IAG0050 An Analysis of Algorithmic Languages

Homework

Intentional XML Processing Language

Ilja Lutov
010691
IASM

Overview

Modern software development process tends to be more human-friendly compared to earlier days of computing when programmers had to write applications in machine languages. However, fundamental problem of managing complexity is still present despite a vast array of graphical integrated development environments, innovative languages, code generation frameworks and other convenient tools that help developers to create quality software. While information science advances, customer and market requirements also grow with the emergence of new technologies that allow implementation of more complex solutions in software business. This simple fact leaves developers with somewhat limited capabilities, because software demands are always a step ahead and require more advanced approaches and higher level abstractions. No matter how useful modern tools are, programmers still write code in general-puirpose languages that have small resemblance to natural human thinking. Every concept in a business domain has to be first converted into computer logic that is accepted by machine, and this is a difficult task which requires enormous amount of effort to accomplish. Code generators do a great job, but they are of a general nature, do not conceptualize well concrete domain entities, and generated code is usually incomplete as only basic building blocks are created while the major portion of the system requires manual programming. Ofcourse the are many domain-specific languages like SQL that follow a natural course of human thinking and allow painless transition of ideas into code, but these are usually used in well established domains and are taken for granted. For example, in web applications development, SQL is an essential part of the whole system, it greatly simplifies manipulations on data model by providing a natural syntax for queries and allowing programmers to focus on relevant data, not algorithms that write, search and retrieve raw bytes. But why everything else is being written in unfriendly machine languages like Java or PHP? Why all specification requirements must go through complex trnaslation process to be properly trnasformed into computer language? Why cannot we write at least some program parts in natural human language without straining our brains by constant encoding of simple concepts into computer code?

Language-oriented programming tries to answer those questions above. It is a style of computer programming in which, rather than solving problems in general-purpose programming languages, the programmer creates one or more domain-specific programming languages for the problem first, and solves the problem in those languages. Intentional and concept-oriented programming paradigms push the idea of domain-specific languages even further by greatly reducing concept to code transformation

phase which takes the largest amount of time in any project.

Intentional XML Processing Languages or simply IXPL is based on the ideas of intentional programming paradigm which is a collection of concepts that enable software source code to reflect the precise information, called intention, which programmers had in mind when conceiving their work. By closely matching the level of abstraction at which the programmer was thinking, browsing and maintaining computer programs becomes easier. The intention to create a natural XML processing language came from a real life project, namely a web application that was written using a mixture of PHP and Java programming languages. Due to architectural reasons, the sytem does not incorporate SQL database, but rather communicates with another system through web services to exchange data. However, local user settings and some other inforamtion must be preserved between sessions. For this purpose XML was chosen to store needed data in plain text files on a local hard drive. During the maintenance phase, as system growed larger, and customers wanted more features to be implemented, the need for local data storage also encreased, and XML data manipulations bacame difficult and time-consuming due to a general nature of corresponding libraries that operate on too low level to be conveniently used for simple tasks. Consider the following XML example:

```
<system_users>
	<user>
		<first_name>Ilja</first_name>
		<last_name>Lutov</last_name>
		<member_of>
			<department>
				<id>I123</id>
				<name>IT</name>
			</department>
			<department>
				<id>M234</id>
				<name>Management</name>
			</department>
		</member_of>
	</user>
	<user>
		<first_name>Toivo</first_name>
		<last_name>Põld</last_name>
		<member_of>
			<department>
				<id>F345</id>
				<name>Finance</name>
			</department>
		</member_of>
```

```
        </user>
</system_users>
```

