実践的プログラミング

氏名: 中村宥哉
学籍番号: C0117237
提出日: 2020/01/31

1. Interpreterの製作

作成したコードを説明する。

ソースコード1 Main.java

```
package newlang4;

public class Main {
    /**
     * @param args
     */
    public static void main(String[] args) {
        LexicalAnalyzer lex;
        LexicalUnit    first;
        Environment    env;

        System.out.println("basic parser\n");
        try {
            lex = new LexicalAnalyzerImpl("./src/newlang4/src.txt");
        } catch (Exception e) {
            e.printStackTrace();
            return;
        }
        env = new Environment(lex);
        try {
            first = lex.peek();
        } catch (Exception e) {
            e.printStackTrace();
            return;
        }

        if (Program.isFirst(first)) {
            Node handler = null;
            try {
                handler = Program.getHandler(first, env);
                handler.parse();
            } catch (Exception e) {
                e.printStackTrace();
                return;
            }
            System.out.println(handler);
            try {
                System.out.println("value = " + handler.getValue());
            } catch (Exception e) {
                e.printStackTrace();
```

```
                return;
            }
        } else {
            System.out.println("syntax error");
        }
    }
}
```

throwsからtry-catchに変更。

38行目のhandler.getValue()を実行することでプログラムが実行される。

ソースコード2 BinExprNode.java

```
package newlang4;

import java.util.Map;
import java.util.HashMap;

public class BinExprNode extends Node {
    Node left = null;
    Node right = null;
    LexicalType operator = null;

    private static final Map<LexicalType, Integer> OPERATORS = new HashMap<>();

    static {
        OPERATORS.put(LexicalType.DIV, 1);
        OPERATORS.put(LexicalType.MUL, 1);
        OPERATORS.put(LexicalType.SUB, 2);
        OPERATORS.put(LexicalType.ADD, 2);
    }

    private BinExprNode (LexicalType operator) {
        this.operator = operator;
        type = NodeType.BIN_EXPR;
    }

    public static boolean isOperator (LexicalType type) {
        return OPERATORS.containsKey(type);
    }

    public static BinExprNode getHandler (LexicalType operator) throws Exception {
        if (!isOperator(operator)) {
            throw new Exception("Syntax Error");
        } else {
            return new BinExprNode(operator);
        }
    }

    public void setLeft (Node left) {
        this.left = left;
    }
```

```
    public void setRight (Node right) {
        this.right = right;
    }

    public int getOperatorPriority () {
        return OPERATORS.get(operator);
    }

    @Override
    public boolean parse () throws Exception {
        throw new Exception("Parsing Error");
    }

    @Override
    public String toString () {
        String opstr;
        switch (operator) {
        case DIV:
            opstr = "/";
            break;
        case MUL:
            opstr = "*";
            break;
        case SUB:
            opstr = "-";
            break;
        case ADD:
            opstr = "+";
            break;
        default:
            opstr = null;
        }
        return left + " " + right + " " + opstr;
    }

    @Override
    public Value getValue () throws Exception {
        if (left == null || right == null) {
            throw new Exception("Runtime Error: null operand found");
        }
        Value lval = left.getValue();
        Value rval = right.getValue();
        if (lval == null || rval == null) {
            throw new Exception("Runtime Error: null operand found");
        }
        double result;

        if (lval.getType() == ValueType.STRING || rval.getType() == ValueType.STRING)
{
            if (operator == LexicalType.ADD) {
                return new ValueImpl(lval.getSValue() + rval.getSValue());
```

```
            } else {
                throw new Exception("Runtime Error: invalid operator");
            }
        }

        switch (operator) {
        case DIV:
            if (rval.getDValue() != .0) {
                result = lval.getDValue() / rval.getDValue();
            } else {
                throw new Exception("Runtime Error: ZeroDivisionError: division by
zero");
            }
            break;
        case MUL:
            result = lval.getDValue() * rval.getDValue();
            break;
        case SUB:
            result = lval.getDValue() - rval.getDValue();
            break;
        case ADD:
            result = lval.getDValue() + rval.getDValue();
            break;
        default:
            throw new Exception("Runtime Error: invalid operator");
        }

        if (rval.getType() == ValueType.DOUBLE || lval.getType() == ValueType.DOUBLE)
{
            return new ValueImpl(result);
        } else {
            return new ValueImpl((int)result);
        }
    }
}
```

二項演算子式のノードである。左の項、右の項、演算子の情報を保持する。11行目で定義したOPERATORSに二項演算子を優先度と共に保持する。122行目では逆ポーランド記法の文字列を返すように記述している。

ソースコード3 BlockNode.java

```
package newlang4;

import java.util.EnumSet;
import java.util.Set;

/**
 *  <block> ::=
 *      <if_prefix> <stmt> <NL>
 *      | <if_prefix> <stmt> <ELSE> <stmt> <ENDIF> <NL>
 *      | <if_prefix> <NL> <stmt_list> <else_block> <ENDIF> <NL>
 *      | <WHILE> <cond> <NL> <stmt_list> <WEND> <NL>
 *      | <DO> <WHILE> <cond> <NL> <stmt_list> <LOOP> <NL>
```

```
 * !    | <DO> <UNTIL> <cond> <NL> <stmt_list> <LOOP> <NL>
 *      | <DO> <NL> <stmt_list> <LOOP> <WHILE> <cond> <NL>
 *      | <DO> <NL> <stmt_list> <LOOP> <UNTIL> <cond> <NL>
 */

public class BlockNode extends Node {
    boolean isIf = false;
    boolean isWhile = false;
    boolean isDoWhile = false;
    boolean isDoUntil = false;
    boolean isDo_While = false;
    boolean isDo_Until = false;

    Node cond = null;
    Node body = null;

    private final static Set<LexicalType> FIRST_SET = EnumSet.of(
        LexicalType.WHILE,
        LexicalType.DO
    );

    private BlockNode (Environment env) {
        super(env);
        type = NodeType.BLOCK;
    }

    public static boolean isFirst (LexicalUnit unit) {
        return FIRST_SET.contains(unit.getType());
    }

    public static Node getHandler (LexicalUnit first, Environment env) throws
Exception {
        if (!isFirst(first)) {
            throw new Exception("Syntax Error");
        } else {
            return new BlockNode(env);
        }
    }

    public boolean parse () throws Exception {
        LexicalUnit unit = null;
        if (env.getInput().peek().getType() == LexicalType.DO) {
            if (env.getInput().peek(2).getType() == LexicalType.UNTIL) {
                isDoUntil = true;
                env.getInput().get(2);
                unit = env.getInput().peek();
                cond = CondNode.getHandler(unit, env);
                cond.parse();
                unit = env.getInput().peek();
                body = StmtListNode.getHandler(unit, env);
                body.parse();
                if (env.getInput().peek().getType() != LexicalType.LOOP) {
```

```
                    throw new Exception("Syntax Error");
                } else {
                    env.getInput().get();
                }
                if (env.getInput().peek().getType() != LexicalType.NL) {
                    throw new Exception("Syntax Error");
                } else {
                    env.getInput().get();
                }
            }
        }
        return false;
    }

    @Override
    public String toString () {
        if (isIf) {
            return null;
        }
        if (isWhile) {
            return null;
        }
        if (isDoWhile) {
            return null;
        }
        if (isDoUntil) {
            return "DO UNTIL " + cond + "\n" + body + "LOOP";
        }
        if (isDo_While) {
            return null;
        }
        if (isDo_Until) {
            return null;
        }
        return null;
    }

    public Value getValue () throws Exception {
        if (isIf) {
            return null;
        }
        if (isWhile) {
            return null;
        }
        if (isDoWhile) {
            return null;
        }
        if (isDoUntil) {
            while (judge()) {
                body.getValue();
            }
            return null;
```

```
        }
        if (isDo_While) {
            return null;
        }
        if (isDo_Until) {
            return null;
        }
        return null;
    }

    private boolean judge () throws Exception {
        if (cond.getValue().getBValue() == true) {
            return true;
        } else {
            return false;
        }
    }
}
```

