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

Методы трансляции

Отчёт по лабораторной работе № 2

«Ручное построение нисходящих синтаксических анализаторов»

Работу выполнил:

Ивченков Дмитрий Артемович, M33341

Санкт-Петербург

2023

1 Разработка грамматики

Вариант 11: Описание массивов в Kotlin.

Модификация: Добавить описание отображений.

Грамматика:

S -> var n : C
C -> A
C -> M
A -> Array < T > N
M -> Map < T , T > N
T -> n T′ N
T -> C
T′ -> < T >
T′ -> ε
N -> ?
N -> ε

Нетерминал	Описание
S	Объявление массива
C	Тип коллекции
A	Тип массива
M	Тип отображения
T	Имя типа
T′	Параметризация типа
N	Nullability типа

Терминал	Описание
var	Объявление переменной
n	Имя переменной или типа
:	Разделитель переменной и типа
Array	Тип массива
Map	Тип отображения
<	Открывающая скобка параметра
,	Запятая
>	Закрывающая скобка параметра
?	Nullability типа

2 Построение лексического анализатора

Enum токенов:

```
package parser.grammar.terminal;

public enum Token {
    VAR("var"),
    NAME("name"),
    SEPARATOR(":"),
    ARRAY("Array"),
    MAP("Map"),
    LEFT_BRACKET("<"),
    COMMA(","),
    RIGHT_BRACKET(">"),
    NULLABLE("?"),
    EMPTY("''"),
    END("$");

    private final String value;

    Token(String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }
}
```

Лексический анализатор:

```
package parser.lexic;

import parser.grammar.terminal.Token;

import java.text.ParseException;
import java.util.ArrayList;
import java.util.List;

public class Lexer {
    private final String string;
    private int position;
    private Token currentToken;

    public Lexer(String string) {
        this.string = string;
        position = 0;
    }

    public Token getCurrentToken() {
        return currentToken;
    }

    public void takeNextToken() throws ParseException {
        skipWhitespace();
        if (!hasNextCharacter()) {
            currentToken = Token.END;
        } else if (isSeparator()) {
            currentToken = Token.SEPARATOR;
```

```
        } else if (isLeftBracket()) {
            currentToken = Token.LEFT_BRACKET;
        } else if (isComma()) {
            currentToken = Token.COMMA;
        } else if (isRightBracket()) {
            currentToken = Token.RIGHT_BRACKET;
        } else if (isNullable()) {
            currentToken = Token.NULLABLE;
        } else {
            int start = position;
            if (isIdentifier()) {
                String name = string.substring(start, position);
                if (isVar(name)) {
                    currentToken = Token.VAR;
                } else if (isArray(name)) {
                    currentToken = Token.ARRAY;
                } else if (isMap(name)) {
                    currentToken = Token.MAP;
                } else {
                    currentToken = Token.NAME;
                }
            } else {
                throw new ParseException("Unknown token found", position);
            }
        }
    }

    public List<Token> getTokensList() throws ParseException {
        List<Token> tokens = new ArrayList<>();
        while (hasNextCharacter()) {
            takeNextToken();
            tokens.add(getCurrentToken());
            skipWhitespace();
        }
        return tokens;
    }

    public int getPosition() {
        return position;
    }

    private boolean isVar(String name) {
        return name.equals(Token.VAR.getValue());
    }

    private boolean isSeparator() {
        return isToken(Token.SEPARATOR);
    }

    private boolean isArray(String name) {
        return name.equals(Token.ARRAY.getValue());
    }

    private boolean isMap(String name) {
        return name.equals(Token.MAP.getValue());
    }

    private boolean isLeftBracket() {
        return isToken(Token.LEFT_BRACKET);
```

```
    }

    private boolean isComma() {
        return isToken(Token.COMMA);
    }

    private boolean isRightBracket() {
        return isToken(Token.RIGHT_BRACKET);
    }

    private boolean isNullable() {
        return isToken(Token.NULLABLE);
    }

    private boolean isToken(Token token) {
        String value = token.getValue();
        for (int i = 0; i < value.length(); i++) {
            if (isAvailablePosition(position + i) && string.charAt(position + i)
!= value.charAt(i)) {
                return false;
            }
        }
        position += value.length();
        return true;
    }

    private boolean isIdentifier() {
        int currentPosition = position;
        if (!Character.isJavaIdentifierStart(string.charAt(currentPosition))) {
            return false;
        }
        while (isAvailablePosition(currentPosition)
                &&
Character.isJavaIdentifierPart(string.charAt(currentPosition))) {
            currentPosition++;
        }
        position = currentPosition;
        return true;
    }

    private void skipWhitespace() {
        while (hasNextCharacter() &&
Character.isWhitespace(string.charAt(position))) {
            position++;
        }
    }

    private boolean hasNextCharacter() {
        return isAvailablePosition(position);
    }

    private boolean isAvailablePosition(int position) {
        return position < string.length();
    }
}
```

