Национальный исследовательский университет ИТМО
Факультет информационных технологий и программирования
Прикладная математика и информатика

Методы трансляции

Отчёт по лабораторной работе № 3

«Использование автоматических генераторов анализаторов Bison и ANTLR»

Работу выполнил:

Ивченков Дмитрий Артемович, M33341

Санкт-Петербург

2023

Вариант 4: Функциональный язык.

Средство автоматической генерации: ANTLR

Разработанный примитивный функциональный язык программирования включает в себя функции и их объявление, выражения целочисленного и булевого типа, условное ветвление.

1 Разработка программы

Грамматика:

```
grammar Converter;

@header {
    import java.util.List;
    import java.util.ArrayList;
    import java.util.Map;
    import java.util.HashMap;
    import java.util.Set;
    import java.util.HashSet;
    import converter.exception.ConversionException;
}

@members {
    Map<String, List<String>> functions = new HashMap<>();
}

WHITESPACE : [ \t\r\n]+ -> skip;

fragment Letter : 'a'..'z' | 'A'..'Z';
fragment DigitStart : '1'..'9';
fragment Digit : '0' | DigitStart;

INT : ('+'|'-')?('0' | DigitStart(Digit)*);
BOOLEAN : 'True' | 'False';

NAME : NameStartChar (NameChar)*;
fragment NameStartChar : Letter | '_';
fragment NameChar : NameStartChar | Digit;

program : functionDeclaration+ EOF;

functionDeclaration
    locals[
        String name,
        List<String> argumentTypes = new ArrayList<>()
    ]
    :
    functionSignature { functions.put($name, $argumentTypes); }
    functionBody;

functionSignature
    :
    NAME '::' functionType
    {
        if (functions.containsKey($NAME.text)) {
            throw new ConversionException("Function already defined: " +
$NAME.text);
```

```
        }
        $functionDeclaration::name = $NAME.text;
    };

functionType : typeName ('->' typeName)*;

typeName
    :
    ('Int' | 'Bool')
    { $functionDeclaration::argumentTypes.add($ctx.getText()); }
    ;

functionBody : (functionCase)+;

functionCase
    locals[
        Map<String, Integer> variables = new HashMap<>(),
        Set<Integer> expressionNumbers = new HashSet<>(),
        int argumentNumber = 0
    ]
    :
    functionArgumentsCase (functionResult | functionGuards);

functionArgumentsCase
    :
    NAME {
        if (!$functionDeclaration::name.equals($NAME.text)) {
            throw new ConversionException("Wrong function name: " + $NAME.text);
        }
    }
    (functionArgument { $functionCase::argumentNumber++; })*
    {
        if ($functionCase::argumentNumber !=
$functionDeclaration::argumentTypes.size() - 1) {
            throw new ConversionException("Wrong arguments number for function "
+ $NAME.text);
        }
    };

functionArgument
    :
    NAME { $functionCase::variables.put($NAME.text,
$functionCase::argumentNumber); }
    |
expression[$functionDeclaration::argumentTypes.get($functionCase::argumentNumber
)]
    { $functionCase::expressionNumbers.add($functionCase::argumentNumber); };

functionResult
    :
    '='
    { List<String> argumentTypes = $functionDeclaration::argumentTypes; }
    expression[argumentTypes.get(argumentTypes.size() - 1)];

functionGuards : condition+;

condition : '|' boolExpression functionResult;

expression[String expectedType]
```

```
    :
    { $expectedType.equals("Int") }? mathExpression
    | { $expectedType.equals("Bool") }? boolExpression;
    catch[NoViableAltException e]
    { throw new ConversionException($expectedType + " expression expected: " +
$ctx.getText()); }

mathExpression
    :
    mathExpression sign=('+'|'-') summand
    | summand;

summand
    :
    summand sign=('*'|'/') factor
    | factor;

factor
    :
    INT
    | functionApplication["Int"]
    | variable["Int"]
    | (minus='-')? '(' mathExpression ')';

boolExpression
    :
    boolExpression '||' conjunction
    | conjunction;

conjunction
    :
    conjunction '&&' boolean
    | boolean;

boolean
    :
    BOOLEAN
    | functionApplication["Bool"]
    | variable["Bool"]
    | comparison
    | (inversion='!')? '(' boolExpression ')';

comparison : mathExpression sign=('<'|'<='|'>'|'>='|'=='|'!=') mathExpression;

variable[String expectedType]
    :
    NAME
    {
        Map<String, Integer> variables = $functionCase::variables;
        String name = $NAME.text;
        if (!variables.containsKey(name)) {
            throw new ConversionException("Unknown variable " + name + "
provided");
        }
        String type =
$functionDeclaration::argumentTypes.get(variables.get(name));
        if (!type.equals($expectedType)) {
            throw new ConversionException(
                String.format(
```

```
                    "Wrong variable %s type: %s expected but %s provided",
                    $NAME.text,
                    $expectedType,
                    type
                )
            );
        }
    };

functionApplication[String expectedType]
    :
    NAME {
        if (!functions.containsKey($NAME.text)) {
            throw new ConversionException("Unknown function expression: " +
$NAME.text);
        }
    }
    {
        List<String> argumentTypes = functions.get($NAME.text);
        int argumentNumber = 0;
    }
    '(' (expression[argumentTypes.get(argumentNumber++)]
            (',' expression[argumentTypes.get(argumentNumber++)])*
        )?
    ')'
    {
        List<String> functionTypes = functions.get($NAME.text);
        if ($ctx.expression().size() != functionTypes.size() - 1) {
            throw new ConversionException(
                String.format(
                    "Wrong arguments number for function %s: %s",
                     $NAME.text,
                     $ctx.getText()
                )
            );
        }
        String type = functionTypes.get(functionTypes.size() - 1);
        if (!type.equals($expectedType)) {
            throw new ConversionException(
                String.format(
                    "Wrong function application %s type: %s expected but %s
provided",
                    $NAME.text,
                    $expectedType,
                    type
                )
            );
        }
    };
```

