присутствующие практически во всех реализациях, такие как `map`, `filter` и др.; этих операций зачастую достаточно для того, чтобы реализовать функциональность, связанную с навигацией по дереву. Списки, моделирующие иерархические структуры, могут встречаться в текстовых процессорах, компиляторах, оптимизаторах и других семейства программных продуктов. Очень частой задачей является обход или поиск в списке как в иерархической структуре. Сама по себе задача работы с деревьями является довольно простой, но, несмотря на это, код реализации может содержать ошибки, требовать поддержку и сопровождение, внесение изменений при изменении условия задачи. В дальнейшем такой подход может привести либо к дублированию кода, порождающему множество новых похожих функций, либо к написанию более обобщённых примитивов навигации, и в конечном итоге к появлению библиотеки, которая при близком рассмотрении оказывается собственной реализацией некоего подмножества языка XPath. Язык XML Path (XPath) – это язык для адресации частей XML-документа. Нет причин ограничивать использование XPath только XML данными. Более того, сами языки XPath и XQuery определяются в терминах абстрактной модели данных, а не традиционного текстового представления XML документа.

На данный момент можно заметить общую тенденцию к «виртуализации» XML, говоря в терминах Virtual XML Garden от IBM – существует целый ряд как научных исследований, так и промышленных стандартов (XLink от Microsoft), которые ставят своей целью предоставить возможность использовать XPath для других форматов данных, отображающихся на XML. Одним из таких направлений является Generative XPath, реализованный на языке Scheme, одном из самых распространённых диалектов Lisp.

Существует целый ряд исследований, использующих тот факт, что XML семантически близок s-выражениям языка Lisp. Среди них можно упомянуть формат SXML, который использует s-выражения языка Scheme для представления информационных единиц XML Infoset. Этот подход

основывается на том факте, что произвольный XML документ может быть отображён на некоторое подмножество s-выражений.

Однако мы хотим подойти к этому вопросу с другой стороны – предложить отображение произвольного s-выражения в XML представление, обладающее некоторыми удобными свойствами, которые могли бы позволить использовать язык XPath для навигации по древесной структуре s-выражения.

Существует множество дискуссий, посвящённых сравнительному анализу s-выражений и XML. Lisp и Scheme программисты имели дело с иерархическими данными задолго до появления XML, и поэтому удивительным является то, что до сих пор не существует инструментов, аналогичных каким-либо XML стандартам.

Таким образом, данная работа ставит своей целью показать, как можно работать с s-выражениями при помощи стандарта XPath, предложить отображение произвольного s-выражения на XPath Data Model, которое было бы удобно использовать для построения Xpath запросов над s-выражениями, а также реализацию этого стандарта на языке Scheme.

Мотивацией для этого является желание избавиться от частных решений и использовать общеизвестный стандарт XPath для решения часто возникающей задачи навигации по s-выражению. Основным преимуществом такого подхода является повышение эффективности труда Scheme программиста: уменьшение количества строк кода, времени, затраченного на реализацию, тестирование, поддержку и модификацию кода при решении задач, связанных с поиском по s-выражениям. Другим преимуществом является разделение высокоуровневой логики приложения, имеющего дело с деревьями, и задачи обработки дерева, представленного списком, что является более общей задачей технического характера.

Хотелось бы отметить, что данная тема представляет не только научный интерес, но имеет также и практическую ценность.

Предисловие

Данная работы была выполнена в рамках дипломного проекта под руководством Олега Паращенко (СПбГУ, мат-мех, кафедра системного программирования, 2007 год). Она посвящена вопросу навигации над S-выражениями посредством XPath. В лучших традициях дипломных работ, вначале (часть №2) представлено обширное введение в предметную область – описание S-выражений, стандартов XML Infoset и XPath, а также вопросы виртуализации XML, так что данная работа может быть прочитана практически без какой-либо подготовки в этой сфере. В части 3 представлен обзор существующих подходов для решения задач в этой области (касательно навигации по S-выражениям), здесь показывается, почему при

решении данной задачи необходим новый подход. В части 4 даётся обзор технологии Generative XPath(автор – Олег Парашенко), которая используется при реализации библиотеки, в остальном же, начиная с четвёртой части, описывается концептуально предлагаемый автором подход, подробности реализации библиотеки, а также примеры её использования и вопросы производительности.

В заключение, хотелось бы выразить особую благодарность Олегу Парашенко за предложенную тему, за помощь и руководство в работе, а также за то, что он ввёл меня в интересный мир S-выражений и Generative XPath.

2. Введение в предметную область

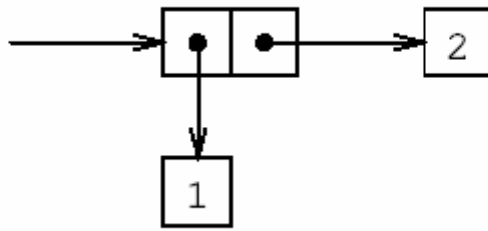
2.1. S-выражения

S-выражения, также известные как “Ess Expressions” [2], являются символьными выражениями в семействе Lisp языков (LISP, от англ. List Processing — «обработка списков»). Наиболее распространёнными представителями семейства Lisp языков являются Scheme [6] и CommonLisp [7]. Далее, говоря о реализации, мы будем в первую очередь иметь в виду язык Scheme, однако s-выражения являются общими для всех Lisp языков. Детали синтаксиса или поддерживаемых типов данных могут отличаться для разных представителей семейства Lisp языков, однако общей чертой является представление s-выражений в скобочной префиксной форме записи. Говоря неформальным языком, s-выражение – это либо атомарное значение (число, строка, символ и т.д.), либо список s-выражений.

Более строго, s-выражение – это [1, 2]:

- Атом: число, строка, литера(character), процедура(lambda-выражение), вектор, символьные данные и т.д.; также иногда выделяют отдельный атом NIL(например, в CommonLisp), или пустой список – ‘()’.
- Составная структура, называемая парой(pair), создаётся при помощи элементарной процедуры cons. Эта процедура принимает два аргумента и возвращает объект данных, который содержит эти два аргумента в качестве частей. Имея пару, мы можем получить ее части с помощью элементарных процедур car и cdr. Pair– это единственный составной тип данных; в некоторых Lisp диалектах определяется предикат atom, согласно которому атомом является всё, что не является pair.

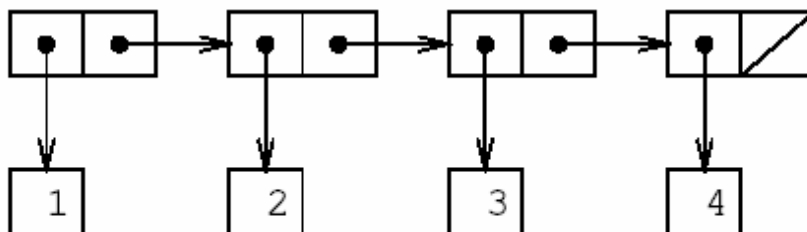
Имея в наличии примитив составных данных pair, мы можем использовать его как строительный блок для создания любых сложных структур. Говоря алгебраическим языком, множество s-выражений замкнуто относительно операции cons. Если операция cons соединяет атомарные данные, то результат называется dotted pair и обозначается как (S1 . S2) – например (1 . 2), или (AAA . #t).



Пару обычно представляют в виде стрелочной диаграммы, состоящей из ячеек и стрелок, указывающих на ячейки. Пара состоит из двух ячеек, причем левая из них содержит (указатель на – в случае, если ячейка состоит не из атомарных данных) *car* этой пары, а правая — ее *cdr*.

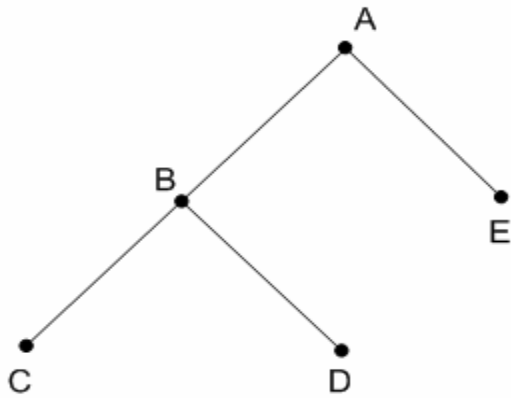
С помощью пар в языке Lisp строятся также списки, которые можно разделить на два типа:

- ProperList: либо пустой список '()', либо пара, чья *cdr* часть является также Proper-списком. Такие списки всегда оканчиваются пустым списком. S-выражение (s1 s2 ... sn), таким образом, является «синтаксическим сахаром» (сокращённой формой записи), и эквивалентно (cons s1 (cons s2 (cons ... (cons sn '())))) в "cons" нотации.
- DottedPair-список – список, заканчивающийся атомарным значением, а не пустым списком(NIL). Примером такого списка является (a b . c) в краткой форме записи, или (cons a (cons b c)).



Последовательность 1, 2, 3, 4, представленная в виде цепочки пар.

S-выражения в семействе Lisp языков представляют не только данные, но также и код, который записывается в префиксной нотации. Отталкиваясь от префиксной формы записи, применяемой для s-выражений, можно предложить отображение s-выражения на древесную структуру: в простейшем случае, s-выражение вида (s1 s2 ... sn), если оно является proper-списком, в голове которого находится символ, даёт дерево, корень которого – это *car*, а лес из детей – это хвост *cdr*. Такое отображение особенно очевидно для тех случаев, когда список является записью в полной скобочной форме (например, список (A (B (C) (D)) (E)) записан в полной скобочной форме, в отличие от (A (B C D) (E)), где элементы C и D не окружены скобками).



Чуть позже будет представлено строгое отображение произвольного s-выражения на модель данных XPath.

Надо сказать, что можно предложить достаточно много отображений s-выражений на какую-либо древесную структуру, однако сам по себе факт, заключающийся в том, что s-выражения можно отображать на деревья, не очень полезен и не нов, гораздо больший интерес представляет то, что, сделав шаг дальше, мы предлагаем способы работы с s-выражениями, являющиеся стандартными для других древообразных структур. А именно, речь идёт об XML и стандарте XPath.