3 Построение синтаксического анализатора

Множества FIRST и FOLLOW для нетерминалов грамматики:

Нетерминал	FIRST	FOLLOW
S	var	$
C	Array, Map	S, >, ,
A	Array	$, >, ,
M	Map	$, >, ,
T	n, Array, Map	>, ,
T′	ε, <	?, >, ,
N	ε, ?	$, >, ,

Структура данных для хранения дерева:

```
package parser.tree;

import java.util.ArrayList;
import java.util.List;
import java.util.Objects;

public class Tree {
    private static int nodeIndex = 0;
    private final String element;
    private final List<Tree> children;

    public Tree(String element) {
        this(element, new ArrayList<>());
    }

    public Tree(String element, List<Tree> children) {
        this.element = element;
        this.children = children;
    }

    public Tree addChild(Tree child) {
        children.add(child);
        return this;
    }

    @Override
    public String toString() {
        if (children.isEmpty()) {
            return element;
        }
        return String.format("(%s %s)", element, children);
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Tree that) {
            return Objects.equals(element, that.element) &&
Objects.deepEquals(children, that.children);
        }
```

```
        return false;
    }

    @Override
    public int hashCode() {
        return element.hashCode() + children.hashCode();
    }

    public String toGraphViz() {
        return toGraphViz(getNextIndex());
    }

    private String toGraphViz(int index) {
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.append(String.format("%d [label = \"%s\"]%n", index,
element));
        for (Tree tree : children) {
            int childIndex = getNextIndex();
            stringBuilder
                    .append(String.format("%d -> %d%n", index, childIndex))
                    .append(tree.toGraphViz(childIndex));
        }
        return stringBuilder.toString();
    }

    private int getNextIndex() {
        return nodeIndex++;
    }
}
```

Enum нетерминалов:

```
package parser.grammar.nonterminal;

public enum NonTerminal {
    S,
    C,
    A,
    M,
    T,
    T_PRIME,
    N
}
```