ProgramVisitor:

```
package converter.visitor;

import converter.parser.ConverterBaseVisitor;
import converter.parser.ConverterParser;

import java.util.ArrayList;
```

```
import java.util.List;
import java.util.Map;

public class ProgramVisitor extends ConverterBaseVisitor<String> {
    private static final String LINE_SEPARATOR = System.lineSeparator();
    private static final String LINE_OFFSET_SEPARATOR = LINE_SEPARATOR + "    ";
    private static final String LINE_SKIP_SEPARATOR = LINE_SEPARATOR +
LINE_SEPARATOR + "    ";
    private static final Map<String, String> PRIMITIVE_TYPES = Map.of(
            "Bool", "boolean",
            "Int", "int",
            "Double", "double"
    );

    @Override
    public String visitProgram(converter.parser.ConverterParser.ProgramContext
ctx) {
        List<String> functions = new ArrayList<>();
        for (int i = 0; i < ctx.getChildCount() - 1; i++) {

functions.add(addOffset(visitFunctionDeclaration(ctx.functionDeclaration(i))));
        }
        return String.format(
                "public class Program {%n    %s%n}",
//                addOffset(
                String.join(LINE_SKIP_SEPARATOR, functions)
//                )
        );
    }

    @Override
    public String
visitFunctionDeclaration(ConverterParser.FunctionDeclarationContext ctx) {
        return String.format(
                "%s {%n    %s%n}",
                visitFunctionSignature(ctx.functionSignature()),
                addOffset(visitFunctionBody(ctx.functionBody()))
        );
    }

    @Override
    public String
visitFunctionSignature(ConverterParser.FunctionSignatureContext ctx) {
        ConverterParser.FunctionTypeContext functionTypeContext =
ctx.functionType();
        List<ConverterParser.TypeNameContext> types =
functionTypeContext.typeName();
        List<String> arguments = new ArrayList<>();
        for (int i = 0; i < types.size() - 1; i++) {
            arguments.add(String.format("%s arg%d", visitTypeName(types.get(i)),
i));
        }
        return String.format(
                "public %s %s(%s)",
                getTypeName(types.get(types.size() - 1).getText()),
                ctx.NAME().getText(),
                String.join(", ", arguments)
        );
    }
```

```
    @Override
    public String visitFunctionBody(ConverterParser.FunctionBodyContext ctx) {
        List<String> cases = new ArrayList<>();
        for (int i = 0; i < ctx.getChildCount(); i++) {
            cases.add(visitFunctionCase(ctx.functionCase(i)));
        }
        return String.join(LINE_SEPARATOR, cases);
    }

    @Override
    public String visitFunctionCase(ConverterParser.FunctionCaseContext ctx) {
        if (ctx.expressionNumbers.isEmpty()) {
            return processSimpleCase(ctx);
        } else {
            return processConditionalCase(ctx);
        }
    }

    @Override
    public String visitTypeName(ConverterParser.TypeNameContext ctx) {
        return getTypeName(ctx.getText());
    }

    private String processSimpleCase(ConverterParser.FunctionCaseContext ctx) {
        if (ctx.functionResult() != null) {
            return processReturn(ctx.functionResult());
        } else {
            return processGuards(ctx.functionGuards());
        }
    }

    private String processConditionalCase(ConverterParser.FunctionCaseContext
ctx) {
        ConverterParser.FunctionArgumentsCaseContext caseContext =
ctx.functionArgumentsCase();
        List<String> conditions = new ArrayList<>();
        for (int i = 0; i < ctx.argumentNumber; i++) {
            if (ctx.expressionNumbers.contains(i)) {
                conditions.add(
                        String.format(
                                "arg%d == %s",
                                i,
                                new VariableMappingVisitor(ctx.variables)
                                        .visitExpression(

caseContext.functionArgument(i).expression()
                                        )
                        )
                );
            }
        }
        String functionReturnStatement;
        if (ctx.functionResult() != null) {
            functionReturnStatement = processReturn(ctx.functionResult());
        } else {
            functionReturnStatement = processGuards(ctx.functionGuards());
        }
        return String.format(
```

```
                "if (%s) {%n    %s%n}",
                String.join(" && ", conditions),
                addOffset(functionReturnStatement)
        );
    }

    private String processReturn(ConverterParser.FunctionResultContext ctx) {
        return String.format(
                "return %s;",
                new VariableMappingVisitor(
                        ((ConverterParser.FunctionCaseContext)
ctx.getParent()).variables
                ).visitExpression(ctx.expression())
        );
    }

    private String processGuards(ConverterParser.FunctionGuardsContext ctx) {
        VariableMappingVisitor mappingVisitor =
                new VariableMappingVisitor(
                        ((ConverterParser.FunctionCaseContext)
ctx.getParent()).variables
                );
        List<String> guards = new ArrayList<>();
        for (int i = 0; i < ctx.getChildCount(); i++) {
            ConverterParser.ConditionContext conditionContext =
ctx.condition(i);
            guards.add(
                    String.format(
                            "if (%s) {%n    return %s;%n}",

mappingVisitor.visitBoolExpression(conditionContext.boolExpression()),

mappingVisitor.visitExpression(conditionContext.functionResult().expression())
                    )
            );
        }
        return String.join(" else ", guards);
    }

    private static String getTypeName(String type) {
        return PRIMITIVE_TYPES.getOrDefault(type, type);
    }

    private static String addOffset(String codeBlock) {
        return codeBlock.replace(LINE_SEPARATOR, LINE_OFFSET_SEPARATOR);
    }
}
```