Given XML document may contain arbitrary number of users and departments for each user. Although the content is simple, the number of steps required to create such document is bigger than expected, and the amount of effort needed to write appropriate code in general-purpose language is unnecessarily large. The main problem here concerns the way in which XML library processes documents, manipulates nodes, and handles content. Every node or element is treated as a separate object that cannot be modified in the context of the whole documents. To change the value of an element one must find a correponding node, create new one with the same name, and replace the existing node. Newly created nodes do not magically appear within the document, but exist solely without any reference to other elements and therefore must be separately appended to parent nodes. To add a new user to the document we need to create user object, first_name, last_name, member_of, several departments with id and name. We also should not forget to append all created elements to correponding parent nodes in a strict sequence. We must keep in mind that appending department to member_of cannot be done before id and name are appended to department itself, because adding an element into the document makes impossible its subsequent modification including insertion of child nodes. That seems complitely frustrating and unnatural. IXPL introduces an idea of travelling between the nodes inside a document and dynamically modifying existing elements or adding new ones. All elements are treated as integral part of a document, not a number of separated objects that have to be explicitly connected to each other. IXPL uses implicit binding instead by following programmer's thoughts as closely as possible and omitting unnecessary intermediate object manipulations. In IXPL, the insertion of a new user would look as follows:

```
Go to root. Add user. Add first_name "Ilja", last_name "Lutov", member_of to user. Go to user > member_of. Add department. Add id "I123", name "IT" to department. Return to user.
```

This small piece of code introduces some key concepts behind intentional XML processing. First of all it looks like ordinary language sentences (in fact, word "sentence" is userd in IXPL parser as an analogue to "statement") with appropriate grammar and punctuation. Secondly, the actuial connectivity between subsequent elements is almost removed from the syntax. When adding a user element, the programmer already knows that he is curently located at root element in the beginning of a document,

and IXPL parser also knows that. So there is no reason to specify the exact path where user element should be stored. The third and most important moment here is the implicit reference to the correct user node and all child elements that the programmer has in mind while working with current user. Since there may be several user elements within a document, the parser should not generate code that simultaneously creates and appends new user to the root node, beacause it would be impossible to locate the correct user in XML document due to the name ambiguity. The complexity encreases even further, since parser does not interprete IXPL source code on the fly, but transforms it into corresponding PHP code that can only be executed after the translation process.is finished. Therefore, new user element must be created without appending, and all subsequent elements should also remain as separate objects with unique variable names. When all elements are created, the program must implicitly connect them between each other and finally append to the document. Two questions arise. How can we detect that programmer has fullfilled the intention to add user element with all the children? At which point should program append all created nodes and assume that the programmer has something other in mind? The second question concerns technical implementation of the correct sequence in which newly created elements are appended to parent nodes.

The first problem can be solved by checking the path boundaries, in which the programmer operates while adding new nodes, and/or by disallowing some language constructs to be present during nodes creation. When the initial parent node is created, its location is stored in parser's internal variable and is used as an upper boundary. If current node path moves above that point, all created elements are appended. Analogously, if the current path exceeds the deepest child node which represents the lowest boundary, or path antirely goes out of the skope, the insertion of elements is considered to be finished. Also if forbidden sentence is encountered that is not needed for adding new nodes, it indicates the end of transaction.

The binding sequence problem can be handled by red-black tree. Since every node has a unique path, it is obvious to use it as a variable name for a corresponding element. Deeper nodes have longer paths and must be appended earlier in the sequence. Red-black tree is composed of nodes that are sorted according to a natural order algorithm. Tree node names could be used to represent paths to elements that are being created. Tree node values could contain destination language (PHP) code statements that append current element to its parent. The relations between parents and children are always visible, because IXPL parser knows the path of a newly created element. The entire sequence of bindings is then composed by traversing a tree upwards from the deepest element to the initial node and generating

corresponding code that appends elements to each other in a correct manner.

The key idea of IXPL is to relieve a programmer from thinking ahead and keeping in mind created elements along with relations to other nodes. Instead, a program code should directly reflect the intentions and immediate demands of a programmer. We want to add a new user element now, and have no desire to think about consequent child dependencies. When user is added, we can safely forget about this element and focus on adding first_name, last_name or any other node.

IXPL keywords

<OPEN>	opens a document
<ADD>	inserts new elements
<TO>	indicates a destination path or an element value
<ROOT>	defines root element path
<GO>	changes current path
<RETURN>	indicates a return point on a current path
<SET>	changes element value
<WITH>	indicates a conditional path that takes into consideration concrete element values
<REMOVE>	deletes an element
<FROM>	indicates a path of an element being deleted
<IDENTIFIER>	document or element name consisting of latin characters, digits and underscores
<LITERAL>	arbitrary string delimited by double quotes

IXPL constructs

This is a demo version if IXPL that is suitable only for demonstrational purposes as it lacks system integration capabilities. Programs cannot operate on external data or send back results.

