

УТВЕРЖДЕНО

Заведующий кафедрой «Управление  
разработкой программного обеспечения»  
\_\_\_\_\_ / Авдошин С.М./  
« \_\_\_\_ » \_\_\_\_\_ 2012 г.

**КОМПОНЕНТНАЯ МОДЕЛЬ С ДЕКЛАРАТИВНЫМ ОПИСАНИЕМ  
СОСТАВНЫХ ТИПОВ: ПАРСЕРЫ**

Текст программы

**ЛИСТ УТВЕРЖДЕНИЯ**

Инв. № подп.	Подп. и дата	Инв. № дубл.	Подп. и дата

Руководитель работы

\_\_\_\_\_ / Гринкруг Е.М./  
« \_\_\_\_ » \_\_\_\_\_ 2012 г.

Исполнитель: студент группы 271ПИ

\_\_\_\_\_ / Дубов М.С. /  
« \_\_\_\_ » \_\_\_\_\_ 2012 г.

Национальный исследовательский университет – Высшая школа экономики  
Факультет бизнес-информатики, отделение программной инженерии

УТВЕРЖДЕНО

**КОМПОНЕНТНАЯ МОДЕЛЬ С ДЕКЛАРАТИВНЫМ ОПИСАНИЕМ  
СОСТАВНЫХ ТИПОВ: ПАРСЕРЫ**

Текст программы

Инв. № подп.	Подп. и дата	Взам. инв. №	Инв. № дубл.	Подп. и дата

Листов 49

## Содержание

Содержание .....	2
1. Пакет ru.hse.se.parsers .....	3
1.1. <i>Parser.java</i> .....	3
1.2. <i>VRMLParser.java</i> .....	7
1.3. <i>X3DParser.java</i> .....	18
2. Пакет ru.hse.se.parsers.errors .....	25
2.1. <i>ParsingError.java</i> .....	25
2.2. <i>LexicalError.java</i> .....	26
2.3. <i>SyntaxError.java</i> .....	27
2.4. <i>TypeMismatchError.java</i> .....	27
2.5. <i>Warning.java</i> .....	27
3. Пакет ru.hse.se.codegenerators .....	28
3.1. <i>CodeGenerator.java</i> .....	28
3.2. <i>VRMLCodeGenerator.java</i> .....	29
3.3. <i>X3DCodeGenerator.java</i> .....	31
4. Пакет ru.hse.se.nodes .....	33
4.1. <i>Appearance.java</i> .....	33
4.2. <i>Box.java</i> .....	33
4.3. <i>Geometry.java</i> .....	33
4.4. <i>Group.java</i> .....	34
4.5. <i>Material.java</i> .....	34
4.6. <i>Node.java</i> .....	35
4.7. <i>Shape.java</i> .....	35
4.8. <i>Sphere.java</i> .....	36
4.9. <i>Text.java</i> .....	36
5. Пакет ru.hse.se.types .....	37
5.1. <i>MFBool.java</i> .....	37
5.2. <i>MFFloat.java</i> .....	37
5.3. <i>MFInt32.java</i> .....	38
5.4. <i>MFNode.java</i> .....	38
5.5. <i>MFString.java</i> .....	39
5.6. <i>MFType.java</i> .....	40
5.7. <i>MFValueType.java</i> .....	41
5.8. <i>SFBool.java</i> .....	43
5.9. <i>SFColor.java</i> .....	44
5.10. <i>SFFloat.java</i> .....	45
5.11. <i>SFInt32.java</i> .....	46
5.12. <i>SFString.java</i> .....	48
5.13. <i>ValueType.java</i> .....	49
5.14. <i>VRMLType.java</i> .....	49

# 1. Пакет ru.hse.se.parsers

## 1.1. Parser.java

```
package ru.hse.se.parsers;

import ru.hse.se.nodes.Node;
import ru.hse.se.parsers.errors.SyntaxError;
import ru.hse.se.types.MFBool;
import ru.hse.se.types.MFFloat;
import ru.hse.se.types.MFInt32;
import ru.hse.se.types.MFString;
import ru.hse.se.types.SFBool;
import ru.hse.se.types.SFFloat;
import ru.hse.se.types.SFInt32;
import ru.hse.se.types.SFString;
import ru.hse.se.types.ValueType;

import java.io.*;
import java.util.ArrayList;

/**
 * An abstract parser class that defines
 * the basic items each parser should
 * have to analyze a scene.
 *
 * @author Mikhail Dubov
 */
public abstract class Parser {

    /**
     * Parses the input file and builds
     * an array of root nodes.
     *
     * @param reader The input stream reader
     * @return nodes array of root nodes
     * @throws IOException if there is no input in the stream
     */
    public ArrayList<Node> parse(InputStreamReader reader)
        throws IOException {

        tokenizer = new StreamTokenizer(new BufferedReader(reader));

        // TODO: Encoding issue??? *.wrl UTF8 -> bad; ANSI -> ok

        setUpTokenizer();

        init();

        sceneGraph = new ArrayList<Node>();
        parsingErrors = new ArrayList<Error>();

        // !!! The entry point !!!
        parseScene();
        // As a result - the filled sceneGraph array,
        // that may be null if there were parsing errors.

        if (! parsingErrors.isEmpty()) {
            return null;
        } else {
            return sceneGraph;
        }
    }

    /**
     * Sets up the tokenizer object.
     */
    protected void setUpTokenizer() {

        tokenizer.resetSyntax();

        tokenizer.wordChars('a', 'z'); // Id's
        tokenizer.wordChars('A', 'Z'); // Id's
    }
}
```

```

tokenizer.wordChars('0', '9'); // Id's
tokenizer.wordChars('_', '_'); // Id's can contain '_'
tokenizer.wordChars('+', '+'); // For floats and ints
tokenizer.wordChars('-', '-'); // For floats and ints
tokenizer.wordChars('.', '.'); // For floats

tokenizer.quoteChar('"');

tokenizer.whitespaceChars(' ', ' ');
tokenizer.whitespaceChars('\n', '\n');
tokenizer.whitespaceChars('\t', '\t');
tokenizer.whitespaceChars('\r', '\r');
tokenizer.whitespaceChars(',', ','); // => for [ children ]
}

/**
 * Initializes the parser by reading the first
 * token and storing it in the lookahead variable.
 */
protected abstract void init() throws IOException;

/**
 * Performs the parsing of the input file
 * and fills out the ArrayList of root nodes,
 * namely the sceneGraph array.
 */
protected abstract void parseScene() throws IOException;

/**
 * Parses the next Node from the input stream.
 * Needed for MFNode parsing.
 */
public abstract Node parseChildNode();

/**
 * Parses a value of particular type from the input stream.
 *
 * @param currentFieldType the type of the value
 * @return the value read from the stream
 * @throws Error
 */
protected Object parseValueType(Class<?> currentFieldType) throws Error {

    Object value = null;
    // TODO: No Syntax Error messages from type classes?..

    /***** a) Value type => call "parse" method in the type class via reflection *****/
    if (ValueType.class.isAssignableFrom(currentFieldType)) {
        try {
            value = currentFieldType.getDeclaredMethod(
                "parse", Parser.class).invoke(null, this);
        } catch (Exception e) { }

        if (value == null) {
            throw new Error("Parse rules for type " +
                currentFieldType.getName() + " not defined.");
        }
    }

    /***** b) Java primitive types => use VRML wrappers (SFBool, SFFloat, ...) *****/
    // TODO: TEST!
    else if (currentFieldType == int.class) {
        value = SFInt32.parse(this).getValue();
    } else if (currentFieldType == int[].class) {
        ArrayList<SFInt32> val = MFInt32.parse(this).getValue();
        value = new int[val.size()];
        for (int i = 0; i < val.size(); i++) {
            ((int[])value)[i] = val.get(i).getValue();
        }
    } else if (currentFieldType == boolean.class) {
        value = SFBool.parse(this).getValue();
    } else if (currentFieldType == boolean[].class) {
        ArrayList<SFBool> val = MFBool.parse(this).getValue();

```

```

        value = new boolean[val.size()];
        for (int i = 0; i < val.size(); i++) {
            ((boolean[])value)[i] = val.get(i).getValue();
        }

    } else if (currentFieldType == double.class) {
        value = SFFloat.parse(this).getValue();

    } else if (currentFieldType == double[].class) {
        ArrayList<SFFloat> val = MFFloat.parse(this).getValue();
        value = new double[val.size()];
        for (int i = 0; i < val.size(); i++) {
            ((double[])value)[i] = val.get(i).getValue();
        }

    } else if (currentFieldType == float.class) {
        value = (float)(SFFloat.parse(this).getValue());

    } else if (currentFieldType == float[].class) {
        ArrayList<SFFloat> val = MFFloat.parse(this).getValue();
        value = new double[val.size()];
        for (int i = 0; i < val.size(); i++) {
            ((float[])value)[i] = (float)(val.get(i).getValue());
        }

    } else if (currentFieldType == String.class) {
        value = (String)(SFString.parse(this).getValue());

    } else if (currentFieldType == String[].class) {
        ArrayList<SFString> val = MFString.parse(this).getValue();
        value = new String[val.size()];
        for (int i = 0; i < val.size(); i++) {
            ((String[])value)[i] = (String)(val.get(i).getValue());
        }

    }

    //else if (currentFieldType == ArrayList.class) {
    // TODO: Process ArrayLists?
    //}

    /***** c) Error otherwise *****/
    else {
        registerError(new Error("Value of unknown type"));
    }

    return value;
}

/**
 * Returns the tokenizer.
 */
public StreamTokenizer tokenizer() {
    return tokenizer;
}

/**
 * Determines whether the lookahead token
 * coincides with the given one.
 */
public boolean lookahead(String token) {
    return (lookahead() != null && token != null && lookahead().equals(token));
}

/**
 * Returns the lookahead token.
 */
public abstract String lookahead();

/**
 * Reads the next token from the input.
 * @returns false when the next token is unavailable, true otherwise
 */
public abstract boolean nextToken();

/**