2.2. XML Information Set

Информационное пространство XML Infoset [5] определяется как абстрактное множество данных, которое описывает информацию, содержащуюся в правильном (well-formed) XML-документе, состоящее из информационных элементов (information items), которые обозначают элементы, атрибуты, символьные данные, команды по обработке и другие компоненты документа. Каждый информационный элемент имеет набор ассоциированных с ним свойств (properties), например, имя и идентификатор пространства имен. XML [3] документ со своими хорошо знакомыми открывающими и закрывающими угловыми скобками является конкретной реализацией XML Infoset. Рекомендация XML Information Set не содержит никаких явных требований к структурам данных или интерфейсу доступа к ним, поэтому возможны различные интерпретации абстрактной модели XML Information Set.

2.3. XPath

XPath (XML Path) является языком для адресации частей XML-документа. В данной работе речь пойдёт о версии стандарта XPath 1.0 [4], отчасти это вызвано тем, что технология GenerativeXPath, которая использовалась для реализации XPath над s-выражениями, реализует эту версию стандарта. XPath трактует XML документ как дерево и работает с ним в терминах XPath Модели Данных, согласно которой дерево может состоять из узлов, имеющих семь типов:

- корневой узел – корень дерева, в XML Infoset он соответствует информационному элементу document.
- узлы элементов – получаются из информационного элемента element. Имеют локальное и расширенное имя, в качестве непосредственных потомков могут иметь узлы элементов, текста, инструкций обработки или узлы комментариев
- текстовые узлы
- узлы инструкций обработки
- узлы комментариев
- также в стандарте рассматриваются узлы атрибутов и узлы пространства имен, однако в нашем отображении s-выражений на XPath модель данных они не будут принимать участия, поэтому мы будем опускать их из рассмотрения

На множестве узлов определяется порядок, называемый порядком появления в документе. Он соответствует появлению узлов в соответствующем XML представлении документа. Следовательно, корневой узел всегда будет первым узлом, а узлы элементов будут предшествовать своим непосредственным потомкам. Таким образом, порядок появления в документе упорядочивает узлы элементов согласно очередности появления в XML документе соответствующих открывающих тэгов.

Основной синтаксической конструкцией языка является выражение, которое соответствует некоторому правилу грамматики. Вычисление выражения осуществляется относительно текущего узла (контекстного узла). Наиболее важным типом выражений является путь адресации (location path), состоящий из шагов доступа (location steps), разделенных символом '/'. Шаг доступа состоит из трёх частей:

- Ось (axis), определяющую соотношение в дереве между узлами, в контексте которых вычисляется шаг доступа, и узлами, которые выбирает шаг доступа.
- Теста узла (node test), определяющего тип и имя узла. Для текстовых узлов правило проверки text() будет возвращать true для любого текстового узла, а правило проверки узлов processing-instruction() – для любой инструкции обработки. Правило node() будет выдавать true для

любого узла, к какому бы типу он не относился, а правило проверки типа * будет возвращать true для узлов, чей тип соответствует основному типу применяемой оси – для всех осей, кроме attribute и namespaces, основным типом является элемент.

- Ноль или более предикатов, использующих произвольные выражения для дополнительной фильтрации выбранных на данном шаге узлов.

Пути адресации бывают двух типов: относительные и абсолютные: в случае относительного путь адресации отсчитывается от текущего контекстного узла, абсолютный путь начинается с символа '/', который находит корневой узел.

Рассмотрим пример XPath выражения:

```
/child::AAA/child::BBB[count(child::* ) > 2]
```

Это выражение состоит из двух шагов доступа. Первый шаг использует ось child, выбирающую всех детей корневого узла, имеющих имя AAA – корневой элемент с именем AAA. Второй шаг выбирает для всех детей узла AAA тех из них, которые удовлетворяют тесту узла (имеют имя BBB), и удовлетворяют предикату, согласно которому количество детей типа «элемент» у выбранных узлов должно быть больше двух.

Результатом вычисления каждого шага доступа и, следовательно, пути адресации, является множество узлов (node-set). Шаги в пути адресации вычисляются по очереди слева направо. Самый левый шаг вычисляется первым, обычно по отношению к узлу, который представляет корень XML-документа. Каждый последующий шаг доступа выбирает набор узлов, который вычисляется по отношению к набору узлов, выбранному предыдущим шагом доступа.

Существует целый ряд аббревиатур для записи шагов и пути адресации, так, приведённый пример можно записать в более короткой форме как /AAA/BBB[count(*) > 2]. Ось child используется по умолчанию, '/' является сокращением для /descendant-or-self::node()/, '..' является сокращением для parent::node(), а '.' – сокращение для текущего узла контекста(self::node()). Способ сокращённой записи интуитивно ясен и знаком тем, кто имеет опыт работы с файловой системой ОС Unix.

Всего различают 13 типов осей. Смысл осей достаточно очевиден из их названия: child, descendant, parent, ancestor, following-sibling, preceding-sibling, following, preceding, self, descendant-or-self, ancestor-or-self. Оси делятся на прямые и обратные, в зависимости от того, возвращают они узлы, предшествующие текущему (обратные), или нет. Таким образом, к обратным осям относятся ancestor, ancestor-or-self, preceding, preceding-sibling. Для некоторых типов предикатов решающим оказывается, является ось, к которой он применяется, прямой или обратной. Так, выражение child::para[position()=last()] (или para[last()]) находит последний потомок para

в порядке следования узлов в документе. Для выражения `ancestor::*[1]` предикат применяется к обратной оси и поэтому вернёт предка текущего узла, т.к. предикат выбирает узлы в обратном порядке. Если предикат применяется к множеству узлов, то узлы выбираются в порядке следования в документе: такое XPath выражение (`ancestor::*`)[1] вернёт корневой узел.

2.4. Виртуализация XML

Проблема обработки и навигации по иерархическим древовидным структурам данных – довольно часто встречающаяся задача в современном мире разработки программного обеспечения. Одним из частных случаев иерархических данных является XML, для адресации элементов которого служит XPath. Существует ряд проектов по виртуализации XML, позволяющих применять стандарт XPath к другим иерархическим моделям данным. Среди них можно назвать следующие проекты:

- Virtual XML Garden(Java) от IBM [19, 20]
- XLink от Microsoft(.NET) [21]
- Generative XPath (Scheme) [22]
- и др.

Рассмотрим на примере проекта Virtual XML Garden основные идеи и понятия виртуализации XML. Часто многие приложения имеют дело с так называемыми устаревшими «legacy» данными, которые при этом имеют иерархическую структуру. Также обычным явлением становится частое преобразование данных в XML и обратно, причём иногда это преобразование производится исключительно для того, чтобы применить XPath к полученным XML данным, и затем вернуться обратно к специфичным для приложения родным(native) данным. Очевидно, что такая наивная модель приводит к дополнительным накладным расходам и является довольно дорогостоящей с точки зрения производительности. Для того, чтобы избежать излишние трансформации данных, авторы [20] предлагают

- на основе анализа XPath запроса определять стратегию доступа только к необходимым узлам
- использовать технику курсоров для обеспечения легковесного интерфейса между данными и XML моделью. Это достигается путём создания адаптеров, которые позволяют рассматривать произвольные структуры данных, как если бы они являлись XML данными, при этом избегая излишних трансформаций данных и связанных с этим накладных расходов. В качестве примера в [20] предлагается «обернуть» доступ к файловой системе в курсор, соответствующий текущему файлу или директории, отображающий в виртуальный XML документ, к которому можно обратиться и при помощи XPath запроса произвести поиск по файловой системе.

3. Обзор родственных работ

3.1. Способы навигации по S-выражениям. *Pattern Matching*.

Существует несколько альтернативных возможностей для навигации по s-выражениям. Наиболее очевидный из них – это написание собственного кода для каждого из случаев навигации, используя базовые операции для работы со списками – `car`, `cdr` и `cons` [1, 2]. Очевидны также и минусы этого подхода. Написание такого кода для навигации по дереву является не очень сложной задачей, однако код может содержать ошибки, требовать поддержки, внесения модификаций и тестирования, что занимает дополнительное время и ресурсы программиста. В конечном итоге, если есть постоянная необходимость в такой навигации, этот подход приводит либо к дублированию кода, либо к формированию вспомогательной библиотеки, которая заново изобретает XPath.

Одним из средств, упрощающих навигацию по s-выражениям, является *pattern matching*. Для языка Scheme реализовано множество виртуальных машин, и для некоторых из них есть свои специфические реализации *pattern matching*-а, причём зачастую в них используются разные функции и разный синтаксис записи шаблонов. Все они основаны на общей идее, имеющей корни в таких языках как Haskell, ML или Prolog. *Pattern matching* позволяет осуществлять довольно сложную передачу управления в зависимости от соответствия s-выражения некоторому шаблону. Например, для диалекта Scheme Bigloo [11] существуют две основные формы *pattern matching*-а: *match-case* и *match-lambda* [12].

match-case `key clause...`

`key` может быть любым выражением, а `clause` имеет форму (`pattern s-expression...`).

Вычисляется *match-case* выражение следующим образом: вначале вычисляется выражение `key`, и результат последовательно сравнивается с каждым из шаблонов `pattern`; как только шаблон `pattern` принимает выражение `key`, вычисляется `s-expression` и возвращается в качестве результата.

Часто встречающаяся функция `map` может быть задана следующим образом для виртуальной машины Bigloo:

```
(define (map f l)
  (match-case l
    (() '())
    ((?x . ?y) (cons (f x) (map f y)))))
```

Для записи шаблонов наиболее часто используются обозначения

- `?`- соответствует произвольному s-выражению

- ?x сопоставляется некоторому выражению и связывает его с переменной x
- ??- может быть использована внутри списка и означает произвольную последовательность
- ???- может встречаться только в хвосте списка, удобен для сопоставления с dotted-списком; запись (a ???-) эквивалентна (a . ?-). Пример: (a ??- (b ???-) ???-) является шаблоном для сопоставления со списком, имеющим в голове символ 'a', и содержащим в хвосте список, начинающийся с символа 'b'.
- и т.д.

Существует также язык pattern matching от Andrew Wright [13](для интерпретаторов Chicken, Gauche, Guile). Та же самая функция map записывается иначе:

```
(define (map f l)
  (match-case l
    (() ())
    ((x . y) (cons (f x) (map f y))) ))
```

На первый взгляд, язык pattern matching-a от Andrew Wright похож на язык, реализованный в Bigloo, однако реально он сильно от него отличается и оказывается довольно ограниченным, так что на практике его использование при реализации навигации по s-выражениям не оказывает помощи, и усилия для написания такого кода могут даже возрасти.

