Project Description

XPLite

In this section we describe XPLite, a path expression language you need to implement.

The syntax of XPLite is described using the following grammar:

$$\text{PathExpression} ::= (['/']\text{LocationStep})^*$$

$$\text{LocationStep} ::= \text{Axis}`::' \text{NodeTest}[`['\text{Predicate}`]']^*$$

$$\begin{aligned}\text{Axis} ::= \ & \texttt{self}| \\ & \texttt{parent}| \\ & \texttt{child}| \\ & \texttt{attribute}| \\ & \texttt{ancestor}| \\ & \texttt{descendant}| \\ & \texttt{following}| \\ & \texttt{preceding}\end{aligned}$$

$$\begin{aligned}\text{NodeTest} ::= \ & NodeName| \\ & \star| \\ & \texttt{node()}| \\ & \texttt{attribute()}| \\ & \texttt{text()}\end{aligned}$$

$$\begin{aligned}\text{Predicate} ::= \ & \text{LocationStep}('/'\text{LocationStep})^*| \\ & \text{PredicateExpression}[\text{ OP PredicateExpression}]\end{aligned}$$

```
PredicateExpression  ::=  function|
                          ConstantValue

           function  ::=  position()|
                          last()|
                          string(PathExpression)|
                          contains(PathExpression, string)|
                          count(PathExpression)|
                          not(PathExpression)|
                          true()
                          false()
                 OP  ::=  = | < | > | >= | <= | <>
```

Informally, XPLite is a (n almost a) subset of XPath that contains all its major axes (self, child, parent, descendant, ancestor, following, preceding, following-sibling, preceding-sibling and attribute), five types of node tests, and a simplified version of predicates. It *does not contain* short syntax conventions (except for the ”*” nodetest).

Each individual predicate is either an XPLite expression or a comparison between values of built-in functions and constants (you can compare a value of one function to a value of another function, as well as a value of a function to a constant; you can also compare constants, but this is less useful). Each location step can contain multiple predicates.

The semantics of XPLite is defined as follows.

- All expressions, except the expressions in the predicates are considered to be absolute, i.e., they start from the root of the XML repository specified in the command.

- As a special case, XPLite query of the form / returns the root of the repository, that is, the content of the entire XML repository[1].

- The XML tree on which XPLite expressions are evaluated has the following types of nodes:

  - **root.** The root of the repository is the only node of this type.
  - **element.** A node that represents an XML element.

[1] XPLite queries will be directly preceeded by the repository name: `RETURN document("RepositoryName")/` is the command to retrieve the root of the given repository. Notice that this is different from the semantics of XPath, where, to get the root element of a repository you have to issue the expression `document("RepositoryName")/child::*`.

- **text.** A node that contains text content only.
- **attribute.** A node that contains information about an attribute.

- The semantics of each **location step** is the same as in XPath: it is treated as a trasnformation of the input node set into the output node set based on the semantics of the individual components of the location step.
- The semantics of the **axes** follows XPath. To simplify things, we explicitely assume that nodes whose content is #PCDATA have no children or descendants. (That is, XML documents used in conjunction with this project will have no *mixed content*).
- `attribute` nodes are not reachable via `child` or `descendant` axes. They are reachable only via the `attribute` axis. However, if the context node is an attribute node, the element node to which it belongs is reachable from it via `parent` axis, and **only** `parent` axis.

  ```
  /descendant::foo/attribute::bar/parent::node()
  ```

  returns all foo nodes that have an attribute bar.
- The semantics of the **node tests** follows XPath XPath. The "*" nodetest is the abbreviation for the `node()` nodetest[2]. `text()` nodetest is satisfied by XML element nodes which have no descendants, but do have non-trivial content[3].

  For example, in the following document,

  ```
  <root>
       <a>This is a</a>
       <b>test</b>
  </root>
  ```

  element `<root>` has two children: `<a>` and `<b>`. Both satisfy the `text()` nodetest.
- The semantics of the **predicates** is as follows.
  - If a location step contains more than one predicate then, a node is added to the output node set iff all predicates are true on it (i.e., we use conjunction).
  - Predicates consiting of a sequence of location steps (essentially a nested XPLite Expression) are considered to be **relative** and are evaluated for each of the nodes in the input nodeset (context). They are evaluated

[2] This is the only abbreviation we allow in XPLite syntax

[3] I.e., those XML element nodes that have a record in the `ContentIndex` structure.

to **true** iff the XPLite expression yields a non-empty nodeset as the answer, otherwise, they evaluate to false.

Consider, for example the following expression.

```
/child::b[child::c]
```

This expression returns all children of the root named `b` that have child nodes named `c`.