```

```

* Compares the token with the lookahead symbol and
* advances to the next input terminal if they match,
* registers a syntax error otherwise
*
* Note: The method is case sensitive.
*
* @param token Token to be matched
* @return true
*/
public boolean match(String token) {
    if(lookahead().equals(token)) {
        nextToken();
    } else {
        registerError(new SyntaxError("Expected '" + token + "', but got '" +
            lookahead() + "'",
            tokenizer.lineno()));
    }
    return true;
}

/**
 * Determines whether the lookahead token
 * is equal to the one passed as a parameter
 * and matches it if the result is true.
 *
 * @param token the token
 * @return true if the matching was successful, false otherwise
 */
public boolean tryMatch(String token) {
    if (lookahead(token)) {
        match(token);
        return true;
    } else {
        possibleError = new SyntaxError("Expected '" + token +
            "'", but got '" + lookahead() + "'", tokenizer.lineno());
        return false;
    }
}

/**
 * Processes a parsing error.
 *
 * @param e the error object
 */
public boolean registerError(Error e) {
    if (e != null) {
        parsingErrors.add(e);
    }
    return true;
}

/**
 * Returns the list of parsing errors.
 *
 * @return the ArrayList of parsing error objects
 */
public ArrayList<Error> getParsingErrors() {
    return parsingErrors;
}

/**
 * Determines whether the node with a given
 * name exists in one of registered node packages,
 * and returns the appropriate Class<?> object
 * if there is such a node type.
 *
 * @param str Node name (simple name)
 * @return the appropriate Class if the node exists,
 *         null otherwise

```

```

    */
    protected Class<?> classForNodeName(String str) {
        Class<?> res = null;
        for (String pkg : nodePackages) {
            try {
                res = Class.forName(pkg + "." + str);
                // here => Class found
                break;
            } catch (ClassNotFoundException e) {}
        }
        return res;
    }
}

/**
 * Creates instance of a node for its name.
 *
 * @param str Node name (simple name)
 * @return the node object (if this node type exists)
 * @throws Exception if there are instantiation errors
 */
protected Node createInstance(String str) throws Exception {
    for (String pkg : nodePackages) {
        try {
            Node res = (Node)(Class.forName(pkg + "." + str).newInstance());
            // here => Class found
            return res;
        } catch (ClassNotFoundException e) {}
    }
    throw new Exception();
}

/**
 * Registers new package that contains nodes
 * that can appear in the input file.
 * Registering is needed in order for the parser
 * to be able to check for errors during the parsing.
 *
 * @param packageName packageName
 */
public void registerNodePackage(String packageName) {
    nodePackages.add(packageName);
}

/** Tokenizer */
protected StreamTokenizer tokenizer;

/** The result of scene parsing */
protected ArrayList<Node> sceneGraph;

/** The errors that occurred during parsing */
protected ArrayList<Error> parsingErrors;

/**
 * The error that occurred during the last call
 * of some tryXxx method; such an error is not
 * registered until registerPossibleError() is called.
 */
protected Error possibleError = null;

/** The nodes package name (needed for reflection) */
protected static final ArrayList<String> nodePackages;

static {
    nodePackages = new ArrayList<String>();
    nodePackages.add("ru.hse.se.nodes");
}
}

```

## 1.2. VRMLParser.java

```
package ru.hse.se.parsers;
```



```

import ru.hse.se.nodes.*;
import ru.hse.se.parsers.errors.*;
import ru.hse.se.types.MFNode;

import java.io.IOException;
import java.io.StreamTokenizer;
import java.lang.reflect.Method;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Stack;

/**
 * VRML parser. Builds up a bunch of beans
 * on the basis of its declarative description.
 *
 * @author Mikhail Dubov
 */
public class VRMLParser extends Parser {

    /**
     * Sets up the tokenizer object
     * according to the VRML grammar.
     * (defines terminals etc.)
     */
    @Override
    protected void setUpTokenizer() {

        super.setUpTokenizer();

        tokenizer.commentChar('#');

        // Terminals
        tokenizer.ordinaryChar('{');
        tokenizer.ordinaryChar('}');
        tokenizer.ordinaryChar('[');
        tokenizer.ordinaryChar(']');

        //tokenizer.parseNumbers(); // => No! Bad for advanced float/int32 parsing
        tokenizer.lowerCaseMode(false); // VRML is not case-sensitive
        tokenizer.eolIsSignificant(false); // We can count lines with tokenizer.lineno()
    }

    /*****
     * Recursive-descent predictive top-down VRML parser.
     *
     * See 2.4 in the "Dragon book" for technique description
     * and http://bit.ly/wF541A for VRML grammar.
     *****/

    /*****
     * vrmlScene ::=
     * statements
     *****/
    @Override
    protected void parseScene() throws IOException {

        parseStatements();

        // Some token left unparsed
        if (lookahead != null) {
            registerError(new SyntaxError("Unrecognized lexeme sequence starting at '"
                + lookahead + "'", tokenizer.lineno()));
        }
    }

    /*****
     * The basic grammar.
     *****/

    /*****
     * statements ::=
     * statement |
     * statement statements |
     * empty
     *****/

```

```

*****/
private boolean parseStatements() {
    while (parseStatement());
    return true;
}

/*****
 * statement ::=
 *     nodeStatement |
 *     protoStatement |
 *     routeStatement
 *****/
private boolean parseStatement() {
    // ! NB: The order is essential, because of the FIRST elements
    return (lookahead != null) &&
        ((parseProtoStatement()) ||
         (parseRouteStatement()) ||
         (parseNodeStatement() && addRootNode()));
}

/**
 * Parses a node and stores the link
 * to that node on the top of the
 * 'currentNodes' stack.
 */
/*****
 * nodeStatement ::=
 *     node |
 *     DEF nodeNameId node |
 *     USE nodeNameId
 *
 * FIRST = nodeId | DEF | USE
 *****/
private boolean parseNodeStatement() {
    // ! NB: The order is essential, because of the FIRST elements
    // In the "DEF" production, parseNode() will store
    // the node in the hash table by its id.
    return (tryMatch("DEF") && matchId() && parseNode()) ||
        (tryMatch("USE") && matchId() && instantiateNodeById()) ||
        (parseNode());
}

/**
 * Parses the next Node from the input stream.
 * Needed for MFNode parsing.
 */
@Override
public Node parseChildNode() {
    if (parseNodeStatement()) {
        return currentNodes.pop();
    } else {
        return null;
    }
}

/*****
 * node ::=
 *     nodeId { nodeBody } |
 *     Script { scriptBody }
 *
 * FIRST = Script | empty
 *****/
private boolean parseNode() {

```

```

// ! NB: The order is essential, because of the FIRST elements

return (tryMatch("Script") &&
        match("{") && parseScriptBody() && match("}")) ||

        (tryMatchTypeId() && instantiateNode() &&
         match("{") && parseNodeBody() && match("}")) ||

        // Handling a typical syntax error case, trying to recover
        // '}' is the only correct lexeme at this point
        (! lookahead("}") &&
         registerError(possibleError) && panicModeRecovery() &&
         (currentNodes.push(null) == null)); // pushing fake node
}

/*****
 * nodeBody ::=
 *   nodeBodyElement |
 *   nodeBodyElement nodeBody |
 *   empty
 *****/
private boolean parseNodeBody() {

    while (parseNodeBodyElement());

    return true;
}

/*****
 * nodeBodyElement ::=
 *   fieldId fieldValue |
 *   fieldId IS fieldId |
 *   eventInId IS eventInId |
 *   eventOutId IS eventOutId |
 *   routeStatement |
 *   protoStatement
 *****/
private boolean parseNodeBodyElement() {

    // Trying to parse routeStatement or protoStatement at first
    // (Since they contain terminals at their FIRST position).

    return (parseRouteStatement()) ||

           (parseProtoStatement()) ||

           (tryMatchFieldId() &&

            ((tryMatch("IS") && matchId()
              /* && ??? -> 3 productions!!! */) ||

             (matchFieldValueAndSetField())))) ||

           // Handling a typical syntax error case, trying to recover
           // '}' is the only correct lexeme at this point
           (! lookahead("}") &&
            registerError(possibleError) && panicModeRecovery());
}

/*****
 *          PROTO & ROUTE statements.
 *****/

/*****
 * protoStatement ::=
 *   proto |
 *   externproto
 *
 * FIRST = PROTO | EXTERNPROTO
 *****/
private boolean parseProtoStatement() {

    return (lookahead("PROTO") && parseProto()) ||

```

```

        (lookahead("EXTERNPROTO") /* && ...ToDo...*/);
    }

    /*****
    * proto ::=
    *     PROTO nodeId
    *     [ interfaceDeclarations ]
    *     { protoBody } ;
    *****/
    private boolean parseProto() {

        return false; //(match("PROTO") && matchTypeId() && instantiateProtoNode() &&
            // match("[") && parseProtoInterface() && match("]") &&
            // match("{") && parseProtoBody() && match("}"));
    }

    /*****
    * routeStatement ::=
    *     ROUTE nodeNameId . eventOutId
    *     TO nodeNameId . eventInId
    *
    * FIRST = ROUTE
    *****/
    private boolean parseRouteStatement() {

        return (tryMatch("ROUTE") /* && ...ToDo...*/);
    }

    /*****
    * scriptBody ::=
    *     scriptBodyElement |
    *     scriptBodyElement scriptBody |
    *     empty
    *****/
    private boolean parseScriptBody() {

        // ToDo

        return false;
    }

    /*****
    * Token operations.
    *****/

    /**
    * Initializes the parser by reading the first
    * token and storing it in the lookahead variable.
    */
    protected void init() throws IOException {

        initFields();

        nextToken();

        if (lookahead == null) {
            throw new IOException();
        }
    }

    /**
    * Returns the lookahead token.
    */
    @Override
    public String lookahead() {
        return (lookahead == null ? "" : lookahead);
    }

    /**
    * Checks a token for being a valid Id.
    * nodeNameId, nodeId, fieldId, eventInId, eventOutId
    * are all id's.