В приложении приведён код реализации на языке Scheme функциональности, которая описывается запросом "//a[b]", с использованием стандартных средств языка Scheme, а также с использованием Bigloo pattern matching. В сравнении с простотой применения XPath запросов, по объёму кода и необходимостью поддержки оба подхода представляются довольно неуклюжими и громоздкими. Вариант с использованием pattern matching позволяет реализовать более простое и лаконичное решение, однако он не лишён минуса, присущего всем частным решениям.

Итак, применение технологии pattern matching имеет следующие минусы и ограничения:

- Существенным минусом является непереносимость кода между различными интерпретаторами Scheme, а также отсутствие реализации pattern matching для некоторых Scheme интерпретаторов
- само по себе применение pattern matching не избавляет от необходимости писать "custom" код для навигации, а лишь облегчает эту задачу благодаря использованию шаблонов
- pattern matching решает иные задачи: он позволяет специальным образом организовать код, в стиле языка Haskell. Pattern matching удобен для применения к s-выражениям как к спискам, а не как деревьям

Описанные выше языки ориентированы на конструирование шаблонов применительно к спискам. В последнее время в мире XML технологий очень часто встречается идея использовать язык шаблонов, как для валидации, так и для поиска в XML документах. В работе [30] формально описывается *regular expression pattern matching*, его применение для XML данных, даётся обзор технологий и исследований в этой области. В [31] предлагается использовать такой подход для s-выражений языка Scheme. Авторы этой работы представляют язык регулярных выражений *trx*. Этот язык может быть использован для написания шаблонов (*tree pattern*) для s-выражений, рассматриваемых именно как дерево, а не как список. В работе приводится целый ряд примеров, показывается, как можно реализовать сложную валидацию s-выражений при помощи языка *trx*, а также утверждается, что его можно использовать и для поиска. К сожалению, реализация языка *trx* отсутствует, поэтому мы не будем подробно рассматривать его синтаксис и его возможности. Однако сам факт такого исследования показывает, что Scheme программистам не хватает инструментов для навигации по s-выражениям как по деревьям. Поэтому это исследование представляется наиболее близким конкурентом данной работы. В связи с этим может возникать вопрос, какое средство является более удобным – *tree pattern matching* или XPath, однако этот вопрос находится за рамками данной работы.

3.2. SXML

SXML [14, 15] – это реализация XML Information Set в виде вложенных списков (S-выражений). Языки SXML и XML могут рассматриваться как два синтаксически различных представления Информационного Пространства XML Infoset.

Поскольку в XML Infoset информационный элемент является суммой своих свойств, список оказывается удобной структурой данных для представления информационного элемента. Для информационного элемента, обозначающего XML-элемент, соответствующий список начинается с имени элемента, за которым, возможно, следует коллекция атрибутов. Оставшаяся часть списка, представляющего элемент, – это упорядоченная последовательность дочерних вершин – текстовых полей, инструкций обработки и, в свою очередь, других элементов. Коллекция атрибутов помечается специальным символом @, также вводятся специальные служебные символы *PI*, *TOP* и др. (для обозначения инструкций обработки и узла документа соответственно.)

Для формата SXML существует целый ряд библиотек, адаптирующих или расширяющих XML стандарты: SSAX, SXPath, SXLT [17] и др. причём

некоторые из этих реализацией полностью соответствуют W3C рекомендациям.

Рассмотрим SXML представление для следующего XML документа:

```
<article id="123" url="http://articles.org/hello.htm">
  <title>Hello</title>
  <para>Hello <object>World</object>!</para>
</article>
```

SXML:

```
(*TOP*
 (article (@ (id "123") (url "http://articles.org/hello.htm"))
 (title "Hello")
 (para "Hello " (object "World") "!")))
```

Таким образом, каждый XML документ отображается на SXML представление, являющееся при этом s-выражением специального вида; от полученного s-выражения(SXML узла) существует возможность вернуться к соответствующему XML узлу.

Свойства и особенности SXML формата подробно излагаются в оригинальной статье [14], также существует ряд дискуссий, посвящённых сравнительному анализу s-выражений, SXML и XML [8, 9]. Однако в рамках данной работы SXML подход представляет несколько иной интерес:

- SXML предназначен прежде всего для работы с XML, представленным в виде s-выражений; в рамках SXML множество XML документов отображается на некоторое подмножество s-выражений, имеющих специальную структуру и ограничение на идентификаторы Scheme, корректные с точки зрения XML.
- для целей данной работы необходимо отображение в обратную сторону: отображение произвольного s-выражения на некоторый XML документ.
- несмотря на принципиальную разницу, у этих отображений есть общие черты, а также общие проблемы. Одной из ключевых проблем SXML, происходящей от s-выражений, является отсутствие у узла указателя на родителя [25].

3.3. SXPath

SXPath [16] – это реализация XPath на языке Scheme, предоставляющая язык запросов к SXML документам. В результате разбора XPath пути доступа получается функция (которая является s-выражением), построенная на основе низкоуровневых примитивов выборки, объединения, фильтрации и т.п. Сконструированная функция применяется к SXML-документу и возвращает список SXML-узлов. Запрос может быть записан в двух различных нотациях:

- в виде списка, состоящего из шагов доступа. В качестве шага доступа может использоваться, в том числе и произвольная пользовательская функция с заданной сигнатурой. Список шагов доступа транслируется в комбинацию примитивов SXPath с помощью набора правил перезаписи. Такие запросы формируются при помощи функции `sxpath`. Пример: `((sxpath ' (// (PERIOD 2) PREVAILING *text*)) doc)`
- запрос может представлять собой выражение XPath, записанное в виде текстового синтаксиса, полностью совместимого со Спецификацией XPath Консорциума Всемирной Сети. Тот же самый запрос при помощи функции `txpath` выглядит следующим образом: `((txpath "//PERIOD[2]/PREVAILING/text()" doc)`.

Необходимо заметить, что ни `sxpath`, ни `txpath` не являются строгой реализацией стандарта XPath. Среди проблем SXPath как библиотеки-аналога XPath замечены следующие отклонения от стандарта: в результате поиска допускаются дубликаты, а также не всегда учитывается порядок следования узлов в документе.

Например, запрос `((sxpath "/a/*/../*" '(*TOP* (a (b) (c))))` возвращает `((b) (c) (b) (c))`, содержащий дубликаты. Также некоторые другие примеры описаны в [29].

Недавно среди нескольких реализаций SXPath появилась библиотека DDO SXPath(Distinct Document Order) [18], которая лишена описанных выше недостатков, и API которой совместим с традиционным SXPath.

Так же как SXML отображение принципиально отличается от отображения множества s-выражений в подмножество XML, также SXPath отличается от целей данной работы – реализации XPath над s-выражениями. SXPath предназначен для навигации по SXML как по представлению XML в виде списков, в то время как данная работа ставит своей целью применять XPath для работы со списками как с деревьями. Поэтому SXPath нельзя считать конкурентом данного подхода, тем самым умаляя новизну данной работы. Однако нельзя не ответить на вопрос, почему, имея реализацию стандарта XPath – SXPath – хотя бы на некотором подмножестве s-выражений, нельзя

попытаться применить его ко всему множеству s-выражений. А именно, довольно напрашивающимся является следующее решение:

- пусть у нас есть некоторое s-выражение
- это выражение каким-то образом отображается (виртуально) на XML документ
- применяя к виртуальному XML документу SXML отображение, получаем SXML документ. Реально на этом шаге произошло отображение некоторого s-выражения на SXML
- выполняем XPath(например, ddo:txpath), получаем набор SXML-узлов
- далее необходимо произвести обратное отображение для каждого полученного SXML-узла в некоторое место s-выражения

Несмотря на очевидность этого подхода, он имеет ряд недостатков, в каком-то смысле он слишком прямолинеен. Преобразование произвольного s-выражения в SXML формат должно привести к полному обходу дерева s-выражения и трансформации всего дерева сразу, даже если оно полностью не понадобится. Например, для коротких запросов типа "/a/b" или "child[last()]" нет необходимости обходить всё дерево, однако, если следовать описанному методу, вначале должна быть произведена полная конвертация s-выражения. Такое поведение можно сравнить с тем, как, если бы мы, имея legacy данные (в терминах Virtual XML Garden) и желая произвести поиск с использованием XPath, вначале преобразовали бы их в XML формат и после поиска сделали бы преобразование обратно. В этой аналогии в качестве legacy данных выступает произвольное s-выражение, а в качестве XML – SXML.

В качестве дополнительного аргумента заметим, что, отображая s-выражение в SXML формат, мы можем получить совершенно новое s-выражение (являющееся одновременно SXML документом), мало похожее на предыдущее, что может вызвать затруднения при возвращении обратно к исходному s-выражению после поиска.

3.4. S-выражения, XPath и Parent Pointers

Родитель для узла x – это узел, для которого x является дочерним. Это свойство, присущее информационному элементу в XML Infoset, оказывается необходимым при работе с обратными осями XPath. S-выражения представляют собой направленные деревья, а деревья не могут иметь обратных ссылок. Поэтому, имея в наличии подписание, мы не можем указать родительское s-выражение. С целью нахождения родителя теоретически всегда существует возможность поиска по всему дереву вниз от корня, чтобы найти тот узел, одним из дочерних узлов которого является данный. Очевидно, что данный метод обладает существенным недостатком,

связанным с его временной дороговизной, поскольку в общем случае требуется просканировать всё дерево s-выражения.

С этой проблемой столкнулись также создатели XPath. В статье [25] в рамках её решения предлагается, помимо поиска родителя, ещё 4 решения. Три из них подразумевают добавление в структуру SXML узла специальной аннотации, содержащей информацию о родителе. Это может быть:

- прямая ссылка. S-выражение становится графом с циклами, что создаёт сложности с обработкой, например печатью, циклических структур
- неявные ссылки – псевдоуказатели, идентификаторы, являющиеся ключами в хэш-таблице, значениями которой являются ссылки на родителя s-выражения
- хранение указателей на родителя, обёрнутых в процедуры, позволяет избавиться от циклических ссылок, но снижает производительность

Все эти варианты имеют своим недостатком то, что в SXML узел добавляется служебная информация.