Blockのノードである。if文、for文、while文、do-while文、do-until文を扱う。

ループ文ではcondとbodyを保持する。

ソースコード4 CallFuncNode.java

```
package newlang4;

import java.util.EnumSet;
import java.util.Set;
import java.util.StringJoiner;

public class CallFuncNode extends Node {
    Function function = null;
    ExprListNode arg = null;

    private final static Set<LexicalType> FIRST_SET = EnumSet.of(
        LexicalType.NAME
    );

    private CallFuncNode (Environment env) {
        super(env);
        type = NodeType.FUNCTION_CALL;
    }

    public static boolean isFirst (LexicalUnit unit) {
        return FIRST_SET.contains(unit.getType());
    }

    public static Node getHandler (LexicalUnit first, Environment env) throws
Exception {
        if (!isFirst(first)) {
            throw new Exception("Syntax Error");
        } else {
```

```
            return new CallFuncNode(env);
        }
    }

    public boolean parse () throws Exception {
        LexicalUnit unit = env.getInput().get();
        function = env.getFunction(unit.getValue().getSValue());
        env.getInput().get(); // LP
        unit = env.getInput().peek();
        arg = (ExprListNode)ExprListNode.getHandler(unit, env);
        arg.parse();
        env.getInput().get(); // RP
        return true;
    }

    @Override
    public String toString () {
        StringJoiner sj = new StringJoiner(", ");
        arg.args.stream().forEach(a -> sj.add(a.toString()));
        return function + "(" + sj + ")";
    }

    public Value getValue () {
        return function.invoke(arg);
    }
}
```

関数呼び出しのノードである。

34行目でenvに登録されている関数から一致する関数を取得する処理を行う。

37行目で引数のExprListNodeのhandlerを取得し、38行目でparseを行っている。

ソースコード5 CallSubNode.java

```
package newlang4;

import java.util.EnumSet;
import java.util.Set;
import java.util.StringJoiner;

public class CallSubNode extends Node {
    Function function = null;
    ExprListNode arg = null;

    private final static Set<LexicalType> FIRST_SET = EnumSet.of(
        LexicalType.NAME
    );

    private CallSubNode (Environment env) {
        super(env);
        type = NodeType.FUNCTION_CALL;
    }
```

```
    public static boolean isFirst (LexicalUnit unit) {
        return FIRST_SET.contains(unit.getType());
    }

    public static Node getHandler (LexicalUnit first, Environment env) throws
Exception {
        if (!isFirst(first)) {
            throw new Exception("Syntax Error");
        } else {
            return new CallSubNode(env);
        }
    }

    public boolean parse () throws Exception {
        LexicalUnit unit = env.getInput().get();
        function = env.getFunction(unit.getValue().getSValue());
        unit = env.getInput().peek();
        arg = (ExprListNode)ExprListNode.getHandler(unit, env);
        arg.parse();
        return true;
    }

    @Override
    public String toString () {
        StringJoiner sj = new StringJoiner(", ");
        arg.args.stream().forEach(a -> sj.add(a.toString()));
        return function + " " + sj;
    }

    public Value getValue () {
        return function.invoke(arg);
    }
}
```

関数呼び出しのノードである。CallFuncNodeとはLP、RPの処理以外基本的に同じである。

ソースコード6 CondNode.java

```
package newlang4;

import java.util.EnumSet;
import java.util.Set;

public class CondNode extends Node {
    Node left = null;
    Node right = null;
    LexicalType operator = null;

    public final static Set<LexicalType> FIRST_SET = EnumSet.of(
        LexicalType.NAME,
        LexicalType.INTVAL,
        LexicalType.DOUBLEVAL,
        LexicalType.LITERAL,
```

```
        LexicalType.LP,
        LexicalType.SUB
    );

    private CondNode (Environment env) {
        super(env);
        type = NodeType.COND;
    }

    public static boolean isFirst (LexicalUnit unit) {
        return FIRST_SET.contains(unit.getType());
    }

    public static Node getHandler (LexicalUnit first, Environment env) throws
Exception {
        if (!isFirst(first)) {
            throw new Exception("Syntax Error");
        } else {
            return new CondNode(env);
        }
    }

    public boolean parse () throws Exception {
        LexicalUnit unit = env.getInput().peek();
        left = ExprNode.getHandler(unit, env);
        left.parse();

        operator = env.getInput().get().getType();

        unit = env.getInput().peek();
        right = ExprNode.getHandler(unit, env);
        right.parse();
        return true;
    }

    public String toString () {
        String opstr;
        switch (operator) {
        case EQ:
            opstr = "=";
            break;
        case GT:
            opstr = ">";
            break;
        case LT:
            opstr = "<";
            break;
        case GE:
            opstr = ">=";
            break;
        case LE:
            opstr = "<=";
```

```
            break;
        case NE:
            opstr = "<>";
            break;
        default:
            opstr = null;
        }
        return left + " " + opstr + " " + right;
    }

    public Value getValue () throws Exception {
        if (left == null || right == null) {
            throw new Exception("Runtime Error: Null operand found");
        }
        Value lval = left.getValue();
        Value rval = right.getValue();
        if (lval == null || rval == null) {
            throw new Exception("Runtime Error: Null operand found");
        }
        if (lval.getType() == ValueType.STRING || rval.getType() == ValueType.STRING)
{
            switch (operator) {
            case EQ:
                return new ValueImpl(lval.getSValue().equals(rval.getSValue()));
            case NE:
                return new ValueImpl(lval.getSValue().equals(rval.getSValue()));
            default:
                throw new Exception("Runtime Error: Invalid operator for String
Cond");
            }
        }

        switch (operator) {
        case EQ:
            return new ValueImpl(lval.getDValue() == rval.getDValue());
        case NE:
            return new ValueImpl(lval.getDValue() != rval.getDValue());
        case GT:
            return new ValueImpl(lval.getDValue() > rval.getDValue());
        case GE:
            return new ValueImpl(lval.getDValue() >= rval.getDValue());
        case LT:
            return new ValueImpl(lval.getDValue() < rval.getDValue());
        case LE:
            return new ValueImpl(lval.getDValue() <= rval.getDValue());
        default:
            throw new Exception("Runtime Error: Invalid operator for Cond");
        }
    }
}
```