```

```

*
*****
* Id ::=
*   IdFirstChar |
*   IdFirstChar IdRestChars
* IdFirstChar ::=
*   Any ISO-10646 character encoded using UTF-8 except:
*   0x30-0x39, 0x0-0x20, 0x22, 0x23, 0x27, 0x2b, 0x2c, 0x2d,
*   0x2e, 0x5b, 0x5c, 0x5d, 0x7b, 0x7d, 0x7f
* IdRestChars ::=
*   Any number of ISO-10646 characters except:
*   0x0-0x20, 0x22, 0x23, 0x27, 0x2c, 0x2e, 0x5b,
*   0x5c, 0x5d, 0x7b, 0x7d, 0x7f
*****
* @param id
* @return true if the token is a correct id, false otherwise
*/
private boolean lookaheadIsId() {

    // TODO: More Id checking (see rules)

    return (lookahead != null) && (lookahead != "") &&
        (Character.isLetter(lookahead.charAt(0)) || lookahead.charAt(0) == '_');
}

/**
 * Reads the next token that represents an Id.
 *
 * @return true
 */
private boolean matchId() {

    if(lookaheadIsId()) {

        currentId = lookahead;
        nextToken();

    } else {

        registerError(new SyntaxError("'" + lookahead + "' is not a valid id",
            tokenizer.lineno()));

    }

    return true;
}

/**
 * Determines whether the current token is
 * a field name of the current node.
 * @return true, if the current token is a field, false otherwise
 */
private boolean lookaheadIsFieldName() {

    if (currentNodes.isEmpty()) {
        return false;
    }

    boolean isFieldName = false;

    Method[] methods = currentNodes.peek().getClass().getDeclaredMethods();
    for (Method m : methods) {
        if (m.getName().startsWith("get")) {
            String field = Character.toLowerCase(m.getName().charAt(3)) +
                m.getName().substring(4);

            if (field.equals(lookahead)) {
                isFieldName = true;
                break;
            }
        }
    }

    return isFieldName;
}

/**
 * Returns the hash set of the current node fields,

```

```

* used for lexical error diagnostics.
*
* @return HashSet of the current node fields
*/
private HashSet<String> getCurrentNodeFields() {

    if (currentNodes.isEmpty()) {
        return null;
    }

    HashSet<String> res = new HashSet<String>();

    Method[] methods = currentNodes.peek().getClass().getDeclaredMethods();
    for (Method m : methods) {
        if (m.getName().startsWith("get")) {
            String field = Character.toLowerCase(m.getName().charAt(3)) +
                           m.getName().substring(4);
            res.add(field);
        }
    }

    return res;
}

/**
 * Reads the next token that represents a fieldId
 * (which is also an Id), and pushes it into the stack.
 */
private boolean tryMatchFieldId() {

    if(lookaheadIsFieldName()) {

        currentField.push(lookahead);
        nextToken();

        return true;

    } else {

        // There is no field with the given name
        possibleError = new LexicalError("'" + lookahead +
                                           "' is not a valid field name",
                                           tokenizer.lineno(), lookahead,
                                           getCurrentNodeFields());

        return false;

    }
}

/**
 * Reads the next token that represents a type
 * (which is also an Id).
 * @return true, if lookahead is a valid node name, false otherwise
 */
private boolean tryMatchTypeId() {

    Class<?> nodeType = classForNodeName(lookahead);
    if (nodeType != null) {

        currentType = lookahead;
        nextToken();

        // There is a node with the given name,
        // but it should be checked for type matching
        try {
            Class<?> fieldType = currentNodes.peek().getClass().
                getDeclaredMethod("get" + Character.toUpperCase(
                    currentField.peek().charAt(0)) +
                    currentField.peek().substring(1)).getReturnType();

            if (! fieldType.isAssignableFrom(nodeType) &&
                ! fieldType.isAssignableFrom(MFNode.class)) {
                possibleError = new TypeMismatchError
                    (nodeType, fieldType, tokenizer.lineno());
                currentId = null; // to preserve invalid DEF assignments

                return false;
            }
        }
    }
}

```

```

    }
    } catch (Exception e) { }

    return true;
} else {

    // There is no node with the given name

    possibleError = new LexicalError("'" + lookahead +
                                     "' is not a valid node name",
                                     tokenizer.lineno(), lookahead, null);
    currentId = null; // to preserve invalid DEF assignments

    return false;
}
}

/**
 * Reads the next token from the input.
 * @return false when the next token is unavailable, true otherwise
 */
@Override
public boolean nextToken() {

    try {
        int ttype = tokenizer.nextToken();

        if (ttype == '{' || ttype == '}' ||
            ttype == '[' || ttype == ']') {
            // Terminals
            lookahead = String.valueOf(((char)tokenizer.ttype));
        } else if (ttype == StreamTokenizer.TT_WORD) {
            // Non-terminals
            lookahead = tokenizer.sval;
        } else if (ttype == '"') {
            // Quoted Strings
            lookahead = tokenizer.sval;
        } else if (ttype == StreamTokenizer.TT_EOF) {
            // End of file
            lookahead = null; // to indicate EOF
            return false;
        } // No TT_NUMBER or TT_EOL can arise
    } catch (IOException e) {
        return false;
    }

    return true;
}

/*****
 * Error recovery.
 *****/

/**
 * Tries to recover from errors in order
 * for the parser to be able to continue
 * reading the input stream.
 *
 * See 4.1 in the "Dragon book" for technique description.
 *
 * @return true, if recovery proceeded, false otherwise
 */
private boolean panicModeRecovery() {

    // The error is an invalid node name
    // or its absence.

    // Recovery possibility - if the current
    // or the next tokens is an opening parenthese.
    while (! lookahead("{") && // for Nodes
           ! lookahead("}") && // for ValueTypes
           lookahead != null) {
        nextToken();
    }

```

```

    }

    // Trying to recover by reading parentheses
    // until the end of the "damaged" Node is reached
    if (lookahead("{")) {
        int parentheses = 1;
        while (nextToken()) {
            if (lookahead("{")) {
                parentheses++;
            } else if (lookahead("}")) {
                parentheses--;
            }
            if (parentheses == 0) {
                nextToken();
                return true;
            }
        }
    }

    // ValueType
    if (lookahead("}")) {
        return true;
    }

    return false;
}

/*****
 *      Building up the JavaBeans components.      *
 *****/

/**
 * Instantiates the next Node Bean by its type.
 */
private boolean instantiateNode() {
    try {
        // Uses REFLECTION
        Node node = createInstance(currentType);

        // If there was the "DEF" keyword
        if (currentId != null) {
            node.setId(currentId);

            // Warning if the named node is already defined
            if (defNodesTable.containsKey(currentId)) {
                registerError(new Warning("Node named '" + currentId +
                    "' is already defined", tokenizer.lineno()));
            }

            // Store the node in the hash table
            defNodesTable.put(currentId, node);
        }

        // Pushing the node into the stack
        currentNodes.push(node);

        System.out.println("Instantiated Node" +
            (currentId == null ? "" : (" '" + currentId + "'")
            + " of type " + currentType);

        currentId = null;

        return true;
    } catch (Exception e) {
        return false;
    }
}

/**
 * Searches for an existing Node Bean in the hash table

```



```

* by its id and, if found, acts like instantiateNode().
*
* @return true
*/
private boolean instantiateNodeById() {
    Node node = defNodesTable.get(currentId);

    // can be null
    currentNodes.push(node);

    if (node == null) {
        registerError(new LexicalError("Node named '" + currentId +
            "' is not declared.", tokenizer.lineno(), currentId,
            new HashSet<String>(defNodesTable.keySet())));
    } else {
        System.out.println("Instantiated existing Node" +
            (currentId == null ? "" : (" '" + currentId + "'")));
    }

    currentId = null;

    return true;
}

/**
 * Adds the parsed node into the sceneGraph array.
 */
private boolean addRootNode() {
    System.out.print("Added root node ");
    System.out.println(currentNodes.peek() == null ? "null" :
        currentNodes.peek().getClass().getSimpleName());
    System.out.println();

    sceneGraph.add(currentNodes.pop());

    return true;
}

/**
 * Gets the value of the next field and stores it
 * in the appropriate Bean. Works both for value types
 * and for Node types (recursively).
 */
private boolean matchFieldValueAndSetField() {
    // ! NB: Here, it was NOT the grammar which gave the information
    // on the type of the field. To retrieve it, reflection was used.

    Object value = null;
    Class<?> currentFieldType;

    try {
        currentFieldType = currentNodes.peek().getClass().getDeclaredMethod("get" +
            Character.toUpperCase(currentField.peek().charAt(0)) +
            currentField.peek().substring(1)).getReturnType();
    } catch (Exception e) {
        return false;
    }

    /***** a) Node type => recursive call of the appropriate parser Methods *****/
    if (Node.class.isAssignableFrom(currentFieldType)) {
        /*****
         * sfnodeValue ::=
         *     nodeStatement |
         *     NULL
         *****/
        if (tryMatch("NULL")) {
            value = null;

```

```

    } else {

        // involves currentNodes.push(...)
        parseNodeStatement();

        // after parseNodeStatement the node is on the top
        value = currentNodes.pop();
    }
}

/***** b) Value type => call "parse" method in the type class via reflection *****/
else {
    // Implementation - in the superclass.
    value = parseValueType(currentFieldType);
}

/***** Invoking setXxx(value) *****/
try {
    currentNodes.peek().getClass().getDeclaredMethod("set" +
        Character.toUpperCase(currentField.peek().charAt(0)) +
        currentField.peek().substring(1),
        new Class[] {currentFieldType}).
        invoke(currentNodes.peek(), value);

    System.out.println("    Set the " +
        currentField.peek()
        + " field to value " +
        ((value == null) ? "null" : ("of type " +
        value.getClass().getName() +
        ": " + value.toString())));

    currentField.pop();

    return true;
} catch (Exception e) {
    return false;
}
}

/*****
 * (Private fields).
 *****/

/**
 * Initializes the private fields before the parser starts.
 */
private void initFields() {
    defNodesTable = new HashMap<String, Node>();
    //protoNodesTable = new HashMap<String, Node>();
    currentNodes = new Stack<Node>();
    currentField = new Stack<String>();
    currentId = null;
}

/*
 * Hash table that stores named (DEF) nodes
 * for their further use in USE statements.
 */
private HashMap<String, Node> defNodesTable;
//private HashMap<String, Node> protoNodesTable;

/* current Token */
protected String lookahead;

/* current Node id */
private String currentId;
/* current Node type */
private String currentType;

// ! NB: The nested structure of VRML nodes requires
//       maintaining of two stacks: for field names
//       and for the appropriate nodes (if needed).