Первоначально XPath использовал самый первый, неэффективный метод поиска родителя по всему дереву. На данный момент применяется более эффективный способ, основанный на идее предварительного анализа XPath выражения: в процессе вычисления XPath запроса происходит сохранение родительских узлов контекста в том случае, если они могут потребоваться при дальнейшем вычислении выражения [26]. Такой метод имеет ряд достоинств, а именно, запоминаются только те узлы, которые понадобятся в дальнейшем, и наоборот, если XPath запрос не содержит обратных осей, то предки узлов не запоминаются вообще. Также этот алгоритм более удобен с точки зрения сборки мусора. Процесс автоматического управления памятью, который производит поиск и освобождение областей памяти, занимаемых теми объектами программы, на которые программа никогда больше не будет ссылаться в будущем, используется во всех языках семейства Lisp, в том числе и в Scheme. В случае если сборка мусора будет запущена в процессе выполнения XPath запроса, те части SXML документа, которые больше не потребуются для дальнейшего вычисления выражения, смогут быть возвращены сборщиком мусора.

4. Описание предлагаемого подхода

4.1. Отображение S-выражений на XPath Data Model

Выше было продемонстрировано, как можно сопоставить «хорошему» s-выражению, записанному в полной скобочной форме и имеющему непустую головную часть, древовидную структуру.

Теперь будет построено отображение произвольного s-выражения на XPath модель данных [4]. Поскольку узлы модели данных XPath могут быть получены из информационных элементов в XML Information Set [5], то также можно говорить и об отображении произвольного s-выражения на XML [3].

Прежде всего, введём некоторые уточнения и ограничения на структуру s-выражений. Несложно показать, что s-выражение может иметь внутри себя циклы – такое s-выражение содержит ссылку на само себя и имеет вид `#sexp=("bbb" # sexp #)`. С их помощью можно моделировать графы. Т.к. мы собираемся реализовать XPath над s-выражениями, а применительно к графам нет возможности говорить об XPath адресации, то циклические s-выражения будут исключены из рассмотрения. Также, из записи пар(pair) при помощи операции cons, следует то, что одно и то же s-выражение может быть записано многими способами. Например, `((a) . (b))` эквивалентно `((a) b)`. Для каждого s-выражения можно рассматривать его полную запись и сокращённую запись, например `(a b c)` и `(cons a (cons b (cons c '())))`. При построении отображения удобно считать, что отображаемое s-выражение записано в сокращённой форме, т.е. к нему полностью применён «синтаксический сахар».

Рассмотрим пример:

```
(define (my-hello who . rest)
  (display "Hello, _")
  (display (my-string (my-string who)))
  (display "!\n"))
```

Это простое s-выражение также является синтаксически корректным Scheme кодом. Оно определяет функцию "my-hello", принимающую один или более параметров, при этом используется только первый из них(who), а остальные(rest) игнорируются. При печати приветствия вызывается функция my-string, которая введена с целью дальнейшей иллюстрации применения XPath.

Прежде чем будет представлено отображение, сформулируем некоторые требования, которым оно должно удовлетворять:

1. элементы списка должны отображаться на узлы так, чтобы имя этих узлов совпадало со строковым значением элемента
2. отображение должно быть инъективным, т.е. разным s-выражениям следует сопоставлять разные деревья, иначе пользователь библиотеки будет ограничен в возможности максимально эффективно использовать её для поиска
3. желательно, чтобы при обратном отображении узлам дерева сопоставлялись реальные части отображаемого списка(s-выражения)

Теперь рассмотрим отображение s-выражения на XPath модель данных:

- Список отображается на узел элементов.
 1. Если голова списка является символьной переменной – имеет тип `symbol`(с непустым именем), то создаётся узел элемента с таким же именем, как имя символьной переменной, и элементы, находящиеся в хвосте списка, становятся непосредственными потомками этого элемента.
 2. В противном случае создаётся узел элемента с пустым именем, и все элементы списка становятся непосредственными потомками данного узла.
- атомарное значение, имеющее тип строка, число, литера(`character`), или тип `boolean`, отображается на текстовый узел, текстовым значением которого является текстовое представление данного атомарного значения
- символьное значение (типа `symbol`) отображается на узел инструкции обработки, строковое значение которой совпадает со строковым значением символьной переменной (типа `symbol`)
- атомарное значение, имеющее тип процедура (`lambda`-выражение), вектор (`vector`), или какой-либо другой тип, отличный от рассмотренных выше, отображается на узел комментария
- в данном отображении отсутствуют узлы атрибутов и узлы пространства имен. Соответственно, расширенное имя для каждого узла всегда совпадает с сокращённым именем.
- согласно XPath модели данных, корнем дерева документа является специальный корневой узел. В соответствии с этим требованием, для s-выражения создаётся виртуальный top-узел, сформированный как (*TOP* sexp).

Данное отображение удовлетворяет сформулированным требованиям; единственным исключением является то, что в случае отображения `dotted`-списка, мы получаем узел, имеющий имя "." и не соответствующий никакому

элементу s-выражения. А именно, отображая dotted-список, среди его детей мы вводим искусственного ребёнка, означающего "." (cons).

Приведённое выше s-выражение отображается на следующий XML:

```
<define>
  <my-hello><?who?><?.?><?rest?></my-hello>
  <display>Hello, _</display>
  <display><my-string><my-string><?who?></my-string></my-string>
  </display>
  <display>!&#xa;</display>
</define>
```

Можно заметить, что описанное отображение может породить недействительный XML документ, или формально нарушать XPath модель данных. Так, в данном примере мы получаем недействительный XML документ, т.к. инструкция обработки не может иметь имя ".". Также, данное отображение может породить узлы с пустым именем или с именем, содержащим недопустимые с точки зрения XML символы.

Например, s-выражение ((a b) "ccc") отображается на следующий XML: `<*empty*><a><?b?>"ccc"</*empty*>`, где `*empty*` соответствует узлу с пустым именем. Символьные данные в Scheme могут иметь практически произвольные имена: так, допустимым именем для функции может быть имя `prefix:name`, что недопустимо для имени XML узла (учитывая, что namespaces отсутствуют в нашей модели), или `f!`, `f?`, `f@`, `a1:a2:a3`. Допустимым s-выражением является также такое: `(set! a 1)`, где `set!` – ключевое слово языка Scheme [6], однако XML узел не может иметь имя "set!".

Несмотря на это, такая ситуация не является для нас проблемой, ведь мы собираемся не производить сериализацию s-выражений в XML, а работать с «виртуальным» XML. Мы всего лишь ослабляем непринципиальные требования, при этом сохраняя возможность поиска при помощи XPath даже для узлов со специфическим именем. Для нахождения узлов с пустым именем можно написать такой XPath запрос: `"//*[name()='']"`. При генерации XPath запроса для узлов с недопустимыми символами могут возникнуть некоторые проблемы, но они также решаются, подробнее об этом будет рассказано позднее.

К нарушению XPath модели данных можно отнести то, что наше отображение позволяет появление подряд нескольких текстовых узлов, что имеет место в случае отображения s-выражения `(list 1 2 3)` на XML документ `<list> 1 2 3 </list>`, но это нарушение также является лишь формальным.

Возвращаясь к нашему примеру, можно заметить в определении функции "my-hello" лишний вызов функции "my-string". Предположим, что мы

реализуем оптимизатор кода, который ищет вхождения "(my-string (my-string object))" и заменяет их на "(my-string object)". Конечно, эту функциональность можно реализовать, используя стандартные возможности языка Scheme, однако, используя XPath запрос "//my-string[my-string]", она решается быстро и элегантно. Этот запрос является частным случаем запроса "//a[b]", к которому мы ещё вернёмся. Он ищет все такие списки, у которых голова – это символ 'a', а в хвосте есть список, начинающийся с символа 'b'. Или, в терминах XPath, все узлы, имеющие имя 'a', и среди своих детей имеющие узел типа элемент с именем 'b'.

4.2. Generative XPath

Generative XPath (GXPath) [23] – это XPath 1.0 процессор, который может быть адаптирован для различных иерархических структур данных и для различных языков программирования. Особенностью этого подхода является то, что XPath запрос интерпретируется специальной виртуальной машиной.

GXPath состоит из двух основных компонент:

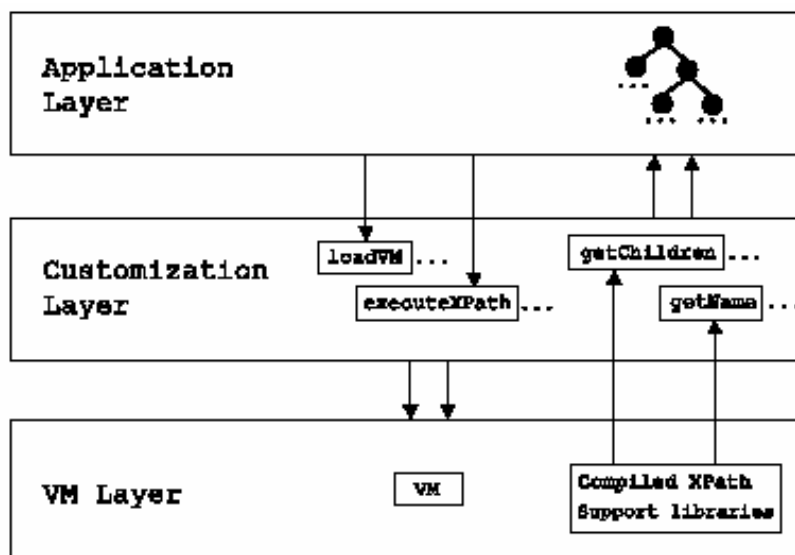
- компилятор XPath запроса в инструкции виртуальной машины, которые интерпретируются при выполнении запроса; компилятор реализован на языке Scheme
- среда выполнения



Среда выполнения(runtime environment) в свою очередь может быть разделена на следующие логические части:

- прикладной уровень – та часть программы, которая использует XPath для своих иерархических данных
- уровень виртуальной машины, интерпретирующей XPath (VM layer)
- промежуточный уровень между приложением и виртуальной машиной (customization layer)

Уровень виртуальной машины состоит из самой виртуальной машины и библиотеки исполнения, содержащей примитивы для выполнения запроса XPath. Сама виртуальная машина является виртуальной машиной для языка Scheme R5RS [6], и использует некоторые расширения SRFI [24]. В общем случае, для того, чтобы использовать GXPath из произвольного языка программирования, необходимо, чтобы некоторая Scheme машина была интегрирована в приложение.