<>	angle brackets denote a language keyword
[]	square brackets enclose an identifier or a string literal
()	round brackets define a terminal that consists of a single symbol
{}	curly brackets indicate a block
|	logical OR operator that allows the incorporation of alternative paths
?	used along with block delimeters to indicate the presence of zero or one block
*	used along with block delimeters to indicate the presence of zero or more blocks

Opening a document

```
<OPEN> [IDENTIFIER] (.)
```

Adding elements

This construct starts with ADD terminal followed by one or more element names (identifiers) separated by commas. Each element can have an optional value (string literal) associated with it. After elemets list goes TO terminal that indicates the consequent path where listed elements should be added. Path is composed of one or more nodes (identifiers) separated by ">" symbol. The first can be also represented by ROOT terminal. All nodes except ROOT can be have optional WITH terminal followed by one or more comma-separated elements (identifiers) with associated values (string literals).

```
<ADD> [IDENTIFIER]
{
        [LITERAL]
}?
{
        (,) [IDENTIFIER]
        {
                [LITERAL]
        }?
}*
<TO>
{
        <ROOT> |
        {
```

```
                [IDENTIFIER]
                {
                        <WITH> [IDENTIFIER] [LITERAL]
                        {
                                (,) [IDENTIFIER] [LITERAL]
                        }*
                }?
        }
}
{
        (>)  [IDENTIFIER]
        {
                <WITH> [IDENTIFIER] [LITERAL]
                {
                        (,) [IDENTIFIER] [LITERAL]
                }*
        }?
}*
{
        [LITERAL]
}?
(.)
```

Moving to a node with an arbitrary path

```
<GO> <TO>
{
        <ROOT> |
        {
                [IDENTIFIER]
                {
                        <WITH> [IDENTIFIER] [LITERAL]
                        {
                                (,) [IDENTIFIER] [LITERAL]
                        }*
                }?
        }
}
{
        (>)  [IDENTIFIER]
        {
                <WITH> [IDENTIFIER] [LITERAL]
                {
                        (,) [IDENTIFIER] [LITERAL]
                }*
        }?
}*
```

```
{
        [LITERAL]
}?
(.)
```

Moving back to a node on a current path

```
<RETURN> <TO>
{
        <ROOT> | [IDENTIFIER]
}
(.)
```

Setting an element value

```
<SET>
{
        {
                <ROOT> |
                {
                        [IDENTIFIER]
                        {
                                <WITH> [IDENTIFIER] [LITERAL]
                                {
                                        (,) [IDENTIFIER] [LITERAL]
                                }*
                        }?
                }
        }
        {
                (>)  [IDENTIFIER]
                {
                        <WITH> [IDENTIFIER] [LITERAL]
                        {
                                (,) [IDENTIFIER] [LITERAL]
                        }*
                }?
        }*
        {
                [LITERAL]
        }?
}?
<TO> [LITERAL] (.)
```

Deleting an element

```
<REMOVE> [IDENTIFIER]
{
        <WITH> [IDENTIFIER] [LITERAL]
        {
                (,) [IDENTIFIER] [LITERAL]
        }*
}?
{
        (>)  [IDENTIFIER]
        {
                <WITH> [IDENTIFIER] [LITERAL]
                {
                        (,) [IDENTIFIER] [LITERAL]
                }*
        }?
}*
{
        [LITERAL]
}?
(.)
```

Handling parse errors

All syntactic errors are automatically handled by parsers generated with JavaCC. Semantic errors on the other hand do not violate language syntax and therefor require custom error handling algorithms. IXPL parser recognizes three categories of semantic errors.

Fatal error

This kind of error occurs when IXPL code tries to perform an operation on a non-existent document. It means that some document must be loaded in the first sentence. Further analysis is impossible without an opened document. Parser is terminated.