```

```

    /* Field stack */
    private Stack<String> currentField;
    /* Node stack */
    private Stack<Node> currentNodes;
}

```

### ***1.3. X3DParser.java***

```

package ru.hse.se.parsers;

import ru.hse.se.nodes.Node;
import ru.hse.se.parsers.errors.SyntaxError;
import ru.hse.se.types.MFNode;

import java.io.IOException;
import java.io.StreamTokenizer;
import java.util.HashMap;
import java.util.Stack;

/**
 * XML parser. Builds up a bunch of beans
 * on the basis of its declarative description.
 *
 * @author Mikhail Dubov
 */
public class X3DParser extends Parser {

    /**
     * Sets up the tokenizer object
     * according to the XML grammar.
     * (defines terminals etc.)
     */
    @Override
    protected void setUpTokenizer() {

        super.setUpTokenizer();

        // TODO: comments??

        // Terminals
        tokenizer.ordinaryChar('<');
        tokenizer.ordinaryChar('>');
        tokenizer.ordinaryChar('/');
        tokenizer.ordinaryChar('=');
        tokenizer.ordinaryChar('\\'); // Reading attributes manually

        //tokenizer.parseNumbers(); // => No! Bad for advanced float/int32 parsing
        tokenizer.lowerCaseMode(false); // X3D is case-sensitive
        tokenizer.eolIsSignificant(false); // We can count lines with tokenizer.lineno()
    }

    /**
     * XML parser built according to the SAX approach.
     * (that is, an event-driven parser)
     */
    /**
     * Performs the parsing of the input file
     * and returns an ArrayList of root nodes.
     */
    @Override
    protected void parseScene() throws IOException {

        parseXML();
    }

    /**
     * The main parsing routine that
     * goes through the XML file
     * and reports events, such as
     * opening tag, closing tag etc.
     */
    private void parseXML() {

```

```

// Works like a DFA.
while (lookahead != null) {

    // 1. Opening or closing tag starts
    if (tryMatch("<")) {

        readingTag = true;

        if (lookahead("/")) {
            match("/");
            if (currentTags.isEmpty()) {
                registerError(new SyntaxError ("Closing tag + ' " + lookahead
                    + "' does not match any opening tag.",
                    tokenizer.lineno()));
            } else {
                String openingTag = currentTags.pop();

                if (! lookahead(openingTag)) {
                    registerError(new SyntaxError ("Closing tag + ' " + lookahead
                        + "' does not match the opening tag + ' "
                        + openingTag + "'.", tokenizer.lineno()));
                } else {
                    closingTag(lookahead);
                }
            }

            nextToken();
            match(">");
            readingTag = false;
        } else {
            currentTags.push(lookahead);
            openingTag(lookahead);

            nextToken();
        }
    }
    // 2. Opening tag ends
    else if (tryMatch(">")) {

        readingTag = false;
    }
    // 3. Opening tag is closed at once
    else if (tryMatch("/")) {

        match(">");

        readingTag = false;
        closingTag(currentTags.pop());
    }
    // 4. Attribute
    else if (readingTag) {
        matchAttributeId();
        match("=");

        attribute(currentAttribute);
    }
    // 5. Text node
    else {
        StringBuilder value = new StringBuilder();
        do {
            value.append(lookahead);
            value.append(" ");
            nextToken();
        } while(! lookahead("<"));
        textNode(value.toString());
    }
}

}

/**
 * Parses the next Node from the input stream.
 * Needed for MFNode parsing.
 */

```

```

@Override
public Node parseChildNode() {
    return null;
}

/*****
 *                      Events.                      *
 *****/

/**
 * Called whenever the parser meets an opening tag.
 *
 * @param name tag name
 */
private void openingTag(String name) {

    System.out.println("Opening: " + name);

    if (name.equals("X3D") || name.equals("Scene") ||
        name.equals("fieldValue")) {

        // X3D and Scene are simply
        // root nodes with no functionality;

        // fieldValue is used to read fields,
        // including nested MFNode values.

        return;
    }

    // Nested nodes (SFNode/MFNode); not value types
    try {

        // Uses REFLECTION
        Node currentNode = createInstance(name);

        // If the second top tag is <fieldValue>,
        // then we have one of the nodes in MFNode value
        boolean isMFNode = false;
        if (currentTags.size() > 1) {
            String temp = currentTags.pop();
            if (currentTags.peek().equals("fieldValue")) {
                isMFNode = true;
            }
            currentTags.push(temp);
        }

        // MFNode
        if (isMFNode) {
            fieldValueMFNodes.peek().add(currentNode);
        }
        // SFNode
        else {
            if (!currentNodes.isEmpty()) {

                // Child node is some field of the parent node.
                // To determine which field is to be set,
                // we use the containerField property.

                Node parentNode = currentNodes.peek();

                String field = currentNode.containerField();

                Class<?> currentFieldType = parentNode.getClass().
                    getDeclaredMethod("get" +
                        Character.toUpperCase(field.charAt(0)) +
                        field.substring(1)).getReturnType();

                /***** Invoking setXxx(value) *****/
                parentNode.getClass().getDeclaredMethod("set" +
                    Character.toUpperCase(field.charAt(0)) + field.substring(1),
                    new Class[] {currentFieldType}).
                    invoke(currentNodes.peek(), currentNode);
            }
        }
    }
}

```

```

    }
}

currentNodes.push(currentNode);

} catch (Exception e) {
    registerError(new Error("Could not instantiate node " + name));
}
}

/**
 * Called whenever the parser meets a closing tag.
 *
 * @param name tag name
 */
private void closingTag(String name) {

    System.out.println("Closing: " + name);

    if (name.equals("X3D") || name.equals("Scene")) {

        // X3D and Scene are simply
        // root nodes with no functionality
        return;
    }

    if (name.equals("fieldValue")) {

        // Pop the MFNode value from stack,
        // if there is one on the top
        Class<?> fieldType = null;

        try {
            fieldType = currentNodes.peek().getClass().
                getDeclaredMethod("get" +
                    Character.toUpperCase(fieldValueNameAttributes.peek().
                        charAt(0)) + fieldValueNameAttributes.peek().
                        substring(1)).getReturnType();
        } catch (Exception e) { }
        if (MFNode.class.isAssignableFrom(fieldType)) {
            fieldValueMFNodes.pop();
        }

        // Pop the last field name from the stack
        fieldValueNameAttributes.pop();

        return;
    }

    Node closed = currentNodes.pop();

    // Adds a root node to the sceneGraph array
    if (currentNodes.isEmpty()) {
        sceneGraph.add(closed);
    }
}

/**
 * Called whenever the parser meets an attribute
 * inside the opening tag.
 *
 * @param name attribute name
 */
private void attribute(String name) {

    System.out.println("Attribute: " + name);

    match("");

    // DEF keyword
    if (name.equals("DEF")) {

        Node currentNode = currentNodes.peek();
        currentNode.setId(lookahead);

        defNodesTable.put(lookahead, currentNode);
    }
}

```

```

        nextToken();
    }
    // USE keyword
    else if (name.equals("USE")) {

        // The just instantiated Node was a "fake node"
        currentNodes.pop();

        // Get the Node from the hash table
        Node node = defNodesTable.get(lookahead);

        if (node != null) {
            if (! currentNodes.isEmpty()) {

                // Child node is some field of the parent node.
                // To determine which field is to be set,
                // we use the containerField property.

                Node parentNode = currentNodes.peek();
                String field = node.containerField();

                try {

                    Class<?> currentFieldType = parentNode.getClass().
                        getDeclaredMethod("get" +
                            Character.toUpperCase(field.charAt(0)) +
                            field.substring(1)).getReturnType();

                    /***** Invoking setXxx(value) *****/
                    parentNode.getClass().getDeclaredMethod("set" +
                        Character.toUpperCase(field.charAt(0)) + field.substring(1),
                        new Class[] {currentFieldType}).
                        invoke(currentNodes.peek(), node);
                } catch (Exception e) {

                    registerError(new Error("Could not use node " + lookahead));
                }
            }

            currentNodes.push(node);
        } else {

            registerError(new SyntaxError("Node named '" + lookahead +
                "' is not declared.", tokenizer.lineno()));
        }

        nextToken();
    }
    // Reading a field name through a special tag
    // it may be given for an MFNode.
    else if (name.equals("name") && currentTags.peek().equals("fieldValue")) {

        fieldValueNameAttributes.push(lookahead);
        Class<?> fieldType = null;

        try {
            fieldType = currentNodes.peek().getClass().
                getDeclaredMethod("get" +
                    Character.toUpperCase(lookahead.charAt(0)) +
                    lookahead.substring(1)).getReturnType();
        } catch (Exception e) {
            registerError(new SyntaxError("Field " + lookahead +
                " is not declared.", tokenizer.lineno()));
        }
        if (MFNode.class.isAssignableFrom(fieldType)) {
            try {
                MFNode value = (MFNode)(fieldType.newInstance());
                fieldValueMFNodes.push(value);

                /***** Invoking setXxx(value) *****/
                currentNodes.peek().getClass().getDeclaredMethod("set" +
                    Character.toUpperCase(lookahead.charAt(0)) +
                    lookahead.substring(1),

```

```

        new Class[] {fieldType}).
        invoke(currentNodes.peek(), value);
    } catch (Exception e) {
        registerError(new Error("Could not set the value of " +
                                " field " + lookahead));
    }
}

nextToken();
}
// Reading a field value through a special tag
else if (name.equals("value") && currentTags.peek().equals("fieldValue")) {
    String fieldName = fieldValueNameAttributes.peek();
    matchFieldValueAndSetField(fieldName);
}
// Fields (value types, NOT nested nodes)
else {
    matchFieldValueAndSetField(name);
}

match("");
}

/**
 * Called whenever the parser meets a text node.
 *
 * @param value text
 */
private void textNode(String value) {

    System.out.println("Text node: " + value);

    registerError(new SyntaxError("No text nodes allowed in X3D format",
                                tokenizer.lineno()));
}

/*****
 * Building up the JavaBeans components.
 *****/

/**
 * Gets the value of the given field and stores it
 * in the appropriate Bean.
 *
 * @param name field name
 */
private void matchFieldValueAndSetField(String name) {
    Node currentNode = currentNodes.peek();
    Class<?> currentFieldType;

    try {
        /***** Getting the field type *****/
        currentFieldType = currentNode.getClass().
            getDeclaredMethod("get" +
                Character.toUpperCase(name.charAt(0)) +
                name.substring(1)).getReturnType();

        Object attrValue = parseValueType(currentFieldType);

        /***** Invoking setXxx(value) *****/
        currentNode.getClass().getDeclaredMethod("set" +
            Character.toUpperCase(name.charAt(0)) + name.substring(1),
            new Class[] {currentFieldType}).
            invoke(currentNode, attrValue);

    } catch (Exception e) {
        registerError(new Error("Could not set the value of field " + name));
    }
}

/*****
 * Token operations.
 *****/