Архитектура GXPath

Прикладной уровень взаимодействует с уровнем виртуальной машины:

- для её инициализации
- для исполнения XPath запроса

Промежуточный уровень `customization layer` служит для организации взаимодействия между прикладным уровнем и виртуальной машиной. Основной его задачей является отображение иерархической структуры, с которой имеет дело приложение, в структуры виртуальной машины. Именно на этом уровне происходит процесс виртуализации XML.

Уровень виртуальной машины использует `customization layer`:

- для получения коллекции узлов, находящихся в некотором отношении с данным узлом (соответствует XPath осям)
- для сравнения узлов в порядке появления в документе
- для получения свойств узлов, таких как имя, строковое представление и др.

Для уровня виртуальной машины совершенно безразлично, что именно представляют собой узлы и каково их внутреннее представление. Виртуальная машина трактует их как «чёрные ящики»(black box).

Типы данных XPath `"boolean"`, `"number"` и `"string"` отображаются на соответствующие типы данных виртуальной машины, т.е. на родные типы для Scheme; множество узлов в XPath соответствует списку в языке Scheme.

Код, порождаемый компилятором XPath, а также библиотека выполнения скомпилированного запроса, также являются программами на языке Scheme, и не содержат сложных особенностей, присущих каким-то конкретным реализациям; среди расширений языка используются следующие:

- SRFI 8 - Binding to multiple values
- SRFI 13 - String Library
- SRFI 42 - Eager Comprehensions

Первоначально GXPath был реализован для Guile [10] интерпретатора Scheme, однако совсем недавно он был перенесён для других интерпретаторов языка Scheme – SISC, MzScheme, Bigloo.

Таким образом, для использования GXPath необходимо

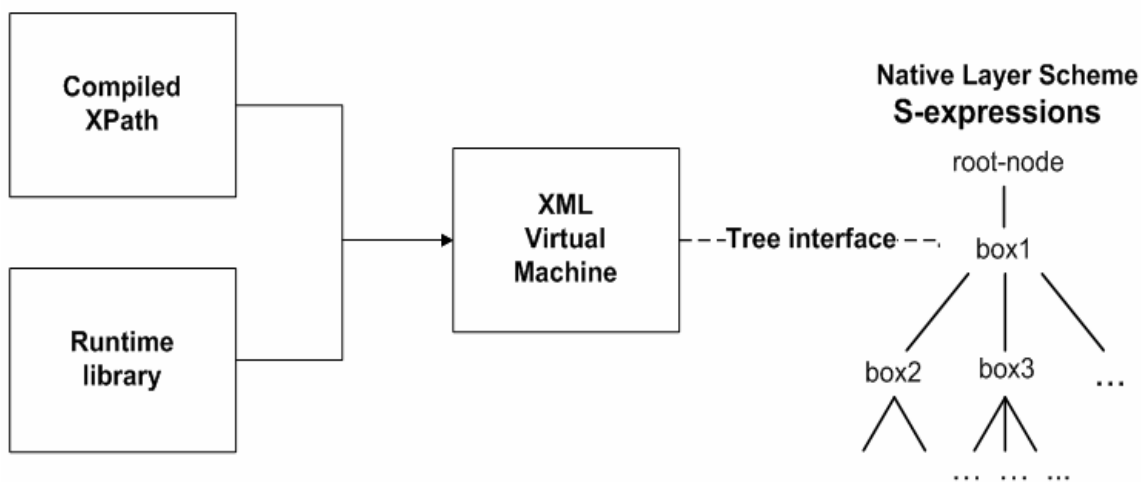
- найти подходящую реализацию виртуальной машины
- реализовать промежуточный уровень(customization layer)

Для предоставления интерфейса к иерархическим данным должны быть реализованы следующие функции, являющиеся частью customization layer-a:

- **box** `gx-ffi:root(box c)` на входе получает узел(box) и возвращает корневой узел документа
- **list** `gx-ffi:axis-<name>(box c, symbol node-test, string/symbol namespace-uri, string/symbol local-name):` семейство из 13 функций, соответствующих осям XPath, где <name> - это имя оси, семантика этих функций абсолютно полностью соответствует семантике осей XPath – они возвращают множество узлов(node set), а на вход получают контекстный узел и тест узла.
- **number/boolean** `gx-ffi:<=>(box n1, box n2)` функция сравнивает узлы в порядке их появления в документе
 1. -1, если узел n1 предшествует узлу n2
 2. 0, если узлы совпадают
 3. 1, если узел n1 идёт после узла n2
 4. #f может быть возвращено в случае, если узлы из разных документов
- ряд функций, эквивалентных функциям XPath: `string string(box n)`, `string gx-ffi:local-name(box n)` и некоторые другие

4.3. Технические аспекты реализации XPath над S-выражениями

С целью реализации XPath над s-выражениями в данной работе был выбран проект GXPath. Основной задачей является реализация промежуточного уровня(customization layer) на основе описанного отображения s-выражений на XPath модель данных. Первоначально в box помещается только корень дерева. Все остальные узлы будут обёрнуты в box только по необходимости, при вызове функций `gx-ffi:axis-<name>`. Продолжая аналогию с технологией Virtual XML Garden, здесь виртуализацию обеспечивает интерфейс customization layer-a с его функциями осей и свойств узлов – аналог легковесных адаптеров в [20].



4.3.1. Boxes

Пожалуй, в первую очередь должно быть описано устройство узла, «чёрного ящика». Его внутреннее строение безразлично для GXPath, однако важно для функционирования customization layer-a, т.к. узлы должны содержать информацию, необходимую для работы промежуточного уровня и для возвращения обратно к родному для приложения представлению узла – s-выражению. Это требование определяет первую компоненту box-a, хранящую соответствующее s-выражение.

Первоначально в движок GXPath попадает корень дерева s-выражения, затем новые узлы будут попадать при вызове функций осей `gx-ffi:axis-<name>`, причём соответствующие s-выражения будут оборачиваться в box только при необходимости. Таким образом, достигается «ленивое» отображение.

Новые узлы могут порождаться только прямыми осями, но не обратными. Это связано с проблемой s-выражений, с которой столкнулись также создатели SXPath для SXML – отсутствие в s-выражении указателя на родителя. Одними из возможных путей её решения в рамках SXML рассматривались варианты хранения дополнительной служебной информации в SXML узле [25]. В нашей ситуации нет необходимости добавлять служебную информацию в s-выражение, т.к. её можно хранить в box-e. Поэтому в качестве второго параметра, хранящегося в box-e, выступает ссылка на box-родитель. Таким образом, каждый узел, кроме корневого, имеет ненулевую ссылку на своего родителя, и далее на родителя родителя, что позволяет восстановить путь до корня включительно. Это свойство обеспечивается простым утверждением:

При вычислении выражения языка XPath над некоторым XML-документом, для того, чтобы выбрать в документе некоторый

контекстный узел, всегда необходимо сначала посетить все узлы, которые являются для него предками. [26].

Помимо простоты, данное решение обладает следующим преимуществом. После выполнения XPath запроса, к полученным узлам box-ам в дальнейшем можно снова применить XPath запрос, т.к. они хранят информацию о родителях. Таким образом, имеется возможность вычисления XPath выражения относительно не корня документа, а каких-либо нижележащих узлов, полученных в результате поиска. В случае же SXPath, полученные после поиска SXML узлы теряют информацию о предке.

В узле можно хранить и другую служебную информацию, необходимую в процессе вычисления XPath запроса. Для корректного выполнения XPath запросов необходимо иметь возможность сравнивать порядок следования узлов в документе. С этой целью в интерфейсе GXPath определяется функция `gx-ffi:<=>`. Для её эффективного вычисления используется специальная нумерация.

Итак, box представляет собой тип данных вектор (vector) с тремя полями, содержащими:

1. соответствующее узлу s-выражение
2. ссылку на box предка
3. номер для организации эффективного сравнения узлов

Для создания по s-выражению box-а служит функция `box gx-ffi:box-root(sexp)`, все остальные box-ы порождаются в customization layer-e функцией `box gx-ffi:box(sexp parent box-num)`. Функция `gx-ffi:box-root` создаёт корневой узел документа, оборачивая s-выражения в (*TOP* sexp).

Имея box, по нему можно получить соответствующее s-выражение, применив функцию `sexp gx-ffi:unbox(box)`.

Названия "box" в первую очередь происходит от понятия «чёрный ящик», однако может вызвать аналогии с технологией .NET и с средой CLR, для которой широко известными являются термины «боксинг» и «анбоксинг», означающие запаковывание и распаковывание элементарных типов данных. В данном случае эта аналогия совершенно уместна, как с точки зрения упаковывания/распаковывания, так и с точки зрения производительности: так же, как и в CLR, функция `gx-ffi:box` создаёт новый объект – вектор, и является более дорогостоящей операцией, чем `gx-ffi:unbox`, которая всего лишь извлекает из вектора данные.

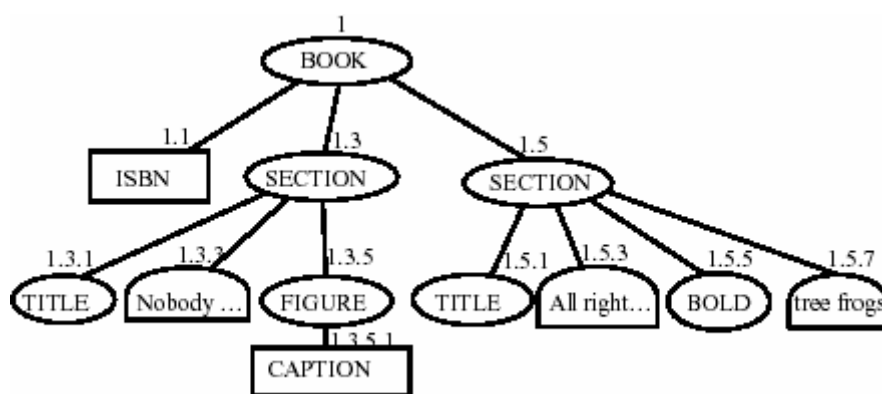
4.3.2. Организация сравнения узлов

Для вычисления XPath запроса существенным оказывается порядок следования узлов, при применении предикатов, содержащих положение близости (proximity position). Результат, возвращаемый функциями-осями

промежуточного уровня, всегда отсортирован в требуемом стандартном порядке. Однако этого не всегда достаточно, например, для запроса '//CCC | //BBB' сортировки осей недостаточно; при слиянии результатов поиска '//CCC' и '//BBB' необходимо сравнивать узлы.