条件式のノードである。左辺、右辺、演算子の情報を保持する。

39行目で左辺、45行目で右辺のhandlerを取得して保持している。

ソースコード7 ConstNode.java

```
package newlang4;

import java.util.EnumSet;
import java.util.Set;

public class ConstNode extends Node {
    Value value = null;

    public static final Set<LexicalType> FIRST_SET = EnumSet.of(
        LexicalType.INTVAL,
        LexicalType.DOUBLEVAL,
        LexicalType.LITERAL
    );

    private ConstNode (LexicalUnit first, Environment env) throws Exception {
        super(env);
        switch (first.getType()) {
        case INTVAL:
            type = NodeType.INT_CONSTANT;
            break;
        case DOUBLEVAL:
            type = NodeType.DOUBLE_CONSTANT;
            break;
        case LITERAL:
            type = NodeType.STRING_CONSTANT;
            break;
        default:
            throw new Exception("Syntax Error: Invalid input for ConstNode");
        }
        value = first.getValue();
    }

    public static boolean isFirst (LexicalUnit unit) {
        return FIRST_SET.contains(unit.getType());
    }

    public static Node getHandler (LexicalUnit first, Environment env) throws
Exception {
        if (!isFirst(first)) {
            throw new Exception("Syntax Error");
        } else {
            return new ConstNode(first, env);
        }
    }

    @Override
    public boolean parse () throws Exception {
        throw new Exception ("Parsing Error");
    }
```

```
    @Override
    public Value getValue () {
        return value;
    }

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

定数ノードである。整数、浮動小数点数、文字列定数を扱う。

valueを保持しそれを返す機能のみを持つ。

ソースコード8 EndNode.java

```
package newlang4;

import java.util.EnumSet;
import java.util.Set;

public class EndNode extends Node {
    public static final Set<LexicalType> FIRST_SET = EnumSet.of(
        LexicalType.END
    );

    private EndNode (Environment env) {
        super(env);
        type = NodeType.END;
    }

    public static boolean isFirst (LexicalUnit unit) {
        return FIRST_SET.contains(unit.getType());
    }

    public static Node getHandler (LexicalUnit first, Environment env) throws
Exception {
        if (!isFirst(first)) {
            throw new Exception("Syntax Error: Invalid token");
        } else {
            return new EndNode(env);
        }
    }

    @Override
    public boolean parse () throws Exception {
        if (env.getInput().expect(LexicalType.END)) {
            env.getInput().get();
            return true;
        } else {
            throw new Exception("Parse Error");
```

```
        }
    }

    @Override
    public String toString () {
        return "End";
    }
}
```

ENDノードである。

ソースコード9 Environment.java

```
package newlang4;

import java.util.Hashtable;
import java.util.Map;

public class Environment {
       LexicalAnalyzer input;
       Map<String, Function> library;
       Map<String, VariableNode> var_table;

        public Environment(LexicalAnalyzer my_input) {
            input = my_input;
            library = new Hashtable<>();
            library.put("PRINT", new PrintFunc());
            var_table = new Hashtable<>();
        }

        public LexicalAnalyzer getInput() {
            return input;
        }
        public Function getFunction(String fname) {
            return (Function) library.get(fname);
        }

        public VariableNode getVariable(String vname) {
            VariableNode v;
            v = (VariableNode) var_table.get(vname);
            if (v == null) {
                v = new VariableNode(vname);
                var_table.put(vname, v);
            }
            return v;
        }
}
```

SyntaxAnalyzer全体で使用するクラスである。LexicalAnalyzerを保持し、getやungetする際はこのクラスのLexicalAnalyzerを使用する。

また、FunctionやVariableの定義を持つ。

ソースコード10 ExprListNode.java

```
package newlang4;

import java.util.Set;
import java.util.List;
import java.util.ArrayList;

public class ExprListNode extends Node {
    List<Node> args = new ArrayList<>();

    private final static Set<LexicalType> FIRST_SET = ExprNode.FIRST_SET;

    private ExprListNode (Environment env) {
        super(env);
        type = NodeType.EXPR_LIST;
    }

    public static boolean isFirst (LexicalUnit unit) {
        return FIRST_SET.contains(unit.getType());
    }

    public static Node getHandler (LexicalUnit first, Environment env) throws
Exception {
        if (!isFirst(first)) {
            throw new Exception("Syntax Error");
        } else {
            return new ExprListNode(env);
        }
    }

    public boolean parse () throws Exception {
        LexicalUnit unit;
        while (true) {
            unit = env.getInput().peek();
            Node tmp = ExprNode.getHandler(unit, env);
            tmp.parse();
            args.add(tmp);
            unit = env.getInput().peek();
            if (unit.getType() == LexicalType.COMMA) {
                env.getInput().get(); // ,
            } else {
                break;
            }
        }
        return false;
    }

    @Override
    public String toString () {
        return null;
    }

    public Value getValue () {
```

```
        return null;
    }
}
```

演算子式リストノードである。

31行目からのwhile文でカンマ区切りの演算子式をArrayListに格納している。

ソースコード11 ExprNode.java