```



```

*****/

/**
 * Initializes the parser by reading the first
 * token and storing it in the lookahead variable.
 */
protected void init() throws IOException {

    initFields();

    nextToken();

    if (lookahead == null) {
        throw new IOException();
    }
}

/**
 * Returns the lookahead token.
 */
public String lookahead() {
    return lookahead;
}

/**
 * Checks a token for being a valid Id.
 *
 * @param id
 * @return true if the token is a correct id, false otherwise
 */
private boolean lookaheadIsId() {

    // TODO: More Id checking (see rules)

    return (lookahead != null) && (lookahead != "") &&
        (Character.isLetter(lookahead.charAt(0)) || lookahead.charAt(0) == '_');
}

/**
 * Reads the next token that represents an Id.
 *
 * @return true, if matching is successful
 */
private boolean matchAttributeId() {

    if(lookaheadIsId()) {

        currentAttribute = lookahead;
        nextToken();

    } else {

        registerError(new SyntaxError("'" + lookahead + "' is not a valid id",
            tokenizer.lineno()));

    }

    return true;
}

/**
 * Reads the next token from the input.
 * @returns false when the next token is unavailable, true otherwise
 */
@Override
public boolean nextToken() {

    try {
        int ttype = tokenizer.nextToken();

        if (ttype == '<' || ttype == '>' ||
            ttype == '/' || ttype == '=' ||
            ttype == '\\') {
            // Terminals
            lookahead = String.valueOf(((char)tokenizer.ttype));
        } else if (ttype == StreamTokenizer.TT_WORD) {
            // Non-terminals

```

```

        lookahead = tokenizer.sval;
    } else if (ttype == '"') {
        // Quoted Strings
        lookahead = tokenizer.sval;
    } else if (ttype == StreamTokenizer.TT_EOF) {
        lookahead = null; // to indicate EOF
        return false;
    } // No TT_NUMBER or TT_EOL can arise
} catch (IOException e) {
    return false;
}

return true;
}

/*****
 * (Private fields).
 *****/

/**
 * Initializes the private fields before the parser starts.
 */
private void initFields() {
    defNodesTable = new HashMap<String, Node>();
    //protoNodesTable = new HashMap<String, Node>();
    currentNodes = new Stack<Node>();
    currentTags = new Stack<String>();
    readingTag = false;
    fieldValueNameAttributes = new Stack<String>();
    fieldValueMFNodes = new Stack<MFNode>();
}

/*
 * Hash table that stores named (DEF) nodes
 * for their further use in USE statements.
 */
private HashMap<String, Node> defNodesTable;
//private HashMap<String, Node> protoNodesTable;

/* current Token */
private String lookahead;

/** Determines whether we are inside a tag */
private boolean readingTag;

/* current Attribute id */
private String currentAttribute;

// ! NB: The nested structure of XML nodes requires
//        maintaining of two stacks: for tag names
//        and for the appropriate nodes (if needed).

/* Tag stack */
private Stack<String> currentTags;
/* Node stack */
private Stack<Node> currentNodes;
/* For nested MFNodes */
private Stack<String> fieldValueNameAttributes;
private Stack<MFNode> fieldValueMFNodes;
}

```

## 2. Paket ru.hse.se.parsers.errors

### 2.1. *ParsingError.java*

```

package ru.hse.se.parsers.errors;

public abstract class ParsingError extends Error {

```

```

public ParsingError(String msg, int line) {
    super("Line " + line + ": " + msg);
}

public int getLine() {
    return line;
}

protected int line;

private static final long serialVersionUID = 1L;
}

```

## 2.2. *LexicalError.java*

```

package ru.hse.se.parsers.errors;

import java.util.ArrayList;
import java.util.HashSet;

public class LexicalError extends ParsingError {

    public LexicalError(String error, int line, String badToken,
        HashSet<String> possibleSubstitutions) {

        super(error +
            suggestSubstitutions(badToken, possibleSubstitutions), line);
    }

    private static String suggestSubstitutions(String badToken,
        HashSet<String> possibleSubstitutions) {

        if (possibleSubstitutions == null) {
            return "";
        }

        // !!! Suggesting substitutions !!!
        ArrayList<String> toSuggest = new ArrayList<String>();
        String subst;

        // 1. Transposing of adjacent letters
        for (int i = 0; i < badToken.length()-1; i++) {
            subst = badToken.substring(0, i) +
                badToken.charAt(i+1) +
                badToken.charAt(i) +
                badToken.substring(i+2);
            if (possibleSubstitutions.contains(subst)) {
                toSuggest.add(subst);
            }
        }

        // 2. Removal of each letter
        for (int i = 0; i < badToken.length(); i++) {
            subst = badToken.substring(0, i) + badToken.substring(i+1);
            if (possibleSubstitutions.contains(subst) &&
                (i == 0 || badToken.charAt(i) != badToken.charAt(i-1))) {
                toSuggest.add(subst);
            }
        }

        // 3. Replacement of each letter
        for (int i = 0; i < badToken.length(); i++) {
            for (char c = 'a'; c <= 'z'; c++) {
                subst = badToken.substring(0, i) +
                    c + badToken.substring(i+1);
                if (possibleSubstitutions.contains(subst)) {
                    toSuggest.add(subst);
                }
            }
        }

        // 4. Inserting any letter at any position in a word
        for (int i = 0; i < badToken.length(); i++) {
            for (char c = 'a'; c <= 'z'; c++) {

```

```

        subst = badToken.substring(0, i) +
            c + badToken.substring(i);
        if (possibleSubstitutions.contains(subst) &&
            subst.charAt(i) != subst.charAt(i+1)) {
            toSuggest.add(subst);
        }
    }
}

if (! toSuggest.isEmpty()) {
    String res = " (did you mean ";
    for (String s : toSuggest) {
        res += ("'" + s + "'/");
    }
    res = res.substring(0, res.length()-1) + "?)";
    return res;
} else {
    return "";
}
}

private static final long serialVersionUID = 1L;
}

```

### ***2.3. SyntaxError.java***

```

package ru.hse.se.parsers.errors;

public class SyntaxError extends ParsingError {

    public SyntaxError(String error, int line) {

        super(error, line);
    }

    private static final long serialVersionUID = 1L;
}

```

### ***2.4. TypeMismatchError.java***

```

package ru.hse.se.parsers.errors;

public class TypeMismatchError extends ParsingError {

    public TypeMismatchError(Class<?> given, Class<?> required, int line) {

        super("type '" + given.getSimpleName() +
            "' does not match type '" +
            required.getSimpleName() + "'",
            line);
    }

    private static final long serialVersionUID = 1L;
}

```

### ***2.5. Warning.java***

```

package ru.hse.se.parsers.errors;

public class Warning extends ParsingError {

    public Warning(String error, int line) {

        super(error, line);
    }
}

```

```

    }

    private static final long serialVersionUID = 1L;
}

```

### 3. Пакет ru.hse.se.codegenerators

#### 3.1. *CodeGenerator.java*

```

package ru.hse.se.codegenerators;

import java.io.InputStreamReader;
import java.io.PrintStream;
import java.util.ArrayList;
import ru.hse.se.nodes.Node;
import ru.hse.se.parsers.VRMLParser;
import ru.hse.se.parsers.X3DParser;

/**
 * Represents an abstract code generator,
 * which can generate code by
 * introspecting the scene graph.
 *
 * @author Mikhail Dubov
 */
public abstract class CodeGenerator {

    /**
     * Generates code by introspecting the scene graph.
     *
     * @param sceneGraph the scene graph
     * @param output the output stream
     */
    public abstract void generate(ArrayList<Node> sceneGraph, PrintStream output);

    /**
     * Converts a VRML representation
     * of the scene graph into an X3D code file.
     *
     * @param input The input stream that contains VRML code
     * @param output The output stream for X3D code
     * @return true, if the conversion succeeded, false otherwise
     */
    public static boolean VRMLtoX3D(InputStreamReader input, PrintStream output) {
        try {
            ArrayList<Node> sceneGraph = (new VRMLParser()).parse(input);
            (new X3DCodeGenerator()).generate(sceneGraph, output);

            return true;

        } catch (Exception e) {
            return false;
        }
    }

    /**
     * Converts a X3D representation
     * of the scene graph into an VRML code file.
     *
     * @param input The input stream that contains X3D code
     * @param output The output stream for VRML code
     * @return true, if the conversion succeeded, false otherwise
     */
    public static boolean X3DtoVRML(InputStreamReader input, PrintStream output) {
        try {
            ArrayList<Node> sceneGraph = (new X3DParser()).parse(input);
            (new VRMLCodeGenerator()).generate(sceneGraph, output);

            return true;
        }
    }
}

```

```

        } catch (Exception e) {
            return false;
        }
    }
}

```

### 3.2. *VRMLCodeGenerator.java*

```

package ru.hse.se.codegenerators;

import java.io.PrintStream;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Stack;

import ru.hse.se.nodes.Node;
import ru.hse.se.types.MFNode;
import ru.hse.se.types.ValueType;

/**
 * VRML code generator,
 * which can generate code by
 * introspecting the scene graph.
 *
 * @author Mikhail Dubov
 */
public class VRMLCodeGenerator extends CodeGenerator {

    /**
     * Generates code by introspecting the scene graph.
     *
     * @param sceneGraph the scene graph
     * @param output the output stream
     */
    public void generate(ArrayList<Node> sceneGraph, PrintStream output) {

        defNodes = new HashSet<String>();
        nodes = new Stack<Node>();
        this.output = output;

        for (int i = 0; i < sceneGraph.size(); i++) {
            process(sceneGraph.get(i));
            output.println();
        }
    }

    private void process(Node n) {

        nodes.push(n);

        if (n.getId() != null) {
            // Already described; write "USE"
            if (defNodes.contains(n.getId())) {

                output.println("USE " + n.getId());
                nodes.pop();
                return;
            }
            // Node name should be stored in hash table
            else {
                output.print("DEF " + n.getId() + " ");
                defNodes.add(n.getId());
            }
        }
        output.println(n.getClass().getSimpleName() + " {}");

        try {

            Method[] methods = n.getClass().getDeclaredMethods();
            for (Method m : methods) {

                if (m.getName().startsWith("get")) {