VariableMappingVisitor:

```
package converter.visitor;

import converter.parser.ConverterBaseVisitor;
import converter.parser.ConverterParser;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
```

```
public class VariableMappingVisitor extends ConverterBaseVisitor<String> {
    private static final Map<String, String> BOOLEANS = Map.of(
            "False", "false",
            "True", "true"
    );
    private final Map<String, Integer> variableNumbers;

    public VariableMappingVisitor(Map<String, Integer> variableNumbers) {
        this.variableNumbers = variableNumbers;
    }

    @Override
    public String visitExpression(ConverterParser.ExpressionContext ctx) {
        if (ctx.mathExpression() != null) {
            return visitMathExpression(ctx.mathExpression());
        } else {
            return visitBoolExpression(ctx.boolExpression());
        }
    }

    @Override
    public String visitMathExpression(ConverterParser.MathExpressionContext ctx)
{
        if (ctx.mathExpression() != null) {
            return String.format(
                    "%s %s %s",
                    visitMathExpression(ctx.mathExpression()),
                    ctx.sign.getText(),
                    visitSummand(ctx.summand())
            );
        }
        return visitSummand(ctx.summand());
    }

    @Override
    public String visitSummand(ConverterParser.SummandContext ctx) {
        if (ctx.summand() != null) {
            return String.format(
                    "%s %s %s",
                    visitSummand(ctx.summand()),
                    ctx.sign.getText(),
                    visitFactor(ctx.factor())
            );
        }
        return visitFactor(ctx.factor());
    }

    @Override
    public String visitFactor(ConverterParser.FactorContext ctx) {
        if (ctx.INT() != null) {
            return ctx.INT().getText();
        } else if (ctx.functionApplication() != null) {
            return visitFunctionApplication(ctx.functionApplication());
        } else if (ctx.variable() != null) {
            return "arg" + variableNumbers.get(ctx.variable().getText());
        } else {
            String minus = "";
            if (ctx.minus != null) {
```

```
                minus = ctx.minus.getText();
            }
            return String.format(
                    "%s(%s)",
                    minus,
                    visitMathExpression(ctx.mathExpression())
            );
        }
    }

    @Override
    public String visitBoolExpression(ConverterParser.BoolExpressionContext ctx)
{
        if (ctx.boolExpression() != null) {
            return String.format(
                    "%s || %s",
                    visitBoolExpression(ctx.boolExpression()),
                    visitConjunction(ctx.conjunction())
            );
        }
        return visitConjunction(ctx.conjunction());
    }

    @Override
    public String visitConjunction(ConverterParser.ConjunctionContext ctx) {
        if (ctx.conjunction() != null) {
            return String.format(
                    "%s && %s",
                    visitConjunction(ctx.conjunction()),
                    visitBoolean(ctx.boolean_())
            );
        }
        return visitBoolean(ctx.boolean_());
    }

    @Override
    public String visitBoolean(ConverterParser.BooleanContext ctx) {
        if (ctx.BOOLEAN() != null) {
            return BOOLEANS.get(ctx.BOOLEAN().getText());
        } else if (ctx.variable() != null) {
            return "arg" + variableNumbers.get(ctx.variable().getText());
        } else if (ctx.comparison() != null) {
            return visitComparison(ctx.comparison());
        } else {
            String inversion = "";
            if (ctx.inversion != null) {
                inversion = "!";
            }
            return String.format(
                    "%s(%s)",
                    inversion,
                    visitBoolExpression(ctx.boolExpression())
            );
        }
    }

    @Override
    public String visitComparison(ConverterParser.ComparisonContext ctx) {
        return String.format(
```

```
                "%s %s %s",
                visitMathExpression(ctx.mathExpression(0)),
                ctx.sign.getText(),
                visitMathExpression(ctx.mathExpression(1))
        );
    }

    @Override
    public String
visitFunctionApplication(ConverterParser.FunctionApplicationContext ctx) {
        List<String> arguments = new ArrayList<>();
        for (ConverterParser.ExpressionContext expression : ctx.expression()) {
            arguments.add(visitExpression(expression));
        }
        return String.format(
                "%s(%s)",
                ctx.NAME.getText(),
                String.join(", ", arguments)
        );
    }
}
```