Warning with automatic correction

When a programmer tries to add several identically named elements simultaneously in one sentence, parser displays a warning and assumes that the last occuring element is the correct one.

Simple warning

If element name in a RETURN TO sentence does not exist on the current path, parser displays a warning, but tekes no actions and leaves the current path intact.

IXPL parser source code

```
options {
	JDK_VERSION = "1.5";
}

PARSER_BEGIN(IXPLTranslator)

package translator;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.OutputStreamWriter;
import java.io.FileNotFoundException;
import java.io.UnsupportedEncodingException;
import java.io.IOException;
import java.io.BufferedWriter;
import java.util.Map;
import java.util.HashMap;
import java.util.TreeMap;
import java.util.Iterator;
import java.util.Collections;

public class IXPLTranslator {
	public static void main(String args[]) throws ParseException {
		try {
			output = new BufferedWriter(new OutputStreamWriter(new
FileOutputStream("test.php"), "UTF-8"));
			output.write("<?php" + lineSeparator);

			IXPLTranslator translator = new IXPLTranslator(new FileInputStream("test.ixpl"));
			translator.IXPL();

			output.write("?>" + lineSeparator);
			output.close();

			System.out.println(endingMessage);
		}
		catch (FileNotFoundException e) {
			System.out.println(e.getMessage());
		}
		catch (UnsupportedEncodingException e) {
			System.out.println(e.getMessage());
		}
		catch (IOException e) {
			System.out.println(e.getMessage());
		}
	}

	public static void saveCurrentDocument() {
		if (currentDocumentName != null) {
			generatedCode = "file_put_contents(\"" + currentDocumentName + ".xml\", $_document-
>saveXML());" + lineSeparator;
			generatedCode += "echo \"Document saved: " + currentDocumentName + "\\n\";" +
lineSeparator;

			try {
```

```
                        output.write(generatedCode);
                    }
                    catch (IOException e) {
                        System.out.println(e.getMessage());
                    }
                }
            }

            public static String composeFullElementName(String path, String name) {
                String fullName = path.replace("/", "_") + (name == null ? "" : "_" + name);

                return fullName.replaceAll("[^A-Za-z0-9_]", "_");
            }

            public static void commitTransaction() {
                Iterator iterator = transactionCode.entrySet().iterator();

                while (iterator.hasNext()) {
                    Map.Entry codeBlock = (Map.Entry) iterator.next();

                    try {
                        output.write(codeBlock.getValue().toString());
                    }
                    catch (IOException e) {
                        System.out.println(e.getMessage());
                    }
                }

                transactionCode.clear();
                transactionPath = null;
            }

            public static void recoverFromError(String message) {
                if (message == null) {
                    ParseException e = generateParseException();
                    message = e.getMessage();
                }

                System.out.println(message);
                endingMessage = "Failed.";

                Token token;

                do {
                    token = getNextToken();
                }
                while (!token.image.equals("."));
            }

            public static String getParentPath(String path) {
                int nodeIndex = path.lastIndexOf("/");

                return path.substring(0, nodeIndex);
            }

            public static String deriveElementNameFromPath(String path) {
                int nodeIndex = path.lastIndexOf("/") + 1;
```

```
                return path.substring(nodeIndex);
        }

        public static BufferedWriter output;
        public static String lineSeparator = System.getProperty("line.separator");