```

```

String field = Character.toLowerCase(m.getName().charAt(3)) +
    m.getName().substring(4);

// Node type => process recursively
if (Node.class.isAssignableFrom(m.getReturnType())) {

    Node child = (Node)m.invoke(n);

    if (child != null) {
        for (int i = 0; i < nodes.size(); i++) {
            output.print(" ");
        }
        output.print(field + " ");
        process(child);
    }
}
// Multiple node type => process them all
else if (MFNode.class.isAssignableFrom(m.getReturnType())) {

    MFNode value = (MFNode)m.invoke(n);
    for (int i = 0; i < nodes.size(); i++) {
        output.print(" ");
    }
    output.println(field + " [");

    nodes.push(null); // Fake node; just for code indent

    for (Node child : value.getValue()) {

        for (int i = 0; i < nodes.size(); i++) {
            output.print(" ");
        }
        process(child);
    }

    nodes.pop();

    for (int i = 0; i < nodes.size(); i++) {
        output.print(" ");
    }
    output.println("]");
}
// Value type => print value
else if (ValueType.class.isAssignableFrom(m.getReturnType())) {

    ValueType value = (ValueType)m.invoke(n);
    for (int i = 0; i < nodes.size(); i++) {
        output.print(" ");
    }
    output.println(field + " " + value);
}
// Other => Java primitive type
else {

    // TODO: check for accepted types
    Object value = m.invoke(n);
    for (int i = 0; i < nodes.size(); i++) {
        output.print(" ");
    }
    output.println(field + " " + value);
}
}
} catch (Exception e) {}

for (int i = 0; i < nodes.size()-1; i++) {
    output.print(" ");
}

output.println("}");

nodes.pop();
}

Stack<Node> nodes;
PrintStream output;

```

```

    HashSet<String> defNodes;
}

```

### 3.3. *X3DCodeGenerator.java*

```

package ru.hse.se.codegenerators;

import java.io.PrintStream;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Stack;
import ru.hse.se.nodes.Node;
import ru.hse.se.types.MFNode;
import ru.hse.se.types.MFValueType;
import ru.hse.se.types.ValueType;

/**
 * X3D code generator,
 * which can generate code by
 * introspecting the scene graph.
 *
 * @author Mikhail Dubov
 */
public class X3DCodeGenerator extends CodeGenerator {

    /**
     * Generates code by introspecting the scene graph.
     *
     * @param sceneGraph the scene graph
     * @param output the output stream
     */
    public void generate(ArrayList<Node> sceneGraph, PrintStream output) {

        defNodes = new HashSet<String>();
        nodes = new Stack<Node>();
        this.output = output;

        output.println("<Scene>");
        nodes.push(null); // Simulates the Scene node
        for (int i = 0; i < sceneGraph.size(); i++) {
            process(sceneGraph.get(i));
            output.println();
        }
        nodes.pop();
        output.println("</Scene>");
    }

    private void process(Node n) {

        nodes.push(n);

        for (int i = 0; i < nodes.size()-1; i++) {
            output.print(" ");
        }

        if (n.getId() != null) {
            // Already described; write "USE"
            if (defNodes.contains(n.getId())) {

                output.println("<" + n.getClass().getSimpleName() +
                    " USE='" + n.getId() + "' />");
                nodes.pop();
                return;
            }
            // Node name should be stored in hash table
            else {
                output.print("<" + n.getClass().getSimpleName() +
                    " DEF='" + n.getId() + "'");
                defNodes.add(n.getId());
            }
        } else {

```



```

    output.print("<" + n.getClass().getSimpleName());
}

try {
    Method[] methods = n.getClass().getDeclaredMethods();

    for (Method m : methods) {

        if (m.getName().startsWith("get")) {

            // Value type => attribute
            if (ValueType.class.isAssignableFrom(m.getReturnType())) {

                String field = Character.toLowerCase(m.getName().charAt(3)) +
                    m.getName().substring(4);

                ValueType value = (ValueType)m.invoke(n);

                // MFNodes - processed later
                if (! (value instanceof MFNode)) {

                    // Different patterns of printing values (!)
                    if (value instanceof MFValueType) {
                        // Erasing '[' and ']'
                        output.print(" " + field + "=" +
                            value.toString().substring(2,
                                value.toString().length()-2) + "");
                    } else {
                        output.print(" " + field + "=" + value + "");
                    }
                }
            }
        }
    }

    output.println(">");

    for (Method m : methods) {

        if (m.getName().startsWith("get")) {

            // Nested nodes
            if (Node.class.isAssignableFrom(m.getReturnType())) {

                Node child = (Node)m.invoke(n);
                if (child != null) {
                    process(child);
                }
            }
            // MFNodes
            else if (MFNode.class.isAssignableFrom(m.getReturnType())) {

                String field = Character.toLowerCase(m.getName().charAt(3)) +
                    m.getName().substring(4);
                MFNode value = (MFNode)m.invoke(n);

                nodes.push(null); // Fake node; just for code indent

                for (int i = 0; i < nodes.size()-1; i++) {
                    output.print(" ");
                }

                output.println("<fieldValue name='" + field + "'>");

                for (Node child : value.getValue()) {
                    process(child);
                }

                for (int i = 0; i < nodes.size()-1; i++) {
                    output.print(" ");
                }

                output.println("</fieldValue>");

                nodes.pop();
            }
        }
    }
}

```

```

        }
    }
}

} catch (Exception e) {}

for (int i = 0; i < nodes.size()-1; i++) {
    output.print(" ");
}

output.println("</" + n.getClass().getSimpleName() + ">");

nodes.pop();
}

Stack<Node> nodes;
PrintStream output;
HashSet<String> defNodes;
}

```

## 4. Пакет ru.hse.se.nodes

### 4.1. *Appearance.java*

```

package ru.hse.se.nodes;

import java.io.Serializable;

public class Appearance extends Node implements Serializable {

    public Appearance() {
        material = new Material();
    }

    public void setMaterial(Material m) {
        material = m;
    }

    public Material getMaterial() {
        return material;
    }

    public String containerField() {
        return "appearance";
    }

    private Material material;

    private static final long serialVersionUID = 1L;
}

```

### 4.2. *Box.java*

```

package ru.hse.se.nodes;

import java.io.Serializable;

public class Box extends Geometry implements Serializable {

    public Box() {
    }

    private static final long serialVersionUID = 1L;
}

```

### 4.3. *Geometry.java*

```

package ru.hse.se.nodes;

import java.io.Serializable;

public abstract class Geometry extends Node implements Serializable {

    public Geometry() {

    }

    public String containerField() {
        return "geometry";
    }

    private static final long serialVersionUID = 1L;
}

```

#### ***4.4. Group.java***

```

package ru.hse.se.nodes;

import java.io.Serializable;

import ru.hse.se.types.MFNode;

public class Group extends Node implements Serializable {

    public Group() {
        children = new MFNode();
    }

    public void setChildren(MFNode ch) {
        children = ch;
    }

    public MFNode getChildren() {
        return children;
    }

    public String containerField() {
        return "children";
    }

    private MFNode children;

    private static final long serialVersionUID = 1L;
}

```

#### ***4.5. Material.java***

```

package ru.hse.se.nodes;

import java.io.Serializable;

import ru.hse.se.types.SFColor;

public class Material extends Node implements Serializable {

    public Material() {
        diffuseColor = new SFColor(0.8, 0.8, 0.8);
    }

    public void setDiffuseColor(SFColor c) {
        diffuseColor = c;
    }

    public SFColor getDiffuseColor() {
        return diffuseColor;
    }

    public String containerField() {

```

```

        return "material";
    }

    private SColor diffuseColor;

    private static final long serialVersionUID = 1L;
}

```

## 4.6. *Node.java*

```

package ru.hse.se.nodes;

import java.io.Serializable;

import ru.hse.se.types.VRMLType;

public abstract class Node extends VRMLType implements Serializable {

    public Node() {
        id = null;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getId() {
        return id;
    }

    public abstract String containerField();

    protected String id;

    private static final long serialVersionUID = 1L;
}

```

## 4.7. *Shape.java*

```

package ru.hse.se.nodes;

import java.io.Serializable;

public class Shape extends Node implements Serializable {

    public Shape() {
        appearance = null;
        geometry = null;
    }

    public void setAppearance(Appearance a) {
        appearance = a;
    }

    public void setGeometry(Geometry g) {
        geometry = g;
    }

    public Appearance getAppearance() {
        return appearance;
    }

    public Geometry getGeometry() {
        return geometry;
    }

    public String containerField() {
        return "children";
    }

    private Appearance appearance;
    private Geometry geometry;
}

```

```

    private static final long serialVersionUID = 1L;
}

```

## 4.8. *Sphere.java*

```

package ru.hse.se.nodes;

import java.io.Serializable;
import ru.hse.se.types.SFFloat;

public class Sphere extends Geometry implements Serializable {

    public Sphere() {
        radius = new SFFloat(0);
    }

    public void setRadius(SFFloat r) {
        radius = r;
    }

    public SFFloat getRadius() {
        return radius;
    }

    private SFFloat radius;

    private static final long serialVersionUID = 1L;
}

```

## 4.9. *Text.java*

```

package ru.hse.se.nodes;

import java.io.Serializable;
import ru.hse.se.types.MFFloat;
import ru.hse.se.types.MFString;
import ru.hse.se.types.SFFloat;

public class Text extends Geometry implements Serializable {

    public Text() {
        string = new MFString();
        length = new MFFloat();
        maxExtent = new SFFloat(0);
    }

    // public void setFontStyle(FontStyle fst) {
    //     fontStyle = fst;
    // }

    public void setString(MFString str) {
        string = str;
    }

    public void setLength(MFFloat len) {
        length = len;
    }

    public void setMaxExtent(SFFloat ext) {
        maxExtent = ext;
    }

    public MFString getString() {
        return string;
    }

    //public FontStyle getFontStyle() {
    //    return fontStyle;
    //}
}