С целью эффективного сравнения узлов, в данной работе применяется техника ORDPATH[27]. Согласно ORDPATH, каждому узлу в XML документе присваивается уникальный номер, позволяющий сравнивать следование узлов в порядке появления их в документе. Примером номера в формате ORDPATH является "1.5.3.9.1". Для эффективного хранения и сравнения номеров предлагается использовать сжатое бинарное представление. Также одним из свойств метода ORDPATH является возможность вставки и удаления узлов в произвольном месте XML документа без перенумерации остальных узлов (в данной работе это свойство не применяется). В этом случае могут возникать строки с отрицательными номерами (например, "1.5.3.-9.11"), что тоже поддерживается данным методом.

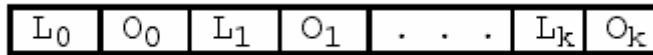
```
<BOOK ISBN="1-55860-438-3">
  <SECTION>
    <TITLE> Bad Bugs</TITLE>
    Nobody loves bad bugs.
    <FIGURE CAPTION="Sample bug"/>
  </SECTION>
  <SECTION>
    <TITLE> Tree Frogs </TITLE>
    All right-thinking people
    <BOLD> love </BOLD> tree frogs.
  </SECTION>
</BOOK>
```



При нумерации дерева номер узла формируется как номер родительского узла плюс номер узла в порядке следования соседних узлов(sibling), начиная с 1, используются нечётные числа (этот момент важен только для обеспечения вставки и удаления узлов). Так, если родитель имеет номер

"1.5", то дети получают номера "1.5.1", "1.5.3", "1.5.5" и т.д. Такой принцип формирования номеров также позволяет с помощью номера определять отношение между узлами «родитель-сын», «братья».

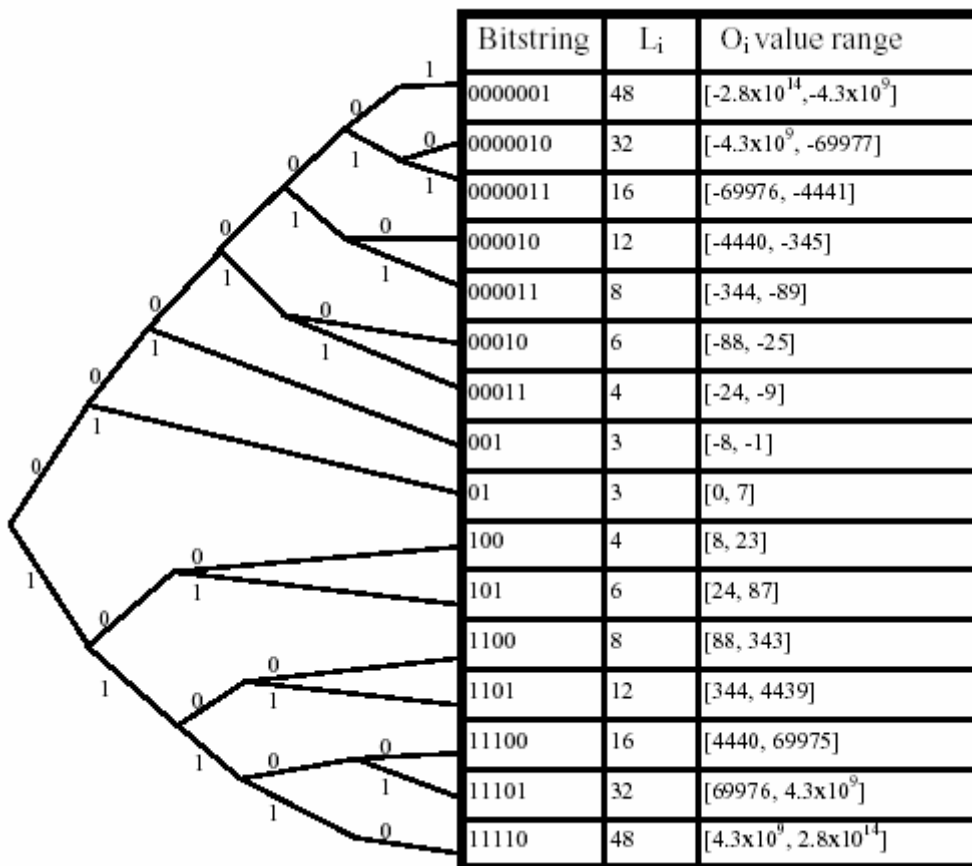
Теперь рассмотрим технику битового представления ORDPATH номеров.



В каждой паре L_i/O_i компонента L_i хранит длину последующей битовой строки O_i , которая является частью номера строкового представления ORDPATH. Битовая строка L_i является префиксным кодом (никакая строка не может быть префиксом другой строки), благодаря чему всегда можно определить, где она заканчивается. В таблице битовых значений каждому значению L_i сопоставляется длина строки O_i и ранг возможных значений O_i .

Например, для строки "1.5.3.-9.11" битовая строка будет выглядеть как

01 001 01 101 01 011 00011 1111 100 0011
 $L_0=3$ $O_0=1$ $L_1=3$ $O_1=5$ $L_2=3$ $O_2=3$ $L_3=4$ $O_3=-9$ $L_4=4$ $O_4=11$



Строка $L_0 = "01"$ означает, что O_0 будет иметь длину три, а битовые строки (000, 001, 010, . . . , 111) будут соответствовать числам из диапазона (0, 1, 2, . . . , 7). Строка "01001" декодируется в значение 1, а "01101" – в 5. $L_4 = "100"$ означает, что строка O_4 будет иметь длину и значение в диапазоне [8; 23]. Так, числу 8 соответствует строка "0000", 9 - "0001" и т.д. до 23 - "1111". Поэтому строка "1000011" декодируется в число 11.

Эта конструкция позволяет определять порядок узлов, сравнивая побитово битовые строки, без декодирования значений. Таким образом, для реализации сравнения и формирования новых битовых строк нужны только арифметические и основные битовые операции над числами. Согласно стандарту R5RS [6], различные реализации языка Scheme поощряются, но не обязаны поддерживать числа неограниченной длины. Так, в Guile такие числа поддерживаются, а в Bigloo числа могут иметь длину в 32 или 64 бита. Для Guile сравнение было реализовано с использованием стандарта SRFI-60 "Integers and bits" [28], который определяет набор побитовых операций, а также полагает, что числа имеют неограниченную длину. Для Bigloo и других интерпретаторов, не поддерживающих это свойство, требуется отдельная реализация.

Таким образом, функция `gx-ffi:<=>` сравнивает два узла, сравнивая их ORDPATH номера. Т.к. теоретически могут сравниваться два узла из разных документов (что дозволительно для XSLT), то 0 возвращается, только если совпадают номера узлов, и s-выражения равны по предикату `eq?`.

4.3.3. Реализация функций осей XPath

Все функции осей и свойств узлов реализованы, отталкиваясь от определения отображения s-выражения в XPath модель данных, в соответствии с требованиями стандарта XPath. Прежде всего, оси должны быть отсортированы в порядке появления узлов в документе, за исключением четырёх обратных осей, которые сортируются в обратном порядке: "ancestor", "ancestor-or-self", "preceding", "preceding-sibling". Из 13 осей реализованы 11, т.к. оси `attribute` и `namespace` отсутствуют. Благодаря тому, что s-выражение содержит внутри себя структуру всего поддерева, отображение получается «ленивым».

Опуская технические подробности, «боксинг» и «анбоксинг», фильтрацию узлов, узлы с пустыми именами и другие случаи, можно сказать, что функции осей XPath для customization layer-a реализованы следующим образом:

- **child**: если узел имеет непустое имя, то дети – это `cdt` списка, иначе они содержатся в самом списке

- **descendant(descendant-or-self)** получается путём полного обхода дерева s-выражения
- для получения братьев берётся родитель, и среди его детей всё, что предшествует узлу (используя `ordpath`), – это его **preceding-sibling**, а всё, что идёт после – **following-sibling**
- для получения родителя (ось **parent**) используется поле `box-a`, **ancestor (ancestor-or-self)** – (сам узел), предок, предок предка и т.д. – получают последовательным извлечением из `box-a` свойства родитель
- ось **following** формально определяется таким образом: «в документе, где располагается текущий узел контекста, находит все узлы, которые записанные после узла контекста. В число отобранных не попадают потомки текущего узла контекста, а также узлы атрибутов и пространств имен». Ось реализована следующим способом: на пути от данного узла к корню, начиная от узла, для каждого из узлов вызывается `following-sibling`, добавляется в результат, и также в результат добавляются все `descendant` для полученных `following-sibling`. Аналогичным образом реализована ось **preceding**.
- ось **self** даёт список, состоящий из данного узла

4.4. Применение библиотеки

Для применения XPath запроса необходимо вначале скомпилировать запрос в код виртуальной машины(Scheme+runtime library). Код может быть порождён динамически при помощи GXPath процедуры `gx:xpath-to-code-full`, и трансформирован после небольших манипуляций в функцию при помощи Scheme процедуры `eval`. Однако, если запрос известен заранее, лучше сгенерировать его и сохранить как функцию. В состав GXPath входит утилита, в режиме командной строки печатающая код запроса. Обернув такой код в процедуру, выполняющую запрос, можно следующим образом:

```
(define (xpathNNN c)
... original code goes here ...
),
```

где `c` – контекстный узел в виде `box-a` для выполнения XPath запроса.

Для удобства был создан макрос, генерирующий функцию, имеющую имя `xpath-name`, для запроса `xpath-expr`. Этот макрос находится в файле `codegen.scm`; код функции можно получить, добавив в него новый запрос и запустив его.

Для данной библиотеки был реализован целый ряд тестов, и все XPath-функции для тестов были получены таким образом.

Для того чтобы исполнить XPath-функцию, вначале необходимо создать `box` для *s*-выражения, соответствующий корневому узлу XML документа. Это можно сделать при помощи функции `gx-ffi:box-root`. Также в функцию `xpathNNN` может поступить любой другой `box`, полученный ранее в результате поиска.

Результат поиска – всегда список, может интерпретироваться следующим образом [23]:

- если список пустой, то результат – пустое множество узлов
- если тип первого элемента – не `box` (в данном случае не `vector`), то результатом является этот элемент
- иначе, результат – множество узлов в виде `box`

Можно предложить общий рецепт по применению XPath для *s*-выражений. Предположим, у нас есть список, и нужно найти в нём какие-то данные, удовлетворяющие некоторому условию. Вначале необходимо перевести это условие с языка списков язык XPath модели данных, используя построенное отображение. Далее нужно написать XPath запрос для поиска, и после поиска можно вернуться обратно к спискам.