- Predicates of the form $PredicateExpression$ are evaluated as follows. PredicateExpression is evaluated. If it evaluates to `true`, a non-zero number, a non-empty nodeset or a non-empty string, then the entire predicate expression evaluates to `true`. Otherwise, the predicate expression evaluates to `false`.

  Note: This particular syntactic construct will mostly be used with the `not()` function to allow for expressions of the form:

  ```
  /child::a[not(child::b)]
  ```

  in addition to

  ```
  /child::a[not(child::b) = true()]
  ```

  However, for consistency sake, expressions of the form:

  ```
  /child::a[position()]
  ```

  are now also allowed.

- Predicates of the form $PredicateExpression$ OP $PredicateExpression$ are evaluated as follows. Both the lefthandside and the righthandside predicate expressions are evaluated. Their values are compared using the specified operator. If the comparison holds, the expression evaluates to `true`, otherwise, it evaluates to `false`.

  The semantics of all operators is traditional. On boolean expressions, `true` > `false`, so, a predicate `[true() > false()]` shall evaluate to `true` (see the semantics of `true()` and `false()` below). On strings, only equality, =, and inequality, <>, are defined. If two strings are compared using any other comparison, expression evaluates to `false`.

- function `position()` returns the position of the current node in its current context - i.e., in the current node set. For example,

  ```
  /child::chapter[position()=4]
  ```

  returns the fourth `chapter` child of the root. At the same time,

  ```
  /descendant::f/parent::d[position()=2]
  ```

  looks for the node `<d>` which has a child `<f>` and which is the second such node in the document order. In the following example:

```
<root>
    <d><f>1</f></d>
    <d><g>2</g></d>
    <f><d>3</d></f>
    <d><f>4</f></d>
</root>
```

the abovementioned query will retrieve in the last element `<d>` (`<d><f>4</f></d>`).

The context is *always ordered in **document order***, i.e., the order of appearance of the starting tag of each node in the XML document[4].

- function `string()` has triple meaning. On nodes of type `text` (i.e., those that pass `text()` node test), this function returns the #PCDATA content of the node. On nodes of type `attribute` it returns the value of the current attribute. On all other nodes it returns empty string. Additionally, it can take as input a $PathExpression$. If a path expression is empty, the function operates on the context node.

  If the path expression is not empty, the path expression is executed, and the function operates on the first context node. The result of this function must be compared to a string value.

  For example, the following query

  ```
  /child::a[string(child::c) = "Hello!"]
  ```

  returns all `<a>` elements that are children of the root element whose first `<c>` child has content `"Hello"`. Consider the following XML document:

  ```
  <root>
     <a><c>Hello!</c></a>
     <a><b>Boo</b>
        <c>Hello!</c>
     </a>
     <a> <c>FooBar</c>
         <c>Hello!</c>
     </a>
  </root>
  ```

  The query above returns the first two `<a>` elements, **but not** the third.

- function `contains(PathExpression, string)` evaluates the string content of the path expression (i.e., essentially, evaluates the `string(PathExpression)` function) first, and then checks if the section argument is a substring of it. It returns `true` if the second argument is a substring of the string represented by the first argument, and `false` otherwise.

[4]Which, hopefully, coincides with NodeId in your StructureIndex.

- function `last()` returns the size of the current context, i.e. the number of nodes in the context nodeset (nodelist). Using the XML document from above, in the following expression

  ```
  /descendant::a[last()=3]
  ```

  `last()` will evaluate to 3 – there are three nodes `<f>` found in the document, they form the node set that `last()` gets as the input.

  At the same time,

  ```
  /descendant::a[position()=last()]
  ```

  returns only the last `<a>` node from the document (`<a> <c>FooBar</c> <c>Hello!</c> </a>`).
- function `count(PathExpression)` evaluates the PathExpression first, and then returns the size of the context (i.e., the nodeset retrieved by the path expression). Note that PathExpression here can be both *absolute* and *relative*. Absolute path expressions are evaluated from the root, relative path expressions are evaluated from the context nodes.
- function `not(PathExpression)` evaluates PathExpression and then takes the logical NOT of it. If PathExpression evaluates to `false`, empty nodeset, empty string or 0, `not(PathExpression)` evaluates to `true`. Otherwise, it evaluates to `false`.
- functions `true()` and `false()` return `true` and `false` respectively. XPLite does not have specific constants for `true` and `false`, so, these functions can be used as surragates for the truth values.

- XPLite constants are of two types: numeric and string. String constants are enclosed in double quotes. Numeric constants are proper integers, possibly negative.