```

```

    //}

    public MFFloat getLength() {
        return length;
    }

    public SFFloat getMaxExtent() {
        return maxExtent;
    }

    private MFString string;
    //private FontStyle fontStyle;
    private MFFloat length;
    private SFFloat maxExtent;

    private static final long serialVersionUID = 1L;
}

```

## 5. Пакет ru.hse.se.types

### 5.1. MFBool.java

```

package ru.hse.se.types;

import java.util.ArrayList;
import java.util.zip.DataFormatException;
import ru.hse.se.parsers.Parser;

public class MFBool extends MFValueType<SFBool> {

    public MFBool(ArrayList<SFBool> value) {
        super(value);
    }

    public MFBool() {
        super();
    }

    public static MFBool parse(Parser parser) {
        return MFValueType.<SFBool, MFBool>parseGeneric
            (parser, MFBool.class, SFBool.class);
    }

    public static MFBool parse(String str) throws DataFormatException {
        return MFValueType.<SFBool, MFBool>parseGeneric
            (str, MFBool.class, SFBool.class);
    }

    public static MFBool tryParse(String str) {
        return MFValueType.<SFBool, MFBool>tryParseGeneric
            (str, MFBool.class, SFBool.class);
    }
}

```

### 5.2. MFFloat.java

```

package ru.hse.se.types;

import java.util.ArrayList;
import java.util.zip.DataFormatException;
import ru.hse.se.parsers.Parser;

public class MFFloat extends MFValueType<SFFloat> {

    public MFFloat(ArrayList<SFFloat> value) {
        super(value);
    }
}

```

```

public MFFloat() {
    super();
}

public static MFFloat parse(Parser parser) {
    return MFValueType.<SFFloat, MFFloat>parseGeneric
        (parser, MFFloat.class, SFFloat.class);
}

public static MFFloat parse(String str) throws DataFormatException {
    return MFValueType.<SFFloat, MFFloat>parseGeneric
        (str, MFFloat.class, SFFloat.class);
}

public static MFFloat tryParse(String str) {
    return MFValueType.<SFFloat, MFFloat>tryParseGeneric
        (str, MFFloat.class, SFFloat.class);
}
}

```

### 5.3. *MFInt32.java*

```

package ru.hse.se.types;

import java.util.ArrayList;
import java.util.zip.DataFormatException;
import ru.hse.se.parsers.Parser;

public class MFInt32 extends MFValueType<SFInt32> {

    public MFInt32(ArrayList<SFInt32> value) {
        super(value);
    }

    public MFInt32() {
        super();
    }

    public static MFInt32 parse(Parser parser) {
        return MFValueType.<SFInt32, MFInt32>parseGeneric
            (parser, MFInt32.class, SFInt32.class);
    }

    public static MFInt32 parse(String str) throws DataFormatException {
        return MFValueType.<SFInt32, MFInt32>parseGeneric
            (str, MFInt32.class, SFInt32.class);
    }

    public static MFInt32 tryParse(String str) {
        return MFValueType.<SFInt32, MFInt32>tryParseGeneric
            (str, MFInt32.class, SFInt32.class);
    }
}

```

### 5.4. *MFNode.java*

```

package ru.hse.se.types;

import java.util.ArrayList;

import ru.hse.se.nodes.Node;
import ru.hse.se.parsers.Parser;
import ru.hse.se.parsers.errors.SyntaxException;

public class MFNode extends MFType<Node> {

    public MFNode(ArrayList<Node> value) {
        super(value);
    }

    public MFNode() {

```

```

        super();
    }

    /**
     * Reads a MFNode value from the stream.
     *
     * *****
     * mfnodeValue ::=
     *     nodeStatement |
     *     [ ] |
     *     [ nodeStatements ]
     * *****
     *
     * TODO: This method is definitely a "crutch". Any improvements?
     */
    public static MFNode parse(Parser parser) throws SyntaxError {

        MFNode res = new MFNode(new ArrayList<Node>());

        try {
            if (parser.lookahead("[") {
                parser.match("[");
                while(! parser.lookahead("]")) {
                    res.add(parser.parseChildNode());
                }
                parser.match("]");
            } else {
                res.add(parser.parseChildNode());
            }
        } catch (Exception e) {}

        return res;
    }
}

```

## 5.5. MFString.java

```

package ru.hse.se.types;

import java.util.ArrayList;
import java.util.zip.DataFormatException;
import ru.hse.se.parsers.Parser;
import ru.hse.se.parsers.errors.SyntaxError;

public class MFString extends MFValueType<SFString> {

    public MFString(ArrayList<SFString> value) {
        super(value);
    }

    public MFString() {
        super();
    }

    public static MFString parse(Parser parser) throws SyntaxError {
        return MFValueType.<SFString, MFString>parseGeneric
            (parser, MFString.class, SFString.class);
    }

    public static MFString parse(String str) throws DataFormatException {

        MFString res = null;

        try {
            res = new MFString();

            // Some simple trim
            while (str.charAt(0) == ' ' || str.charAt(0) == '[') {
                str = str.substring(1);
            }
            while (str.charAt(str.length()-1) == ' ' ||
                str.charAt(str.length()-1) == ']') {

```



```

        str = str.substring(0, str.length()-1);
    }

    // Main loop
    String elem;
    int i = 0;

    while (i < str.length()) {

        elem = "";

        while (i < str.length() && str.charAt(i) != '') {
            i++;
        }
        i++;

        while (i < str.length() && str.charAt(i) != '') {
            elem += str.charAt(i);
            i++;
        }
        i++;

        while (i < str.length() &&
            (str.charAt(i) == ' ' || str.charAt(i) == ',')) {
            i++;
        }

        res.add(SFString.parse(elem));
    }

    } catch (DataFormatException e) {
        throw e;
    } catch (Exception e) {}

    return res;
}

public static MFString tryParse(String str) {
    try {
        return parse(str);
    } catch (DataFormatException e) {
        return null;
    }
}
}

```

## 5.6. *MFType.java*

```

package ru.hse.se.types;

import java.util.ArrayList;

@SuppressWarnings("unchecked")
public abstract class MFType<T> extends ValueType {

    public MFType(ArrayList<T> value) {
        this.value = (ArrayList<T>)(value.clone());
    }

    public MFType() {
        this.value = new ArrayList<T>();
    }

    public ArrayList<T> getValue() {
        return (ArrayList<T>)(this.value.clone());
    }

    public void add(T t) {
        this.value.add(t);
    }

    public void remove(T t) {
        this.value.remove(t);
    }
}

```

```

    }

    public void remove(int i) {
        this.value.remove(i);
    }

    public int size() {
        return this.value.size();
    }

    @Override
    public String toString() {

        StringBuilder res = new StringBuilder();

        res.append("[ ");
        for (T val : value) {
            res.append(val.toString());
            res.append(' ');
        }
        res.append("]");

        return res.toString();
    }

    protected ArrayList<T> value;
}

```

## 5.7. MFValueType.java

```

package ru.hse.se.types;

import java.lang.reflect.InvocationTargetException;
import java.util.ArrayList;
import java.util.zip.DataFormatException;

import ru.hse.se.parsers.Parser;
import ru.hse.se.parsers.VRMLParser;
import ru.hse.se.parsers.X3DParser;
import ru.hse.se.parsers.errors.SyntaxException;

@SuppressWarnings("unchecked")
public abstract class MFValueType<T extends ValueType> extends MFTYPE<T> {

    public MFValueType(ArrayList<T> value) {
        this.value = (ArrayList<T>)(value.clone());
    }

    public MFValueType() {
        this.value = new ArrayList<T>();
    }

    /*
     !! NOT ALLOWED !!
     public static MFTYPE<T> parse() { }
    */

    /**
     * Reads a MFxxx value from the stream.
     *
     * *****
     * mfXxxValue ::=
     *     sfXxxValue |
     *     [ ] |
     *     [ sfXxxValues ] ;
     * sfXxxValues ::=
     *     sfXxxValue |
     *     sfXxxValue sfXxxValues ;
     * *****
     *
     * TODO: This method is definitely a "crutch". Any improvements?
     */
    protected static <S extends ValueType, M extends MFTYPE<S>> M

```

```

    parseGeneric(Parser parser, Class<M> c1M, Class<S> c1S) throws SyntaxError {

M res = null;

try {
    res = c1M.getConstructor(ArrayList.class).newInstance(new ArrayList<S>());

    // VRML
    if (parser instanceof VRMLParser) {
        if (parser.tryMatch("[") {
            while(! parser.lookahead("]")) {
                res.add((S)c1S.getDeclaredMethod("parse",
                    Parser.class).invoke(null, parser));
            }
            parser.match("]");
        } else {
            res.add((S)c1S.parse(parser));
        }
    }
    // XML
    else if (parser instanceof X3DParser) {
        while(! parser.lookahead("'")) {
            res.add((S)c1S.getDeclaredMethod("parse",
                Parser.class).invoke(null, parser));
        }
    }
} catch (Exception e) {}

return res;
}

/**
 * Reads a MFxxx value from the string.
 *
 * *****
 * mfXxxValue ::=
 *     sfXxxValue |
 *     [ ] |
 *     [ sfXxxValues ] ;
 * sfXxxValues ::=
 *     sfXxxValue |
 *     sfXxxValue sfXxxValues ;
 * *****
 *
 * NB: Does not support MFString (because of quotation marks).
 *
 * NB: Behaves like a DFA.
 */
protected static <S extends ValueType, M extends MType<S>> M
    parseGeneric(String str, Class<M> c1M, Class<S> c1S)
        throws DataFormatException {

M res = null;

try {
    res = c1M.getConstructor().newInstance();

    // Some simple trim
    while (str.charAt(0) == ' ' || str.charAt(0) == '[') {
        str = str.substring(1);
    }
    while (str.charAt(str.length()-1) == ' ' ||
        str.charAt(str.length()-1) == ']') {
        str = str.substring(0, str.length()-1);
    }

    // Main loop
    String elem;
    int i = 0;

    while (i < str.length()) {

        elem = "";

        while (i < str.length() &&