```
package newlang4;

import java.util.EnumSet;
import java.util.Set;
import java.util.List;
import java.util.ArrayList;
import java.util.Map;
import java.util.HashMap;

public class ExprNode extends Node {
    List<Node> operandNodes = new ArrayList<Node>();
    List<BinExprNode> binExprNodes = new ArrayList<BinExprNode>();
    public Node primedNode = null;
    Node value = null;

    public static final Set<LexicalType> FIRST_SET = EnumSet.of(
        LexicalType.NAME,
        LexicalType.INTVAL,
        LexicalType.DOUBLEVAL,
        LexicalType.LITERAL,
        LexicalType.LP,
        LexicalType.SUB
    );

    private static final Map<LexicalType, Integer> OPERATORS = new
HashMap<LexicalType, Integer>();

    static {
        OPERATORS.put(LexicalType.DIV, 1);
        OPERATORS.put(LexicalType.MUL, 1);
        OPERATORS.put(LexicalType.SUB, 1);
        OPERATORS.put(LexicalType.ADD, 2);
    }

    private ExprNode (Environment env) {
        super(env);
        type = NodeType.ASSIGN_STMT;
    }

    public static boolean isFirst (LexicalUnit unit) {
        return FIRST_SET.contains(unit.getType());
    }
```

```
    public static Node getHandler (LexicalUnit first, Environment env) throws
Exception {
        if (!isFirst(first)) {
            throw new Exception("Syntax Error: Invalid token");
        } else {
            return new ExprNode(env);
        }
    }

    @Override
    public boolean parse () throws Exception {
        if (!OPERATORS.containsKey(env.getInput().peek(2).getType())) {
            if (ConstNode.isFirst(env.getInput().peek())) {
                value = ConstNode.getHandler(env.getInput().get(), env);
            } else {
                value = VariableNode.getHandler(env.getInput().peek().getType(), env);
                env.getInput().get();
            }
            return true;
        }
        while (true) {
            Node operand = getOperand();
            operandNodes.add(operand);

            BinExprNode operator = getOperator();
            if (operator == null) {
                break;
            }
            binExprNodes.add(operator);
        }
        if (operandNodes.size() == 1 && binExprNodes.size() == 0) {
            primedNode = operandNodes.get(0);
        } else {
            primedNode = setBinExprNodes();
        }
        return true;
    }

    private Node getOperand () throws Exception {
        switch (env.getInput().peek().getType()) {
        case LP:
            env.getInput().get();
            Node exprHandler = ExprNode.getHandler(env.getInput().peek(), env);
            exprHandler.parse();
            if (env.getInput().expect(LexicalType.RP)) {
                env.getInput().get();
            } else {
                throw new Exception("Syntax Error: Too many '('");
            }
            return exprHandler;
        case SUB:
            LexicalUnit peeked2 = env.getInput().peek(2);
```

```
            if (ExprNode.isFirst(peeked2)
            ||  !(peeked2.getType() == LexicalType.SUB)
            ||  !(peeked2.getType() == LexicalType.LITERAL)) {
                env.getInput().get();
                LexicalUnit unit = new LexicalUnit(LexicalType.INTVAL, new
ValueImpl(-1));
                operandNodes.add(ConstNode.getHandler(unit, env));
                binExprNodes.add(BinExprNode.getHandler(LexicalType.MUL));
                return getOperand();
            } else {
                throw new Exception("Syntax Error");
            }
        case INTVAL:
        case DOUBLEVAL:
        case LITERAL:
            return ConstNode.getHandler(env.getInput().get(), env);
        case NAME:
            if (env.getInput().expect(LexicalType.LP, 2)) {
                // TODO
                // Node handler = CallFuncNode.getHandler(env.getInput().peek(), env);
                // handler.parse();
                // return handler;
            } else {
                Node handler =
VariableNode.getHandler(env.getInput().peek().getType(), env);
                env.getInput().get();
                return handler;
            }
        default:
            throw new Exception("Syntax Error");
        }
    }

    private BinExprNode getOperator () throws Exception {
        LexicalUnit peeked = env.getInput().peek();
        if (!OPERATORS.containsKey(peeked.getType())) {
            return null;
        }
        return BinExprNode.getHandler(env.getInput().get().getType());
    }

    private BinExprNode setBinExprNodes () throws Exception {
        if (operandNodes.size() < 2 || binExprNodes.size() < 1) {
            throw new Exception("Parsing Error");
        }
        boolean hasPending = false;
        int pendingID = 999;
        int priorityPrev, priorityNow;
        binExprNodes.get(0).setLeft(operandNodes.get(0));
        for (int i = 1; i < operandNodes.size() - 1; i++) {
            priorityPrev = binExprNodes.get(i - 1).getOperatorPriority();
            priorityNow = binExprNodes.get(i).getOperatorPriority();
```

```
            if (priorityPrev == priorityNow) {
                binExprNodes.get(i - 1).setRight(operandNodes.get(i));
                binExprNodes.get(i).setLeft(binExprNodes.get(i - 1));
            } else if (priorityPrev > priorityNow) {
                hasPending = true;
                pendingID = i - 1;
                binExprNodes.get(i).setLeft(operandNodes.get(i));
            } else {
                binExprNodes.get(i - 1).setRight(binExprNodes.get(i));
                if (!hasPending) {
                    binExprNodes.get(i).setLeft(binExprNodes.get(i - 1));
                } else {
                    binExprNodes.get(pendingID).setRight(binExprNodes.get(i - 1));
                    binExprNodes.get(i).setLeft(binExprNodes.get(pendingID));
                    hasPending = false;
                }
            }
        }
        BinExprNode lastBinExpr = binExprNodes.get(operandNodes.size() - 2);
        Node lastOperand = operandNodes.get(operandNodes.size() - 1);
        lastBinExpr.setRight(lastOperand);

        if (hasPending) {
            binExprNodes.get(pendingID).setRight(lastBinExpr);
            return binExprNodes.get(pendingID);
        }
        return lastBinExpr;
    }

    @Override
    public String toString () {
        if (primedNode != null) return primedNode.toString();
        return value.toString();
    }

    @Override
    public Value getValue () throws Exception {
        if (primedNode != null) {
            return primedNode.getValue();
        } else {
            return value.getValue();
        }
    }
}
```

演算子式ノードである。括弧や演算子の優先順位を考慮して式を解析する。53行目からのif文ではトークンが一つのみの場合の処理を行っている。62行目からのwhile文では2項以上の演算子式を解析する処理を行っている。76行目からのgetOperandメソッドでは演算子の左の項を取得する処理を記述している。単項演算子や括弧の条件分岐も行っている。121行目からのgetOperatorメソッドでは演算子を取得し、BinExprNodeのhandlerを返している。129行目からのsetBinExprNodesメソッドでは括弧や演算子の優先順位を考慮してbinExprNodesに入っている演算子式を組み立て、最終的な式を完成させる処理を行っている。この解析結果はtoStringメソッドを呼ぶことで確認できる。(逆ポーランド記法)

ソースコード12 Function.java

```
package newlang4;

public class Function {
    public Function () {

    }
    public Value invoke (ExprListNode arg) {
        return null;
    }
}
```

関数の親クラスである。

ソースコード13 LexicalAnalyzer.java