Синтаксический анализатор:

```
package parser.syntax;

import parser.grammar.nonterminal.NonTerminal;
import parser.grammar.terminal.Token;
import parser.lexic.Lexer;
import parser.tree.Tree;

import java.text.ParseException;
import java.util.List;

public class Parser {
    private Lexer lexer;
```

```
public Tree parse(String string) throws ParseException {
    lexer = new Lexer(string);
    lexer.takeNextToken();
    return parseS();
}

private Tree parseS() throws ParseException {
    Tree sTree = new Tree(NonTerminal.S.name())
            .addChild(getExpected(Token.VAR))
            .addChild(getExpected(Token.NAME))
            .addChild(getExpected(Token.SEPARATOR))
            .addChild(parseC());
    expect(Token.END);
    return sTree;
}

private Tree parseC() throws ParseException {
    Tree cTree = new Tree(NonTerminal.C.name());
    Token currentToken = lexer.getCurrentToken();
    if (currentToken == Token.ARRAY) {
        return cTree.addChild(parseA());
    } else if (currentToken == Token.MAP) {
        return cTree.addChild(parseM());
    }
    throw newParseException(List.of(Token.ARRAY, Token.MAP), currentToken);
}

private Tree parseA() throws ParseException {
    return new Tree(NonTerminal.A.name())
            .addChild(getExpected(Token.ARRAY))
            .addChild(getExpected(Token.LEFT_BRACKET))
            .addChild(parseT())
            .addChild(getExpected(Token.RIGHT_BRACKET))
            .addChild(parseN());
}

private Tree parseM() throws ParseException {
    return new Tree(NonTerminal.M.name())
            .addChild(getExpected(Token.MAP))
            .addChild(getExpected(Token.LEFT_BRACKET))
            .addChild(parseT())
            .addChild(getExpected(Token.COMMA))
            .addChild(parseT())
            .addChild(getExpected(Token.RIGHT_BRACKET))
            .addChild(parseN());
}

private Tree parseN() throws ParseException {
    Tree nTree = new Tree(NonTerminal.N.name());

    Token currentToken = lexer.getCurrentToken();
    if (currentToken == Token.NULLABLE) {
        nTree.addChild(new Tree(currentToken.getValue()));
        lexer.takeNextToken();
        return nTree;
    } else if (currentToken == Token.END
            || currentToken == Token.RIGHT_BRACKET
            || currentToken == Token.COMMA) {
        nTree.addChild(new Tree(Token.EMPTY.getValue()));
```

```
            return nTree;
        }
        throw newParseException(
                List.of(Token.NULLABLE, Token.END, Token.RIGHT_BRACKET,
Token.COMMA),
                currentToken
        );
    }

    private Tree parseT() throws ParseException {
        Tree t = new Tree(NonTerminal.T.name());

        Token currentToken = lexer.getCurrentToken();
        if (currentToken == Token.NAME) {
            t.addChild(new Tree(currentToken.getValue()));
            lexer.takeNextToken();
            t.addChild(parseTPrime())
                    .addChild(parseN());
            return t;
        } else if (currentToken == Token.ARRAY || currentToken == Token.MAP) {
            return t.addChild(parseC());
        }
        throw newParseException(
                List.of(Token.NAME, Token.ARRAY, Token.MAP),
                currentToken
        );
    }

    private Tree parseTPrime() throws ParseException {
        Tree tPrime = new Tree(NonTerminal.T_PRIME.name());

        Token currentToken = lexer.getCurrentToken();
        if (currentToken == Token.LEFT_BRACKET) {
            tPrime.addChild(new Tree(currentToken.getValue()));
            lexer.takeNextToken();
            tPrime.addChild(parseT())
                    .addChild(getExpected(Token.RIGHT_BRACKET));
            return tPrime;
        } else if (currentToken == Token.NULLABLE
                || currentToken == Token.RIGHT_BRACKET
                || currentToken == Token.COMMA) {
            tPrime.addChild(new Tree(Token.EMPTY.getValue()));
            return tPrime;
        }
        throw newParseException(
                List.of(Token.LEFT_BRACKET, Token.NULLABLE, Token.RIGHT_BRACKET,
Token.COMMA),
                currentToken
        );
    }

    private Tree getExpected(Token expected) throws ParseException {
        expect(expected);
        lexer.takeNextToken();
        return new Tree(expected.getValue());
    }

    private void expect(Token expected) throws ParseException {
        Token current = lexer.getCurrentToken();
```