        public static String currentDocumentName = null;
        public static String generatedCode;
        public static String currentPath;
        public static String searchPath = null;
        public static String transactionPath = null;
        public static Map transactionCode = new TreeMap(Collections.reverseOrder());
        public static String endingMessage = "Succeeded.";
}

PARSER_END(IXPLTranslator)

SKIP: {" "}
SKIP: {"\t"}
SKIP: {"\r"}
SKIP: {"\n"}

TOKEN: {<OPEN: ("O" | "o") ("P" | "p") ("E" | "e") ("N" | "n")>}
TOKEN: {<ADD: ("A" | "a") ("D" | "d") ("D" | "d")>}
TOKEN: {<TO: ("T" | "t") ("O" | "o")>}
TOKEN: {<ROOT: ("R" | "r") ("O" | "o") ("O" | "o") ("T" | "t")>}
TOKEN: {<GO: ("G" | "g") ("O" | "o")>}
TOKEN: {<RETURN: ("R" | "r") ("E" | "e") ("T" | "t") ("U" | "u") ("R" | "r") ("N" | "n")>}
TOKEN: {<SET: ("S"|"s") ("E"|"e") ("T"|"t")>}
TOKEN: {<WITH: ("W" | "w") ("I" | "i") ("T" | "t") ("H" | "h")>}
TOKEN: {<REMOVE: ("R" | "r") ("E" | "e") ("M" | "m") ("O" | "o") ("V" | "v") ("E" | "e")>}
TOKEN: {<FROM: ("F" | "f") ("R" | "r") ("O" | "o") ("M" | "m")>}
TOKEN: {<IDENTIFIER: ["A" - "Z", "a" - "z"] (["A" - "Z", "a" - "z", "0" - "9", "_"])*>}
TOKEN: {<LITERAL: "\"" (~["\"", "\t", "\r", "\n", "/"])* "\"">}

void IXPL() :
{
        Token token;
}
{
        (
                try {
                        Sentence()
                }
                catch (ParseException e) {
                        recoverFromError(null);
                }
        )+

        <EOF>

        {
                commitTransaction();
                saveCurrentDocument();
        }
}
```

```
void Sentence() throws ParseException :
{
}
{
	(
		OpenDocument() |
		{
			if (currentDocumentName == null) {
				recoverFromError("Cannot perform operation: document is not loaded");

				return;
			}
		}

		AddElements() |
		GoTo() |
		ReturnTo() |
		SetValue() |
		RemoveElement()
	)
	"."
}

void OpenDocument() :
{
	String documentName;
}
{
	<OPEN>
	documentName = Identifier()
	{
		{
			saveCurrentDocument();
		}

		generatedCode = "$_file = '<?xml version=\"1.0\" encoding=\"utf-8\"?><" + documentName +
"></" + documentName + ">';" + lineSeparator;
		generatedCode += "if (file_exists(\"" + documentName + ".xml\")) {" + lineSeparator + "$_file =
file_get_contents(\"" + documentName + ".xml\");" + lineSeparator + "}" + lineSeparator;
		generatedCode += "$_document = new DOMDocument();" + lineSeparator;
		generatedCode += "if (!$_document->loadXML($_file)) {" + lineSeparator + "echo \"Failed to load
document: " + documentName + "\\n\";" + lineSeparator + "exit;" + lineSeparator + "}" + lineSeparator;
		generatedCode += "echo \"Document loaded: " + documentName + "\\n\";" + lineSeparator;
		generatedCode += "$_xpath = new DOMXPath($_document);" + lineSeparator;

		try {
			output.write(generatedCode);
		}
		catch (IOException e) {
			System.out.println(e.getMessage());
		}

		currentDocumentName = documentName;
		currentPath = "/" + documentName;
	}
}
```

```
String Identifier() :
{
	Token identifierToken;
}
{
	identifierToken = <IDENTIFIER>
	{
		return identifierToken.image;
	}
}

String Literal() :
{
	Token literalToken;
	String literalValue;
}
{
	literalToken = <LITERAL>
	{
		literalValue = literalToken.image;

		return literalValue.replace("\"", "");
	}
}

void AddElements() :
{
	Map elements;
	String path = currentPath;
	String parentFullName;
	String childFullName;
}
{
	<ADD> elements = Elements() (<TO> path = Path())?
	{
		if (transactionPath == null) {
			transactionPath = path;
		}

		parentFullName = composeFullElementName(path, null);

		generatedCode = "";

		if (!transactionCode.containsKey(parentFullName)) {
			generatedCode += "$_entries = $_xpath->query(\"" + path + "\");" + lineSeparator;
			generatedCode += "if ($_entries->length == 0) {" + lineSeparator + "echo \"Node does
not exist: " + path + "\\n\";" + lineSeparator + "}" + lineSeparator;
			generatedCode += "else {" + lineSeparator;
			generatedCode += "$" + parentFullName + " = $_entries->item(0);" + lineSeparator;