На данный момент библиотека работает только под интерпретатором Guile, но её перенос для других реализаций Scheme является лишь вопросом времени, т.к. GXPath уже портирован под некоторые из них. Основной частью библиотеки, требующей модификации для других интерпретаторов, является реализация `ordpath`, т.к. не все интерпретаторы поддерживают числа неограниченной длины, а также имеют свой набор битовых операций.

4.5. Ограничения библиотеки

Данную библиотеку можно рекомендовать для использования в тех случаях, когда необходимо производить поиск по *s*-выражениям, имеющим древовидную структуру. Наоборот, использование её для плоских *s*-выражений не имеет смысла.

Одним из ограничений библиотеки может оказаться то, что в XPath модели данных не происходит различия между строковым типом данных и числами, строками и булевыми значениями, которые можно отличить в *s*-выражении. На эту особенность указывают в одной из дискуссий, сравнивающих XML и *s*-выражения [9]. Например, если нужно найти все числа 1 в *s*-выражении, то может быть написан запрос `//text()[(string(.)="1")]`, однако он может найти и строки "1". В случае необходимости в runtime GXPath можно добавить функцию `is-number` и записать запрос так: `//text()[is-number(.) and (string(.)="1")]`. Это приведёт к расширению множества функций XPath, однако такой обходной путь возможен.

4.6. Примеры использования

Прежде чем привести примеры, хотелось бы акцентировать внимание на основном мотиве, послужившем причиной создания библиотеки. Прежде всего, это повышение эффективности труда программиста. Вместо частичных и неполных решений, навигация при помощи XPath позволяет быстро, легко и элегантно решать множество задач, возникающих при работе со списками. Также такой подход позволяет разделить логику приложения от технической части – обработки списков.

Пример 1

Показательным в этом смысле является пример, приведённый в главе «Отображение S-выражений на XPath Data Model». Обобщая тот пример, рассмотрим следующую задачу: для данного списка найти все входящие в него списки, у которых голова – это символ 'a', а в хвосте есть список, начинающийся с символа 'b'. В терминах XPath модели данных, нужно найти все узлы, имеющие имя 'a', и среди своих детей имеющие узел типа элемент с именем 'b'.

Так, для `(c d a e (a k (b x)) (a (b)) (a (c b)) (b))` должно возвращаться `(a k (b x)) (a (b))`, для `(a (b) c) - (a (b) c)`, а для `(a k (b (a (b c)))) - (a k (b (a (b c))))` и `(a (b c))`.

Задача решается при помощи запроса `"//a[b]"`.

Эта сравнительно простая задача была предложена для решения в русском сообществе Scheme программистов. Неожиданно оказалось, что решить её получалось тяжело, причём не самыми плохими программистами, с ошибками и не с первого раза. В конечном итоге было предложено решение с использованием классических средств языка Scheme (операций `car` и `cdr`), 20 строк кода, и решение с использованием `pattern matching`, 9 строк кода. Надо заметить, что `pattern matching` упростил решение задачи, но не избавил от написания рекурсивных вызовов, обходящих дерево [см. приложение]. Автор данной работы также предпринял попытку написать решение (27 строк кода) [см. приложение], но без обработки `DottedPair`-списков. Характерным моментом для подобных задач является то, что не всегда оказывается удобно учитывать появление `DottedPair`-списка. Однако реализованная в данной работе библиотека работает и в этом случае.

Всем решениям присущ один общий недостаток – в случае небольшого изменения условия задачи нужно переписывать всё заново, заново тестировать. При применении XPath процесс изменения и тестирования также присутствует, но заметно облегчается.

Пример 2

Если предыдущий пример был скорее учебным, то следующая задача реально возникла на практике. Ранее уже упоминалось, что символьные

данные в языке Scheme могут содержать недопустимые с точки зрения имён XML узлы символы, например `f!`, `f@`, `a1:a2:a3`. Имена переменных могут содержать внутри себя пробелы – такие символы можно записывать при помощи `|symbol|`, например `|a b c|`. Также такие символьные переменные могут быть получены при помощи функции `string->symbol`. Согласно отображению, символы отображаются в имена XML узлов, и для таких узлов оказывается невозможным записать XPath запрос. Попытка скомпилировать такие XPath запросы при помощи компилятора GXPath приводят к ошибке. Есть возможность записывать запрос в виде `//*[name()='set!']/*[name()='a:b']`, но это не очень удобно. Был придуман обходной путь. Можно использовать кодирование неудобных символов с помощью `%XX`, где XX – шестнадцатеричный код символа. Таким образом, вместо проблемного запроса `"/set!/a:b"` можно скомпилируем такой запрос: `"/set%21/a%3Ab"`.

Затем полученный код необходимо раскодировать. После компиляции XPath запроса получается код на языке Scheme в виде s-выражения. Для этого s-выражения можно применить XPath, чтобы найти в нём узлы, потенциально требующие раскодирования. Эти узлы – текстовые узлы, дочерние для элементов, имя которых начинается с `'gx-ffi:axis-'`.

Например, команда `sh ./run.sh "/set%21/a%3Ab"` порождает такой код:

```
(gx:sidoaed
  (let ((c (gx-ffi:root c)))
    (list-ec
      (:list c
        (list-ec
          (:list c
            (gx-ffi:axis-descendant-or-self c 'node '* '*))
            (:list c2
              (gx-ffi:axis-child c '* '* "set%21"))
              c2))
          (:list c2
            (gx-ffi:axis-child c (quote *) (quote *) "a%3Ab"))
            c2)))
```

Чтобы найти эти узлы, достаточно исполнить запрос `//*[starts-with(name(),'gx-ffi:axis-')]/text()`. Затем для каждого из узлов необходимо проверить, должен ли он быть раскодирован, и если да, то произвести соответствующую замену. В результате вхождения `"set%21"` и `"a%3Ab"` будут заменены на `"set!"` и `"a:b"` соответственно. В ходе решения этой задачи было обнаружено, что существует необходимость по `box`-у получить `pair`, для которой значение `box-a` было бы `car` частью `pair`. Так, список `(gx-ffi:axis-child c '* '* "set%21")` в полной форме выглядит как `(gx-ffi:axis-child . (c . ('* . ('* . ("set%21" . `()))))`. Для того, чтобы заменить `"set%21"` на `"set!"`, нужно получить `pair` `("set%21" . `())` и для неё выполнить `(set-car! (find-my-pair name-box) new-name)`, где `name-box` – это `"set%21"`.

В самом box-e эта информация не хранится, получить её можно несколькими способами – либо посредством изменения структуры самого box-a, либо с помощью поиска. В рамках данной задачи было реализовано упрощённое решение функции find-my-pair, более оптимальное оставлено на будущее. Хотелось бы заметить, что найденный недостаток показывает, что реализованная библиотека уже реально применяется, т.к. ошибки и недостатки не находятся только в неиспользуемых программах. Особенно хочется заметить, что решение применить XPath для поиска в s-выражении было не политическим решением использовать везде, где это возможно, реализованную библиотеку, а вполне естественно возникшим желанием использовать её для решения данной задач – оказалось, что эту задачу действительно удобно и просто решить с её помощью.

Пример 3

Ещё одной возможной областью применения библиотеки является оптимизация кода, генерируемого XPath компилятором. В данный момент запрос компилируется прямолинейно в неоптимальный код. В коде возможен целый ряд конструкций, которые можно упростить. Рассмотрим этот процесс на следующем примере: '1+2' компилируется в

```
(gx:sidoaed
  (gx:unit
    (+ (gx:number (gx:sidoaed (gx:unit 1)))
       (gx:number (gx:sidoaed (gx:unit 2))))))
```

Название функции gx:sidoaed расшифровывается как (generative xpath) sort in document order and eliminate duplicates. Именно это она и делает. Функция gx:unit упаковывает переданное значение в список из одного элемента. Ясно, что любая комбинация (gx:sidoaed (gx:unit xxx)) всегда может быть заменена на (gx:unit xxx). С помощью запроса //gx:sidoaed[gx:unit] можно найти такую комбинацию, и затем преобразовать её. На этом шаге функция в примере упрощается до

```
(gx:unit
  (+ (gx:number (gx:unit 1))
     (gx:number (gx:unit 2))))
```

Функция gx:number извлекает число из списка. Комбинация (gx:number (gx:unit xxx)) может быть найдена при помощи запроса //gx:number[gx:unit] и преобразована в xxx. Код упрощается до

```
(gx:unit
  (+ 1
     2))
```

Теперь, выполнив запрос //+[count(node())=count(text())], можно найти узел с именем "+", имеющий только числа среди своих детей, и сложить их. После замены узла "+" на результат сложения, в конечном итоге код упрощается до (gx:unit 3).

Можно заметить, что многие XPath запросы для этого примера содержат недопустимые символы. Теперь их компиляция возможна благодаря «Примеру 2».

4.7. Оценка производительности

Актуальным является вопрос производительности реализованной библиотеки. Помимо измерений производительности самой библиотеки, был выбран также набор средств для сравнения. Надо сказать, что помимо реализованного решения, не существует ни одного средства для навигации по s-выражениям при помощи XPath. SXPath, как уже было указано ранее, позволяет работать только с SXML, но, однако полезно было сравнить производительность SXPath при выполнении над достаточно хорошим s-выражением, имеющему SXML-подобную структуру. Также конкурентами реализованной автором библиотеки, по части навигации по s-выражениям, является Pattern Matching, но, к сожалению, на данный момент данная библиотека реализовано только для интерпретатора Guile. К конкурентам относительно навигации по s-выражениям можно отнести и подход, основанный на использовании классических средств языка Scheme - `car` и `cdr`. Также интересным является вопрос, насколько замедляет инфраструктура, предоставляемая GXPath технологией, работу `custom layer-a`.