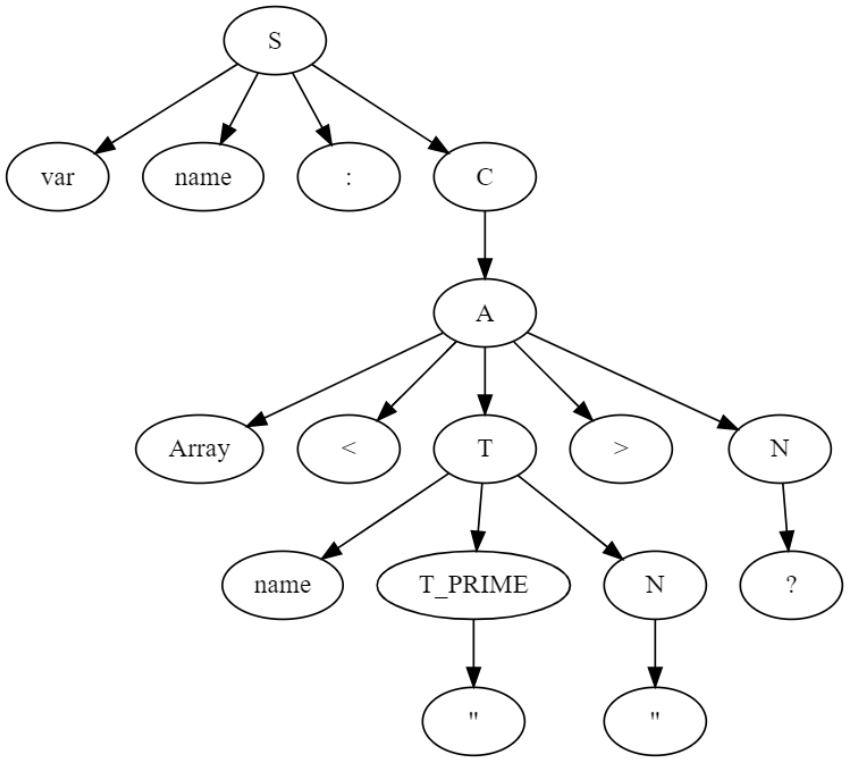
```
        if (current != expected) {
            throw newParseException(List.of(expected), current);
        }
    }

    private ParseException newParseException(List<Token> expected, Token found)
{
        String expectedOptions = String.join(
                " or ",
                expected.stream().map(Token::getValue).toList()
        );
        return new ParseException(
                "Expected " + expectedOptions + " but found " +
found.getValue(),
                lexer.getPosition() - found.getValue().length()
        );
    }
}
```

4 Визуализация дерева разбора

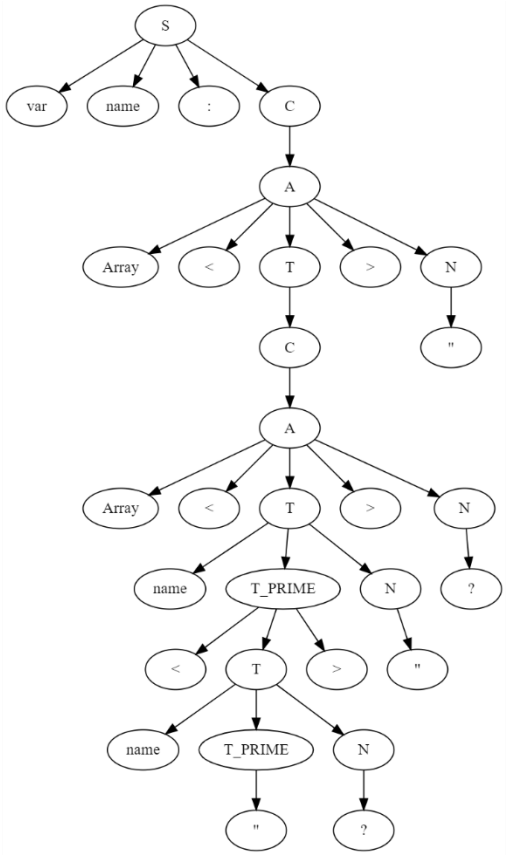
Визуализация проводится с помощью системы GraphViz.