```
package newlang4;

public interface LexicalAnalyzer {
    public LexicalUnit get() throws Exception;
    public LexicalUnit get(int n) throws Exception;
    public boolean expect(LexicalType type) throws Exception;
    public boolean expect(LexicalType type, int n) throws Exception;
    public void unget(LexicalUnit token) throws Exception;
    public LexicalUnit peek() throws Exception;
    public LexicalUnit peek(int n) throws Exception;
}
```

LexicalAnalyzerImplのインタフェース定義である。

ソースコード14 LexicalAnalyzerImpl.java

```
package newlang4;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.PushbackReader;
import java.util.HashMap;
import java.util.Stack;

public class LexicalAnalyzerImpl implements LexicalAnalyzer {
    PushbackReader reader;

    static HashMap<String, LexicalUnit> reservedWords = new HashMap<String,
LexicalUnit>();
    static HashMap<String, LexicalUnit> punctuators = new HashMap<String, LexicalUnit>
();

    static {
        reservedWords.put("if", new LexicalUnit(LexicalType.IF));
        reservedWords.put("then", new LexicalUnit(LexicalType.THEN));
        reservedWords.put("else", new LexicalUnit(LexicalType.ELSE));
```

```
        reservedWords.put("elseif", new LexicalUnit(LexicalType.ELSEIF));
        reservedWords.put("endif", new LexicalUnit(LexicalType.ENDIF));
        reservedWords.put("for", new LexicalUnit(LexicalType.FOR));
        reservedWords.put("forall", new LexicalUnit(LexicalType.FORALL));
        reservedWords.put("next", new LexicalUnit(LexicalType.NEXT));
        reservedWords.put("sub", new LexicalUnit(LexicalType.SUB));
        reservedWords.put("dim", new LexicalUnit(LexicalType.DIM));
        reservedWords.put("as", new LexicalUnit(LexicalType.AS));
        reservedWords.put("end", new LexicalUnit(LexicalType.END));
        reservedWords.put("while", new LexicalUnit(LexicalType.WHILE));
        reservedWords.put("do", new LexicalUnit(LexicalType.DO));
        reservedWords.put("until", new LexicalUnit(LexicalType.UNTIL));
        reservedWords.put("loop", new LexicalUnit(LexicalType.LOOP));
        reservedWords.put("to", new LexicalUnit(LexicalType.TO));
        reservedWords.put("wend", new LexicalUnit(LexicalType.WEND));

        punctuators.put("=", new LexicalUnit(LexicalType.EQ));
        punctuators.put("<", new LexicalUnit(LexicalType.LT));
        punctuators.put(">", new LexicalUnit(LexicalType.GT));
        punctuators.put("<=", new LexicalUnit(LexicalType.LE));
        punctuators.put("=<", new LexicalUnit(LexicalType.LE));
        punctuators.put(">=", new LexicalUnit(LexicalType.GE));
        punctuators.put("=>", new LexicalUnit(LexicalType.GE));
        punctuators.put("<>", new LexicalUnit(LexicalType.NE));
        punctuators.put("\n", new LexicalUnit(LexicalType.NL));
        punctuators.put(".", new LexicalUnit(LexicalType.DOT));
        punctuators.put("+", new LexicalUnit(LexicalType.ADD));
        punctuators.put("-", new LexicalUnit(LexicalType.SUB));
        punctuators.put("*", new LexicalUnit(LexicalType.MUL));
        punctuators.put("/", new LexicalUnit(LexicalType.DIV));
        punctuators.put("(", new LexicalUnit(LexicalType.LP));
        punctuators.put(")", new LexicalUnit(LexicalType.RP));
        punctuators.put(",", new LexicalUnit(LexicalType.COMMA));
    }

    private Stack<LexicalUnit> ungetStack = new Stack<>();

    public LexicalAnalyzerImpl (String fileName) throws FileNotFoundException {
        reader = new PushbackReader(new BufferedReader(new FileReader(fileName)));
    }

    @Override
    public LexicalUnit get() throws Exception {
        if (!ungetStack.isEmpty()) {
            return ungetStack.pop();
        }

        char ch = (char)reader.read();

        char EOF = (char)-1;

        while (ch == ' ' || ch == '\t' || ch == '\r') {
```

```
            ch = (char)reader.read();
            if (ch == EOF) {
                return new LexicalUnit(LexicalType.EOF);
            }
        }

        if (ch == EOF) {
            return new LexicalUnit(LexicalType.EOF);
        }

        String input = String.valueOf(ch);

        reader.unread(ch);

        if (input.matches("[0-9]")) {
            return getNumber();
        }
        if (input.matches("[a-zA-Z_]")) {
            return getString();
        }
        if (input.matches("\"")) {
            return getLiteral();
        }
        return getPunctuator();
    }

    @Override
    public LexicalUnit get (int n) throws Exception {
        LexicalUnit unit = null;
        for (int i = 0; i < n; i++) {
            unit = get();
        }
        return unit;
    }

    public boolean findUnit () {
        return false;
    }

    @Override
    public boolean expect(LexicalType type) throws Exception {
        return  peek().getType() == type;
    }

    @Override
    public boolean expect(LexicalType type, int n) throws Exception {
        return peek(n).getType() == type;
    }

    @Override
    public void unget(LexicalUnit token) throws Exception {
        ungetStack.push(token);
```

```
    }

    private LexicalUnit getNumber () throws IOException {
        String target = "";

        char ch;

        boolean isDouble = false;

        while (true) {
            ch = (char)reader.read();
            if (ch < 0) break;
            if ((target + ch).matches("[0-9]+(\\.[0-9]*)?")) {
                target += ch;
                if (target.matches("[0-9]+\\.[0-9]+")) {
                    isDouble = true;
                }
                continue;
            }
            reader.unread(ch);
            break;
        }

        return isDouble
        ? new LexicalUnit(
            LexicalType.DOUBLEVAL,
            new ValueImpl(target, ValueType.DOUBLE)
        )
        : new LexicalUnit(
            LexicalType.INTVAL,
            new ValueImpl(target, ValueType.INTEGER)
        );
    }

    private LexicalUnit getString () throws IOException {
        String target = "";

        char ch;

        while (true) {
            ch = (char)reader.read();
            if (ch < 0) break;

            if ((target + ch).matches("[a-zA-Z_][a-zA-Z0-9_]*")) {
                target += ch;
                continue;
            }
            reader.unread(ch);
            break;
        }