```

```

        str.charAt(i) != ' ' && str.charAt(i) != ',') {
            elem += str.charAt(i);
            i++;
        }

        while (i < str.length() &&
            (str.charAt(i) == ' ' || str.charAt(i) == ',')) {
            i++;
        }

        res.add((S)clS.getDeclaredMethod("parse",
            String.class).invoke(null, elem));
    }

    } catch (InvocationTargetException e) {
        throw new DataFormatException(e.getMessage());
    } catch (Exception e) {}

    return res;
}

protected static <S extends ValueType, M extends MFType<S>> M
    tryParseGeneric(String str, Class<M> clM, Class<S> clS) {

    try {
        return parseGeneric(str, clM, clS);
    } catch (DataFormatException e) {
        return null;
    }
}
}

```

## 5.8. *SFBool.java*

```

package ru.hse.se.types;

import java.util.zip.DataFormatException;
import ru.hse.se.parsers.Parser;
import ru.hse.se.parsers.errors.SyntaxException;

public class SFBool extends ValueType {

    public SFBool(boolean value) {
        this.value = value;
    }

    public boolean getValue() {
        return value;
    }

    /**
     * Parses a boolean / SFBool value from the stream.
     *
     * *****
     * sfboolValue ::=
     *     TRUE |
     *     FALSE
     * *****
     */
    public static SFBool parse(Parser parser) {

        SFBool res = new SFBool(false);

        try {
            res = parse(parser.lookahead());
            parser.nextToken();
        } catch (DataFormatException e) {
            parser.registerError(new SyntaxError(e.getMessage(),
                parser.tokenizer().lineno()));
        }

        return res;
    }
}

```

```

public static SFBool parse(String str) throws DataFormatException {
    if (str.toUpperCase().equals("TRUE")) {
        return new SFBool(true);
    } else if (str.toUpperCase().equals("FALSE")) {
        return new SFBool(false);
    } else {
        throw new DataFormatException("Expected 'TRUE' or 'FALSE', "+
            "but got '" + str + "'");
    }
}

public static SFBool tryParse(String str) {
    try {
        return parse(str);
    } catch (DataFormatException e) {
        return null;
    }
}

@Override
public String toString() {
    return value ? "TRUE" : "FALSE";
}

private boolean value;
}

```

## 5.9. *SFColor.java*

```

package ru.hse.se.types;

import java.util.zip.DataFormatException;
import ru.hse.se.parsers.Parser;

public class SFColor extends ValueType {

    public SFColor(double r, double g, double b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }

    public double getR() {
        return r;
    }

    public double getG() {
        return g;
    }

    public double getB() {
        return b;
    }

    /**
     * Reads a SFColor value from the stream.
     *
     * *****
     * sfcolorValue ::=
     *      float float float
     * *****
     */
    public static SFColor parse(Parser parser) {

        SFFloat r = SFFloat.parse(parser);
        SFFloat g = SFFloat.parse(parser);
        SFFloat b = SFFloat.parse(parser);

        return new SFColor(r.getValue(), g.getValue(), b.getValue());
    }

    public static SFColor parse(String str) throws DataFormatException {

```

```

String r = "", g = "", b = "";
int i = 0;

while (i < str.length() &&
      str.charAt(i) != ' ' && str.charAt(i) != ',') {
    r += str.charAt(i);
    i++;
}

while (i < str.length() &&
      (str.charAt(i) == ' ' || str.charAt(i) == ',')) {
    i++;
}

while (i < str.length() &&
      str.charAt(i) != ' ' && str.charAt(i) != ',') {
    g += str.charAt(i);
    i++;
}

while (i < str.length() &&
      (str.charAt(i) == ' ' || str.charAt(i) == ',')) {
    i++;
}

while (i < str.length() &&
      str.charAt(i) != ' ' && str.charAt(i) != ',') {
    b += str.charAt(i);
    i++;
}

return new SFFloat(SFFloat.parse(r).getValue(),
                  SFFloat.parse(g).getValue(),
                  SFFloat.parse(b).getValue());
}

public static SFFloat tryParse(String str) {
    try {
        return parse(str);
    } catch (DataFormatException e) {
        return null;
    }
}

@Override
public String toString() {
    return (r + " " + g + " " + b);
}

private double r, g, b;
}

```

## 5.10. *SFFloat.java*

```

package ru.hse.se.types;

import java.util.zip.DataFormatException;
import ru.hse.se.parsers.Parser;
import ru.hse.se.parsers.errors.SyntaxException;

public class SFFloat extends ValueType {

    public SFFloat(double value) {
        this.value = value;
    }

    public double getValue() {
        return value;
    }

    /**

```

```

* Reads a double / SFFloat value from the stream.
*
*****
* sffloatValue ::=
*   floating point number in
*   ANSI C floating point format
*****
*/
public static SFFloat parse(Parser parser) {

    SFFloat res = new SFFloat(0);

    try {
        res = parse(parser.lookahead());
        parser.nextToken();
    } catch (DataFormatException e) {
        parser.registerError(new SyntaxError(e.getMessage(),
                                             parser.tokenizer().lineno()));
    }

    return res;
}

public static SFFloat parse(String str) throws DataFormatException {

    double res = 0;

    try {
        res = Double.parseDouble(str);
    } catch (Exception e) {
        throw new DataFormatException
            ("Expected a double-precision float number" +
             " in ANSI C format, but got '" + str + "'");
    }

    return new SFFloat(res);
}

public static SFFloat tryParse(String str) {
    try {
        return parse(str);
    } catch (DataFormatException e) {
        return null;
    }
}

@Override
public String toString() {
    return String.valueOf(value);
}

private double value;
}

```

### 5.11. *SFInt32.java*

```

package ru.hse.se.types;

import java.util.zip.DataFormatException;
import ru.hse.se.parsers.Parser;
import ru.hse.se.parsers.errors.SyntaxError;

public class SFInt32 extends ValueType {

    public SFInt32(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

```

```

/**
 * Reads an Integer / SInt32 value from the stream.
 *
 * *****
 * sfint32Value ::=
 *   [[+]|-]{{[0-9]+|0x[0-9a-fA-F]+}
 * *****
 */
public static SInt32 parse(Parser parser) {

    SInt32 res = new SInt32(0);

    try {
        res = parse(parser.lookahead());
        parser.nextToken();
    } catch (DataFormatException e) {
        parser.registerError(new SyntaxError(e.getMessage(),
            parser.tokenizer().lineno()));
    }

    return res;
}

public static SInt32 parse(String str) throws DataFormatException {
    int sign = 1;
    if (str.charAt(0) == '+') {
        str = str.substring(1);
    } else if (str.charAt(0) == '-') {
        sign = -1;
        str = str.substring(1);
    }

    int res = 0;

    if(str.startsWith("0x")) { // hex format

        char temp;
        for (int i = 2; i < str.length(); i++) {
            temp = Character.toLowerCase(str.charAt(i));
            if (temp >= '0' && temp <= '9') {
                res = 16*res + (temp-'0');
            } else if (temp >= 'a' && temp <= 'f') {
                res = 16*res + (10+temp-'a');
            } else {
                throw new DataFormatException
                    ("Expected a hexadecimal integer, " +
                     "but got '" + str + "'");
            }
        }
    } else { // decimal format

        try {
            res = Integer.parseInt(str);
        } catch (Exception e) {
            throw new DataFormatException
                ("Expected an integer number, " + "but got '" + str + "'");
        }
    }

    return new SInt32(res*sign);
}

public static SInt32 tryParse(String str) {
    try {
        return parse(str);
    } catch (DataFormatException e) {
        return null;
    }
}

@Override
public String toString() {
    return String.valueOf(value);
}

```



```

    }

    private int value;
}

```

## 5.12. SFString.java

```

package ru.hse.se.types;

import java.util.zip.DataFormatException;
import ru.hse.se.parsers.Parser;
import ru.hse.se.parsers.errors.SyntaxError;

public class SFString extends ValueType {

    public SFString(String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }

    /**
     * Parses a SFString value from the stream.
     *
     * *****
     * sfstringValue ::=
     *     string ;
     * string ::=
     *     "." ... double-quotes must be *
     *     "\", backslashes must be \\... *
     * *****
     */
    public static SFString parse(Parser parser) {
        // TODO: Check whether it is a string (quotation marks)!
        SFString res = new SFString("");

        try {
            res = parse(parser.lookahead());
            parser.nextToken();
        } catch (DataFormatException e) {
            parser.registerError(new SyntaxError(e.getMessage(),
                parser.tokenizer().lineno()));
        }

        return res;
    }

    public static SFString parse(String str) throws DataFormatException {
        // Trim spaces and '"'
        while (str.charAt(0) == ' ') {
            str = str.substring(1);
        }
        while (str.charAt(str.length()-1) == ' ') {
            str = str.substring(0, str.length()-1);
        }
        if (str.charAt(0) == '"') {
            str = str.substring(1);
        }
        if (str.charAt(str.length()-1) == '"') {
            str = str.substring(0, str.length()-1);
        }
        return new SFString(str);
    }

    public static SFString tryParse(String str) {
        try {
            return parse(str);
        } catch (DataFormatException e) {
            return null;
        }
    }
}

```

```

@Override
public String toString() {
    return "'" + value + "'";
}

private String value;
}

```

### 5.13. *ValueType.java*

```

package ru.hse.se.types;

import java.util.zip.DataFormatException;
import ru.hse.se.parsers.Parser;

public abstract class ValueType extends VRMLType {

    /**
     * Parses a value type from the input stream using
     * some parser (that allows to parse both VRML and X3D encoding).
     *
     * @param parser Parser that has the first token of the value as lookahead
     * @return the ValueType object
     */
    public static ValueType parse(Parser parser) {
        return null;
    }

    /**
     * Parses a value type from the input string.
     *
     * @param str String that contains the value
     * @return the ValueType object
     * @throws DataFormatException if there are syntax errors
     */
    public static ValueType parse(String str) throws DataFormatException {
        return null;
    }

    /**
     * Tries to parse a value type from the input string,
     * returns null if parsing didn't succeed.
     *
     * @param strString that contains the value
     * @return the ValueType object or null
     */
    public static ValueType tryParse(String str) {
        return null;
    }
}

```

### 5.14. *VRMLType.java*

```

package ru.hse.se.types;

public abstract class VRMLType {

}

```