			transactionCode.put(parentFullName + "_", "}" + lineSeparator);
		}

		Iterator iterator = elements.entrySet().iterator();

		while (iterator.hasNext()) {
			Map.Entry element = (Map.Entry) iterator.next();
```

```
                childFullName = composeFullElementName(path, element.getKey().toString());

                if (element.getValue() == null) {
                        generatedCode += "$" + childFullName + " = $_document->createElement('" +
element.getKey() + "');" + lineSeparator;
                }
                else {
                        generatedCode += "$" + childFullName + " = $_document->createElement('" +
element.getKey() + "', '" + element.getValue() + "');" + lineSeparator;
                }

                generatedCode += "echo \"Element added: " + path + "/" + element.getKey() + "\\n\";" +
lineSeparator;

                transactionCode.put(childFullName, "$" + parentFullName + "->appendChild($" +
childFullName + ");" + lineSeparator);
        }

        try {
                output.write(generatedCode);
        }
        catch (IOException e) {
                System.out.println(e.getMessage());
        }
    }
}

Map Elements() :
{
    Token elementNameToken;
    String elementValue;
    Map elements = new HashMap();
}
{
    elementNameToken = <IDENTIFIER>
    {
        elementValue = null;
    }

    (elementValue = Literal())?
    {
        if (elements.containsKey(elementNameToken.image)) {
                System.out.println("Duplicate element replaces previous one in the list: " +
elementNameToken.image + " (line " + elementNameToken.beginLine + ", column " +
elementNameToken.beginColumn + ")");
        }

        elements.put(elementNameToken.image, elementValue);
    }

    (
        "," elementNameToken = <IDENTIFIER>
        {
                elementValue = null;
        }
```

```
		(elementValue = Literal())?
		{
			elements.put(elementNameToken.image, elementValue);
		}
	)*

	{
		return elements;
	}
}

String Path() :
{
	Token nodeToken;
	String path = currentPath;
	String condition;
	String value;
}
{
	(
		(
			<ROOT>
			{
				path = "/" + currentDocumentName;
			}
		)

		|

		(
			nodeToken = <IDENTIFIER>
			{
				path += "/" + nodeToken.image;
			}

			(
				condition = Condition()
				{
					path += condition;
				}
			)?
		)
	)

	(
		">" nodeToken = <IDENTIFIER>
		{
			path += "/" + nodeToken.image;
		}

		(
			condition = Condition()
			{
				path += condition;
			}
		)?
	)*
```

```
	(
		value = Literal()
		{
			path += "[.='" + value + "']";
		}
	)?

	{
		if (transactionPath != null && !path.startsWith(transactionPath)) {
			commitTransaction();
		}

		return path;
	}
}

String Condition() :
{
	Token elementNameToken;
	String elementValue;
	String condition;
}
{
	<WITH> elementNameToken = <IDENTIFIER> elementValue = Literal()
	{
		condition = "[" + elementNameToken.image + "='" + elementValue + "'";
	}

	(
		"," elementNameToken = <IDENTIFIER> elementValue = Literal()
		{
			condition += " and " + elementNameToken.image + "='" + elementValue + "'";
		}
	)*

	{
		condition += "]";

		return condition;
	}
}

void GoTo() :
{
	String path;
}
{
	<GO> <TO> path = Path()
	{
		currentPath = path;

		try {
			output.write("echo \"Went to: " + currentPath + "\\n\";" + lineSeparator);
		}
		catch (IOException e) {
			System.out.println(e.getMessage());
```

```
		}
	}
}

void ReturnTo() :
{
	Token nodeToken;
	String nodeName;
	String[] nodes;
	int node;
	int i;
}
{
	<RETURN> <TO>
	(
		nodeToken = <ROOT>
		{
			nodeName = currentDocumentName;
		}

		|

		nodeToken = <IDENTIFIER>
		{
			nodeName = nodeToken.image;
		}
	)

	{
		nodes = currentPath.split("/");

		for (node = nodes.length - 2; node >= 0; node--) {
			if (nodes[node].equals(nodeName) || nodes[node].startsWith(nodeName + "[")) {
				break;
			}
		}

		if (node > 0) {
			currentPath = "";

			for (i = 1; i <= node; i++) {
				currentPath += "/" + nodes[i];
			}

			try {
				output.write("echo \"Returned to: " + currentPath + "\\n\";" + lineSeparator);