ConversionException:

```
package converter.exception;

public class ConversionException extends RuntimeException {
    public ConversionException(String message) {
        super(message);
    }
}
```

FileConverter:

```
package converter;

import converter.parser.ConverterLexer;
import converter.parser.ConverterParser;
import converter.visitor.ProgramVisitor;
import org.antlr.v4.runtime.CharStream;
import org.antlr.v4.runtime.CharStreams;
import org.antlr.v4.runtime.CommonTokenStream;

import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;

public class FileConverter {
    private static final Charset CHARSET = StandardCharsets.UTF_8;

    public static void convert(Path fromFile, Path toFile) throws IOException {
        CharStream charStream = CharStreams.fromPath(fromFile, CHARSET);
        ConverterLexer lexer = new ConverterLexer(charStream);
        CommonTokenStream tokenStream = new CommonTokenStream(lexer);
        ConverterParser parser = new ConverterParser(tokenStream);
```

```
            Files.writeString(
                    toFile,
                    new ProgramVisitor().visitProgram(parser.program()),
                    CHARSET
            );
        }
}
```

2 Разработка тестов

Тесты корректных случаев

Название теста	Описание
NoArgumentsFunctions	Функции, не принимающие аргументов
UnaryFunctions	Функции, принимающие один аргумент
N-aryFunctions	Функции, принимающие несколько аргументов
NotFormattedFunctions	Неформатированный файл с функциями
IntExpressionFunction	Функция, возвращающая числовое выражение
BoolExpressionFunction	Функция, возвращающая логическое значение
ValueArgumentsFunctions	Функции, принимающие в качестве аргументов константные значения
VariableArgumentsFunctions	Функции, принимающие в качестве аргументов переменные
MixedArgumentsFunctions	Функции, принимающие в качестве аргументов выражения, содержащие константы и переменные
GuardsFunctions	Функции с ветвлением по условиям
RecursiveFunctions	Рекурсивные функции
InvokingFunctions	Функции, вызывающие другие функции

Тесты некорректных случаев

Название теста	Описание
FunctionWithoutBody	Функция без описания тела
FunctionNameMismatch	Несовпадение имён функций при объявлении
ReturnTypeMismatch	Несовпадение возвращаемого типа функции
ArgumentsTypeMismatch	Несовпадение типа аргументов функции

WrongArgumentsNumber	Неправильное количество аргументов у функции
NonExistingFunction	Использование несуществующей функции
MathExpressionTypeMismatch	Несовпадение типа с числовым типом
BoolExpressionTypeMismatch	Несовпадение типа с булевым типом
ApplicationArgumentsTypeMismatch	Несовпадение типа аргументов при применении функции