        String key = target.toLowerCase();
```

```
        LexicalUnit lu = reservedWords.get(key);

        if (lu != null) {
            return lu;
        }

        return new LexicalUnit(
            LexicalType.NAME,
            new ValueImpl(target, ValueType.STRING)
        );
    }

    private LexicalUnit getLiteral () throws IOException {
        String target = "";

        char ch;

        reader.read();

        while (true) {
            ch = (char)reader.read();
            if (ch < 0) break;
            if (ch != '"') {
                target += ch;
                continue;
            }
            break;
        }

        return new LexicalUnit(
            LexicalType.LITERAL,
            new ValueImpl(target)
        );
    }

    private LexicalUnit getPunctuator () throws IOException {
        String target = "";

        char ch;
        String str;
        LexicalUnit prev = null;

        while (true) {
            ch = (char)reader.read();
            str = String.valueOf(ch);
            if (ch < 0) break;
            if (str.matches("=|<|>|\\+|-|\\*|/|\\(|\\)|,|\\n")) {
                target += str;
                if (punctuators.get(target) != null || prev == null) {
                    prev = punctuators.get(target);
                    continue;
```

```
                }
            }
            reader.unread(ch);
            break;
        }
        return prev;
    }

    @Override
    public LexicalUnit peek () throws Exception {
        LexicalUnit res = get();
        unget(res);
        return res;
    }

    public LexicalUnit peek (int n) throws Exception {
        Stack<LexicalUnit> peekStack = new Stack<>();
        LexicalUnit res = null;
        for (int i = 0; i < n; i++) {
            peekStack.add(res = get());
        }
        while (!peekStack.empty()) {
            unget(peekStack.pop());
        }
        return res;
    }
}
```

LexicalAnalyzerImplクラスである。詳細は前回のレポートで解説済みである。get、peek、expectにそれぞれn個先のトークンについての処理を行うオーバーロードメソッドの定義を追加している。

ソースコード15 LexicalType.java

```
package newlang4;

public enum LexicalType {
    LITERAL,    // 文字列定数  (例:  〝文字列〞)
    INTVAL,       // 整数定数    (例:  3)
    DOUBLEVAL,   // 小数点定数    (例:  1.2)
    NAME,       // 変数        (例:  i)
    IF,           // IF
    THEN,       // THEN
    ELSE,       // ELSE
    ELSEIF,       // ELSEIF
    ENDIF,       // ENDIF
    FOR,       // FOR
    FORALL,       // FORALL
    NEXT,       // NEXT
    EQ,           // =
    LT,           // <
    GT,           // >
    LE,           // <=, =<
    GE,           // >=, =>
```

```
    NE,             // <>
    DIM,         // DIM
    AS,             // AS
    END,         // END
    NL,             // 改行
    DOT,         // .
    WHILE,         // WHILE
    DO,             // DO
    UNTIL,         // UNTIL
    ADD,         // +
    SUB,         // -
    MUL,         // *
    DIV,         // /
    LP,             // )
    RP,             // (
    COMMA,         // ,
    LOOP,         // LOOP
    TO,             // TO
    WEND,         // WEND
    EOF,         // end of file
}
```

LexicalTypeの定義である。

ソースコード16 LexicalUnit.java

```
package newlang4;


public class LexicalUnit {
    LexicalType type;
    Value value;
    LexicalUnit link;

    public LexicalUnit(LexicalType this_type) {
        type = this_type;
    }

    public LexicalUnit(LexicalType this_type, Value this_value) {
        type = this_type;
        value = this_value;
    }

    public Value getValue() {
        return value;
    }

    public LexicalType getType() {
        return type;
    }

    public String toString() {
        switch(type) {
```

```
        case LITERAL:
            return "LITERAL:\t" + value.getSValue();
        case NAME:
            return "NAME:\t" + value.getSValue();
        case DOUBLEVAL:
            return "DOUBLEVAL:\t" + value.getSValue();
        case INTVAL:
            return "INTVAL:\t" + value.getSValue();
        case IF:
            return ("IF");
        case THEN:
            return ("THEN");
        case ELSE:
            return ("ELSE");
        case FOR:
            return ("FOR");
        case FORALL:
            return ("FORALL");
        case NEXT:
            return ("NEXT");
        case SUB:
            return ("SUB");
        case DIM:
            return ("DIM");
        case AS:
            return ("AS");
        case END:
            return ("END");
        case EOF:
            return ("EOF");
        case NL:
            return ("NL");
        case EQ:
            return ("EQ");
        case LT:
            return ("LT");
        case GT:
            return ("GT");
        case LE:
            return ("LE");
        case GE:
            return ("GE");
        case DOT:
            return ("DOT");
        case WHILE:
            return ("WHILE");
        case DO:
            return ("DO");
        case UNTIL:
            return ("UNTIL");
        case ADD:
            return ("ADD");
```

```
        case MUL:
            return ("MUL");
        case DIV:
            return ("DIV");
        case LP:
            return ("LP");
        case RP:
            return ("RP");
        case COMMA:
            return ("COMMA");
        case LOOP:
            return ("LOOP");
        case TO:
            return ("TO");
        case WEND:
            return ("WEND");
        case ELSEIF:
            return ("ELSEIF");
        case NE:
            return ("NE");
        case ENDIF:
            return ("ENDIF");
        }
        return "";
    }
}
```

LexicalUnitの定義である。変更点はない。

ソースコード17 Node.java

```
package newlang4;

public class Node {
    NodeType type;
    Environment env;

    /** Creates a new instance of Node */
    public Node() {
    }
    public Node(NodeType my_type) {
        type = my_type;
    }
    public Node(Environment my_env) {
        env = my_env;
    }

    public NodeType getType() {
        return type;
    }

    public boolean parse() throws Exception {
        return true;
```

```
    }

    public Value getValue() throws Exception {
        return null;
    }

    public String toString() {
        if (type == NodeType.END) return "END";
        else return "Node";
    }
}
```

すべてのノードの親クラスである。NodeTypeとEnvironmentを保持している。

ソースコード18 NodeType.java

```
package newlang4;

public enum NodeType {
    PROGRAM,
    STMT_LIST,
    STMT,
    FOR_STMT,
    ASSIGN_STMT,
    BLOCK,
    IF_BLOCK,
    LOOP_BLOCK,
    COND,
    EXPR_LIST,
    EXPR,
    BIN_EXPR,
    FUNCTION_CALL,
    STRING_CONSTANT,
    INT_CONSTANT,
    DOUBLE_CONSTANT,
    BOOL_CONSTANT,
    END,
}
```

Nodeの種類の定義である。実装の都合上定義を増やしている。

ソースコード19 PrintFunc.java

```
package newlang4;

public class PrintFunc extends Function {
    @Override
    public Value invoke (ExprListNode arg) {
        for (Node a : arg.args) {
            System.out.println(a.toString());
        }
        return null;
    }
```