			}
			catch (IOException e) {
				System.out.println(e.getMessage());
			}
		}
		else {
			System.out.println("Node does not exist on the current path: " + nodeName + " (line " +
nodeToken.beginLine + ", column " + nodeToken.beginColumn + ")");
		}
	}
}
```

```
void SetValue() :
{
	String childPath = currentPath;
	String parentPath;
	String childFullName;
	String parentFullName;
	String value;
}
{
	<SET> (childPath = Path())? <TO> value = Literal()
	{
		commitTransaction();

		parentPath = getParentPath(childPath);
		parentFullName = composeFullElementName(parentPath, null);
		childFullName = composeFullElementName(childPath, null);

		generatedCode = "$_entries = $_xpath->query(\"" + childPath + "\");" + lineSeparator;
		generatedCode += "if ($_entries->length == 0) {" + lineSeparator + "echo \"Node does not exist: "
+ childPath + "\\n\";" + lineSeparator + "}" + lineSeparator;
		generatedCode += "else {" + lineSeparator;
		generatedCode += "$" + childFullName + "_old = $_entries->item(0);" + lineSeparator;
		generatedCode += "$" + childFullName + "_new = $_document->createElement('" +
deriveElementNameFromPath(childPath) + "', '" + value + "');" + lineSeparator;
		generatedCode += "$" + parentFullName + " = $_xpath->query(\"" + parentPath + "\")->item(0);"
+ lineSeparator;
		generatedCode += "$" + parentFullName + "->replaceChild($" + childFullName + "_new, $" +
childFullName + "_old);" + lineSeparator;
		generatedCode += "echo \"Value set: " + childPath + "\\n\";" + lineSeparator;
		generatedCode += "}" + lineSeparator;

		try {
			output.write(generatedCode);
		}
		catch (IOException e) {
			System.out.println(e.getMessage());
		}
	}
}


void RemoveElement() :
{
	String path;
	String element;
	String[] elementNodes;
	int node;
}
{
	<REMOVE> element = RelativePath() <FROM> path = Path()
	{
		commitTransaction();

		generatedCode = "$_entries = $_xpath->query(\"" + path + element + "\");" + lineSeparator;
		generatedCode += "if ($_entries->length == 0) {" + lineSeparator + "echo \"Node does not exist: "
+ path + element + "\\n\";" + lineSeparator + "}" + lineSeparator;
```

```
                generatedCode += "else {" + lineSeparator;
                generatedCode += "$_element = $_entries->item(0);" + lineSeparator;

                elementNodes = element.split("/");

                for (node = elementNodes.length - 1; node > 1; node--) {
                        generatedCode += "$_element = $_element->parentNode;" + lineSeparator;
                }

                generatedCode += "$_node = $_element->parentNode;" + lineSeparator;
                generatedCode += "$_node->removeChild($_element);" + lineSeparator;
                generatedCode += "echo \"Element removed: " + path + "/" + elementNodes[1] + "\\n\";" +
lineSeparator;
                generatedCode += "}"  + lineSeparator;

                try {
                        output.write(generatedCode);
                }
                catch (IOException e) {
                        System.out.println(e.getMessage());
                }
        }
}

String RelativePath() :
{
        Token nodeToken;
        String path = "";
        String condition;
        String value;
}
{
        (
                nodeToken = <IDENTIFIER>
                {
                        path += "/" + nodeToken.image;
                }

                (
                        condition = Condition()
                        {
                                path += condition;
                        }
                )?
        )

        (
                ">" nodeToken = <IDENTIFIER>
                {
                        path += "/" + nodeToken.image;
                }

                (
                        condition = Condition()
                        {
                                path += condition;
                        }
```

```
		)?
	)*

	(
		value = Literal()
		{
			path += "[.='" + value + "']";
		}
	)?

	{
		return path;
	}
}
```