В связи с поставленными вопросами были произведены измерения выполнения двух типов запросов – запроса, обходящего всё дерево – был выбран запрос `:::list/list-ec`, и запроса, обращающегося только к верхушке дерева - `/AAA/CCC`. Для первого запроса в качестве входных данных было взято s-выражение, получаемое при компиляции запроса `/AAA/BVV/CCC/DDD/*//EEE` – s-выражение получилось довольно большим и приближённым к тому, где предполагается реально использовать реализованную библиотеку – для оптимизации компилируемого GXPath компилятором кода. Для второго запроса в качестве данных было взято не столь большое s-выражение, но это не должно иметь значения, т.к. запрос затрагивает лишь верхушку дерева. Сравнение проводилось для GXPath, для имитации выполнения этого же запроса только средствами Customization layer-a, для `ddo:txpath` (запрос пришлось заменить на `:::list/list-ec`, а структуру самого s-выражения немного изменить, чтобы оно походило на SXML). Для GXPath и `ddo:txpath`, код запроса генерировался заранее и в измерениях не участвовал. Результаты даны в миллисекундах, измерения были проведены так, что погрешность не превышает 5%.

	//:list/list-ec	/AAA/CCC
GXPath	7,4	0,5
ddo:txpath	2,7	0,09
Custom Layer	7	0,45
CustomLayer (оптимальный)	6,3	0,23

Под «Custom Layer (оптимальный)» имеется в виду более оптимальная реализация запроса средствами custom layer-a. А именно, генерируемый код для GXPath на данной его стадии не оптимален. Например, он сортирует множество возвращаемых узлов всегда, даже когда достаточно того, что оси custom layer-a отсортированы. Тесты «Custom Layer (оптимальный)» отличаются от «Custom Layer» тем, что в этом случае не выполняется излишняя сортировка. Из тестов следует, что инфраструктура GXPath не сильно замедляет работу custom layer-a, но здесь есть ресурсы для оптимизации.

Сравнивать именно с ddo:txpath (из возможных также sxpath и txpath) было решено потому, что именно он строго реализует стандарт XPath, поддерживает порядок узлов и не содержит дубликаты. Однако, как оказалось, ddo:txpath не намного быстрее (и не медленней) sxpath и txpath. По сравнению с ddo:txpath, реализованная библиотека работает медленней, но не на порядок, примерно в 3 раза. Можно объяснить это тем, что в рамках данной работы не ставилось своей целью реализовать самое быстрое средство для навигации по s-выражениям, оптимизации оставлены на будущее и являются темой отдельного исследования. Прежде всего, хотелось реализовать удобное средство, позволяющее работать со всем множеством s-выражений. Применение ddo:txpath для произвольного s-выражения требует дополнительных усилий для трансформации его в SXML, к тому же SXML может быть уже совсем другим s-выражением.

В приложении содержится функция findABL, которая решает задачу //a[b], тоже порождающего обход дерева, с использованием классических средств Scheme. Для неё то же самое выражение, полученное как результат компиляции /AAA/BVV/CCC/DDD/*//EEE, было изменено - каждое вхождение :list на a, list-ec - на b. В результате измерений было получено 0,4 миллисекунды на запрос. Против GXPath есть замедление более чем в 10 раз. Однако при небольшом изменении условия задачи нужно переписывать функцию findABL снова и снова, и усилия по её написанию и поддержке довольно велики. К тому же функция findABL не работает с DottedPair-списками.

Здесь замедление – следствие абстракции высокого уровня. В ущерб скорости работы программист получает средство, повышающее скорость разработки и экономящее его труд. Скорость работы библиотеки можно

отнести к одному из ограничений, если производительность является узким местом.

5. Заключение

В рамках данного дипломного проекта был предложен новый подход для работы со списками языка Scheme. Был проведён обзор методов, используемых для решения этой задачи, и показано, что можно по-новому подойти к задаче навигации по структуре s-выражений. С точки зрения XML технологий этот вопрос может быть рассмотрен в свете виртуализации иерархических данных в XML. Было представлено отображение произвольного s-выражения в XPath Data Model, и предложено использовать этот факт для навигации по s-выражениям при помощи стандарта XPath. Конечным результатом является библиотека, реализованная при помощи исследовательского проекта Generative XPath. Был приведён ряд примеров, показывающих удобство применения представленного подхода. Основным преимуществом данного подхода является повышение эффективности труда программиста. Другим положительным аспектом является использование общеизвестного стандарта XPath. Данная библиотека может быть портирована на множество различных Scheme интерпретаторов, что также имеет ценность. Однако платой за повышение уровня абстракции является снижение производительности. Вопрос улучшения производительности является актуальным и будет решён в рамках последующих исследований.

Хотелось бы отметить, что реализованная библиотека имеет практическую ценность, и используется уже сейчас.

6. Список литературы

- [1] Харольд Абельсон, Джеральд Джей Сассман, при участии Джули Сассман «Структура и интерпретация компьютерных программ». Добросвет, 2006
- [2] Ess Expressions. <http://c2.com/cgi/wiki?EssExpressions>
- [3] World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommendation 04 February 2004. <http://www.w3.org/TR/2004/REC-xml-20040204/>
- [4] World Wide Web Consortium. XML Path Language (XPath) Version 1.0, W3C Recommendation 16 November 1999. <http://www.w3.org/TR/1999/REC-xpath-19991116>
- [5] XML Information Set (Second Edition), W3C Recommendation 4 February 2004. <http://www.w3.org/TR/2004/REC-xml-infoset-20040204>
- [6] R. Kelsey, W. Clinger, J. Rees (eds.), Revised5 Report on the Algorithmic Language Scheme, Higher-Order and Symbolic Computation, Vol. 11, No. 1, August, 1998.
- [7] LispWorks Ltd. Common Lisp HyperSpec. <http://www.lispworks.com/documentation/HyperSpec/>
- [8] Lisp Vs Xml. <http://c2.com/cgi/wiki?LispVsXml>
- [9] Xml Isa Poor Copy Of Ess Expressions. <http://c2.com/cgi/wiki?XmlIsaPoorCopyOfEssExpressions>
- [10] Free Software Foundation. Guile (About Guile). <http://www.gnu.org/software/guile/guile.html>
- [11] Bigloo <http://www-sop.inria.fr/mimosa/fp/Bigloo/>
- [12] Pattern matching for Bigloo. <http://www-sop.inria.fr/mimosa/fp/Bigloo/doc/bigloo-7.html#Bigloo-pattern-matching-facilities>
- [13] Pattern Matching for Scheme. Andrew K. Wright and Bruce F. Duba
- [14] Лизоркин Д.А. и Лисовский К.Ю. SXML: XML-документ как S-выражение. Электронные библиотеки, 2003, Том 6, Выпуск 2.
- [15] Oleg Kiselyov. SXML Specification. <http://okmij.org/ftp/Scheme/xml.html#SXML-spec>
- [16] Лизоркин Д.А. и Лисовский К.Ю. Язык XML Path (XPath) и его функциональная реализация SXPath. Электронные Библиотеки, 2003, Том 6, Выпуск 4.
- [17] Oleg Kiselyov and Kirill Lisovsky. XML, XPath, XSLT implementations as SXML, SXPath, and SXSLT. International Lisp Conference: ILC2002, October 2002. <http://www.okmij.org/ftp/papers/SXs.pdf>
- [18] Dmitry Lizorkin. DDO SXPath. <http://modis.ispras.ru/Lizorkin/ddo.html>

- [19] IBM alphaWorks: Virtual XML Garden.
<http://www.alphaworks.ibm.com/tech/virtualxml>
- [20] Phantom XML. Kristoffer Rose and Lionel Villard, XML 2005 Conference.
<http://www.idealliance.org/proceedings/xml05/ship/80/paper.PDF>
- [21] Erik Meijer and Brian Beckman: XLINQ: XML Programming Refactored (The Return Of The Monoids), in Proceedings of XML 2005, November 2005. <http://research.microsoft.com/~emeijer/Papers/XMLRefactored.html>
- [22] Oleg Paraschenko. Reusing XML Processing Code in non-XML Applications Version 1.0, 16 April 2005.
<http://uucode.com/texts/genxml/genxml.pdf>
- [23] Oleg Parashchenko. Generative XPath. XML Prague 2007.
<http://xmlhack.ru/protva/generative-xpath.pdf>
- [24] The SRFI Editors: Scheme Requests for Implementation.
<http://srfi.schemers.org/>
- [25] O. Kiselyov. On parent pointers in SXML trees.
<http://www.okmij.org/ftp/Scheme/parent-pointers.txt>
- [26] Лизоркин Д.А. Оптимизация вычисления обратных осей языка XML Path при его реализации функциональными методами Труды Института Системного Программирования РАН, 2004 г.
- [27] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, Nigel Westbury. ORDPATHs: Insert-Friendly XML Node Labels, SIGMOD 2004
- [28] SRFI-60 "Integers and bits". <http://srfi.schemers.org/srfi-60/srfi-60.html>
- [29] Oleg Paraschenko. Why Not Sx Tools
<http://xmlhack.ru/protva/xquery/index.php/WhyNotSxTools?version=4>
- [30] Haruo Hosoya, Benjamin Pierce. Regular Expression Pattern Matching for XML. <http://www.cis.upenn.edu/~bcpierce/papers/tapat.ps>
- [31] Ilya Bagrak, Olin Shivers. trx: Regular-tree expressions, now in Scheme
<http://repository.readscheme.org/ftp/papers/sw2004/bagrak.pdf>

7. Приложение

В этом разделе содержится код на языке Scheme реализации примера №1. Решение на основе функций car и cdr.

```
(define (has-b-list? lst)
  (any          ; srfi-1
    (lambda (item)
      (and (pair? item)
            (eq? 'b (car item))))
    lst))

(define (candidate? lst)
  (and (pair? lst)
        (eq? 'a (car lst))
        (has-b-list? (cdr lst))))

(define (not-my-list? lst)
  (or (null? lst) (not (list? lst))))

(define (findABL lst)
  (let ((result '()))
    (set! result (filter-map
                  (lambda (lFilter)
                    (if (not-my-list? lFilter)
                        #f
                        (let ((tmp (findABL lFilter)))
                          (and (not (null? tmp)) tmp) ))
                    lst))
        (if (candidate? lst)
            (if (null? result)
                (set! result lst)
                (set! result (list lst result) )))
        result
    ))
```

Решение на основе pattern matching (приведено членом сообщества ru_scheme, http://community.livejournal.com/ru_scheme/19262.html), работает для интерпретатора Bigloo.

```
(define (find-a-b-lists l)
  (define (sub-lists l)
    (append-map (lambda(e)
                  (find-a-b-lists e))
                l))
  (match-case l
    ((a ??- (b ???-) ???-) (cons l (sub-lists l)))
    ((???-) (sub-lists l))
    (else '())))
```