```
    @Override
    public String toString () {
        return "PRINT";
    }
}
```

PRINT関数の定義である。構文解析の時点では動作する必要はないため機能は実装していない。

ソースコード20 Program.java

```
package newlang4;

import java.util.EnumSet;
import java.util.Set;

public class Program extends Node {
    Node stmtList = null;

    private static final Set<LexicalType> FIRST_SET = EnumSet.of(
        LexicalType.NAME,
        LexicalType.FOR,
        LexicalType.END,
        LexicalType.IF,
        LexicalType.WHILE,
        LexicalType.DO,
        LexicalType.NL
    );

    private Program (Environment env) {
        super(env);
        type = NodeType.PROGRAM;
    }

    public static boolean isFirst (LexicalUnit unit) {
        return FIRST_SET.contains(unit.getType());
    }

    public static Node getHandler (LexicalUnit first, Environment env) throws
Exception {
        if (!isFirst(first)) {
            throw new Exception("Syntax Error: Invalid token");
        } else {
            return new Program(env);
        }
    }

    @Override
    public boolean parse () throws Exception {
        stmtList = StmtListNode.getHandler(env.getInput().peek(), env);
        stmtList.parse();

        while (env.getInput().expect(LexicalType.NL)) {
            env.getInput().get();
```

```
        }

        if (!env.getInput().expect(LexicalType.EOF)) {
            throw new Exception("Syntax Error: END token must be end of input");
        }

        return true;
    }

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

    public Value getValue() throws Exception {
        return stmtList.getValue();
    }
}
```

すべての親のノードである。このクラスのparseを呼ぶことでプログラム全体の構文解析が行われる。

Programの子ノードはStmtListNodeのみであるため37行目からのparse関数ではStmtListNodeのhandlerを取得する処理から始まる。もしStmtListのファースト集合にないトークンが読み込まれた場合はエラーを吐く。

ソースコード21 StmtListNode.java

```
package newlang4;

import java.util.ArrayList;
import java.util.EnumSet;
import java.util.List;
import java.util.Set;

public class StmtListNode extends Node {
    List<Node> nodes;

    private static final Set<LexicalType> FIRST_SET = EnumSet.of(
        LexicalType.NAME,
        LexicalType.FOR,
        LexicalType.END,
        LexicalType.IF,
        LexicalType.WHILE,
        LexicalType.DO,
        LexicalType.NL
    );

    private StmtListNode (Environment env) {
        super(env);
        type = NodeType.STMT_LIST;
    }

    public static boolean isFirst (LexicalUnit unit) {
        return FIRST_SET.contains(unit.getType());
```

```
    }

    public static Node getHandler (LexicalUnit first, Environment env) throws
Exception {
        if (!isFirst(first)) {
            throw new Exception("Syntax Error: Invalid token");
        } else {
            return new StmtListNode(env);
        }
    }

    @Override
    public boolean parse () throws Exception {
        nodes = new ArrayList<Node>();
        LexicalUnit peeked;
        Node handler = null;

        while (true) {
            while (env.getInput().expect(LexicalType.NL)) {
                env.getInput().get();
            }
            peeked = env.getInput().peek();

            if (StmtNode.isFirst(peeked)) {
                handler = StmtNode.getHandler(peeked, env);
            } else if (BlockNode.isFirst(peeked)) {
                handler = BlockNode.getHandler(peeked, env);
            } else if (peeked.getType() == LexicalType.LOOP) {
                return true;
            } else {
                throw new Exception("Syntax Error");
            }
            handler.parse();
            nodes.add(handler);

            if (handler.getType() == NodeType.END) {
                return true;
            }
        }
    }

    @Override
    public String toString () {
        String res = "";
        for (Node node : nodes) {
            res += node.toString() + "\n";
        }
        return res;
    }

    @Override
    public Value getValue() throws Exception {
```

```
        Value res = null;
        for (Node node : nodes) {
            res = node.getValue();
        }
        return res; // return last value
    }
}
```

StmtListのノードである。9行目のnodesに子ノードを持つ。50行目からのif文ではStmtとBlockの分岐を行いそれぞれhandlerを取得している。59行目のparseにより子ノードの解析が行われる。handlerがENDだった場合にtrueを返す。

ソースコード22 StmtNode.java

```
package newlang4;

import java.util.EnumSet;
import java.util.Set;

public class StmtNode extends Node {
    private static Node node = null;

    private static final Set<LexicalType> FIRST_SET = EnumSet.of(
        LexicalType.NAME,
        LexicalType.FOR,
        LexicalType.END
    );

    public static boolean isFirst (LexicalUnit unit) {
        return FIRST_SET.contains(unit.getType());
    }

    private StmtNode (Environment env) {
        super(env);
        type = NodeType.STMT;
    }

    public static Node getHandler (LexicalUnit first, Environment env) throws
Exception {
        switch (first.getType()) {
        case NAME:
            switch (env.getInput().peek(2).getType()) {
            case EQ:
                return node = SubstNode.getHandler(first, env);
            case LP:
                return node = CallFuncNode.getHandler(first, env);
            default:
            }
            if (ExprListNode.isFirst(env.getInput().peek(2))) {
                return node = CallSubNode.getHandler(first, env);
            } else {
                throw new Exception("Syntax Error");
            }
//        case FOR:
```

```
        case END:
            return node = EndNode.getHandler(first, env);
        default:
            throw new Exception("Syntax Error: Invalid token");
        }
    }

    @Override
    public boolean parse () throws Exception {
        throw new Exception("Parse Error");
    }

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

    @Override
    public Value getValue () throws Exception {
        return node.getValue();
    }
}
```

Stmtノードである。Stmtの子ノードであるSubstNodeとCallFuncNodeはどちらもNAMEから開始するため、23行目、25行目のswitch文で分岐処理を行っている。

ソースコード23 SubstNode.java

```
package newlang4;

import java.util.EnumSet;
import java.util.Set;

public class SubstNode extends Node {
    public static final Set<LexicalType> FIRST_SET = EnumSet.of(
        LexicalType.NAME
    );

    private Node variable = null;
    private Node expression = null;

    private SubstNode (Environment env) {
        super(env);
        type = NodeType.ASSIGN_STMT;
    }

    public static boolean isFirst (LexicalUnit unit) {
        return FIRST_SET.contains(unit.getType());
    }

    public static Node getHandler (LexicalUnit first, Environment env) throws
Exception {
        if (!isFirst(first)) {
```

```
            throw new Exception("Syntax Error: Invalid token");
        } else {
            return new SubstNode(env);
        }
    }

    @Override
    public boolean parse () throws Exception {
        LexicalUnit unit = null;
        if (env.getInput().expect(LexicalType.NAME)) {
            unit = env.getInput().peek();
            variable = VariableNode.getHandler(unit.getType(), env);
        } else {
            throw new Exception("Parsing Error: Unexpected token: " +
env.getInput().get());
        }
        env.getInput().get();
        if (env.getInput().expect(LexicalType.EQ)) {
            unit = env.getInput().get();
        } else {
            throw new Exception("Parsing Error: Unexpected token: " +
env.getInput().get());
        }
        if (ExprNode.isFirst(env.getInput().peek())) {
            unit = env.getInput().peek();
            expression = ExprNode.getHandler(unit, env);
            expression.parse();
            return true;
        } else {
            throw new Exception("Parsing Error: Unexpcted token: " +
env.getInput().get());
        }
    }

    @Override
    public String toString () {
        return variable + " = " + expression;
    }

    @Override
    public Value getValue () throws Exception {
        ((VariableNode)variable).setValue(expression.getValue());
        return variable.getValue();
    }
}
```

代入文のノードである。左辺に変数、右辺に演算子式を持つ。

ソースコード24 Value.java

```
package newlang4;

public abstract class Value {
```

```
// 実装すべきコンストラクタ
    public Value(String s) {};
    public Value(int i) {};
    public Value(double d) {};
    public Value(boolean b) {};
    public Value(String s, ValueType t) {};
    public abstract String get_sValue();
    public abstract String getSValue();
    // ストリング型で値を取り出す。必要があれば、型変換を行う。
    public abstract int getIValue();
        // 整数型で値を取り出す。必要があれば、型変換を行う。
    public abstract double getDValue();
        // 小数点型で値を取り出す。必要があれば、型変換を行う。
    public abstract boolean getBValue();
        // 論理型で値を取り出す。必要があれば、型変換を行う。
    public abstract ValueType getType();
}
```

ValueImplクラスの抽象クラスである。型変換を行うためのメソッドが定義されている。

ソースコード25 ValueImpl.java

```
package newlang4;

public class ValueImpl extends Value {

    private ValueType type;
    private String valString;
    private int valInteger;
    private double valDouble;
    private boolean valBoolean;

    public ValueImpl (String s) {
        super(s);
        valString = s;
        type = ValueType.STRING;
    }
    public ValueImpl (int i) {
        super(i);
        valInteger = i;
        type = ValueType.INTEGER;
    }
    public ValueImpl (double d) {
        super(d);
        valDouble = d;
        type = ValueType.DOUBLE;
    }
    public ValueImpl (boolean b) {
        super(b);
        valBoolean = b;
        type = ValueType.BOOL;
    }
    public ValueImpl (String s, ValueType t) {
```

```
        super(s, t);
        switch(t) {
        case STRING:
            valString = s;
            break;
        case INTEGER:
            valInteger = Integer.parseInt(s);
            break;
        case DOUBLE:
            valDouble = Double.parseDouble(s);
            break;
        case BOOL:
            valBoolean = Boolean.parseBoolean(s);
            break;
        case VOID:
            break;
        }
        type = t;
    }

    @Override
    public String get_sValue() {
        return getSValue();
    }

    @Override
    public String getSValue() {
        switch(type) {
        case STRING:
            return valString;
        case INTEGER:
            return String.valueOf(valInteger);
        case DOUBLE:
            return String.valueOf(valDouble);
        case BOOL:
            return String.valueOf(valBoolean);
        default:
            return null;
        }
    }

    @Override
    public int getIValue() {
        switch(type) {
        case STRING:
            return Integer.parseInt(valString);
        case INTEGER:
            return valInteger;
        case DOUBLE:
            return (int)valDouble;
        case BOOL:
            return valBoolean ? 1 : 0;
```

```
        default:
            return 0;
        }
    }

    @Override
    public double getDValue() {
        switch(type) {
        case STRING:
            return Double.parseDouble(valString);
        case INTEGER:
            return (double)valInteger;
        case DOUBLE:
            return valDouble;
        case BOOL:
            return valBoolean ? 1.0 : 0.0;
        default:
            return 0.0;
        }
    }

    @Override
    public boolean getBValue() {
        switch(type) {
        case STRING:
            return Boolean.parseBoolean(valString);
        case INTEGER:
            return (valInteger != 0) ? true : false;
        case DOUBLE:
            return (valDouble != 0) ? true : false;
        case BOOL:
            return valBoolean;
        default:
            return false;
        }
    }

    @Override
    public ValueType getType() {
        return type;
    }
}
```

値を保持するクラスである。型毎の値取得メソッドが定義されている。

ソースコード26 ValueType.java

```
package newlang4;

public enum ValueType {
    VOID,
    INTEGER,
    STRING,
```

```
    DOUBLE,
    BOOL,
}
```

値の型の定義である。

ソースコード27 VariableNode.java

```
package newlang4;

public class VariableNode extends Node {
    String var_name;
    Value v;

    /** Creates a new instance of variable */
    public VariableNode(String name) {
        var_name = name;
    }
    public VariableNode(LexicalUnit u) {
        var_name = u.getValue().getSValue();
    }

    public static boolean isMatch(LexicalType first) {
        return (first == LexicalType.NAME);
    }

    public static Node getHandler(LexicalType first, Environment my_env) {
        if (first == LexicalType.NAME) {
            VariableNode v;

            try {
                LexicalUnit lu = my_env.getInput().peek();
                String s = lu.getValue().getSValue();
                v = my_env.getVariable(s);
                return v;
            } catch(Exception e) {
                e.printStackTrace();
            }
        }
        return null;
    }

    public void setValue(Value my_v) {
        v = my_v;
    }

    public Value getValue() {
        return v;
    }

    public String toString() {
        String str = var_name;
        return str;
```

```
    }
}
```

変数ノードである。20行目ではLexicalTypeがNAMEであることを確認して解析を行っている。valueを保持し、getValueメソッドを呼ぶとそれを返す。

2. 実行例

ソースコード28 src.txt

```
PRINT ("** OUTPUT **")

a = 5

DO UNTIL a > 1

PRINT ("Hello")

a = a - 1

LOOP

END
```

ソースコード28は講義ページのソースプログラムの例を改変したものである。

実行結果1 実行結果

```
basic parser

PRINT(** OUTPUT **)
a = 5
DO UNTIL a > 1
PRINT(Hello)
a = a 1 -
LOOP
End

** OUTPUT **
Hello
Hello
Hello
Hello
value = null
```

パース結果の下にインタプリタの実行結果が表示されている。DO-UNTILループ、PRINT関数等が正常に実行されていることがわかる。