Примеры:

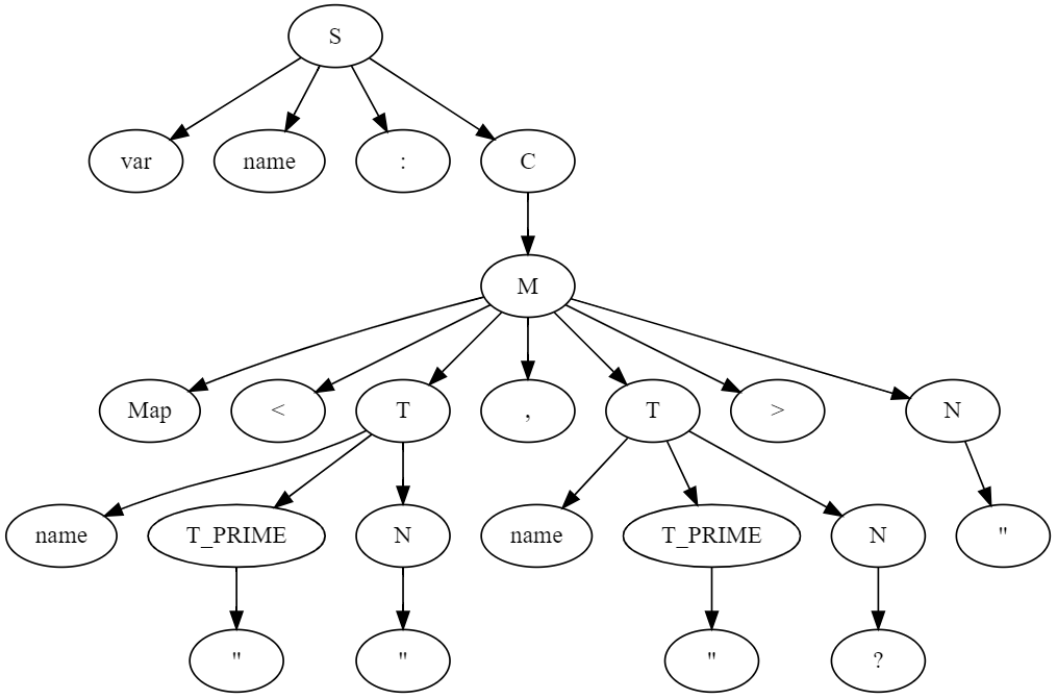
var a : Array<Int>?



var a : Array<Array<Comparator<String?>>?>



var a : Map<Int, String?>



5 Подготовка набора тестов

Тесты лексического анализатора

Тест	Описание
var	Токен ключевого слова
:	Токен разделителя
Array	Токен массива
Map	Токен отображения
<	Токен открывающей скобки
,	Токен запятой
>	Токен закрывающей скобки
?	Токен Nullability
name	Корректное имя
Name	Корректное имя
NAME	Корректное имя
Name123	Корректное имя
array123name	Корректное имя
array_name	Корректное имя
_name	Корректное имя
varvar	Корректное имя, не два токена var
ArrayArray	Корректное имя, не два токена Array
MapMap	Корректное имя, не два токена Map
123name	Некорректное имя
!name	Некорректное имя
-name	Некорректное имя
.name	Некорректное имя
<<>>	Последовательность токенов
abc def ghi	Последовательность токенов
Array<Array<Int>>	Последовательность токенов
Array<String?>?	Последовательность токенов
var array:Array<Int>	Последовательность токенов
var map:Map<Int, String>	Последовательность токенов
var \t \n\r	Токен с пробельными символами
\n var \r var \n \t var \t	Последовательность токенов с пробельными символами
var\n\n\n a \n : \t\t Array \n < \t b\r \n>\n \t	Последовательность токенов с пробельными символами

Тесты синтаксического анализатора:

Тест	Описание
var a : Array<Unit>	Примитивный массив
var a : Array<A>	Та же самая структура
var a : Array<Array<String>>	Массив массивов
var a : Array<Array<A>>	Та же самая структура
var a: Array<Set<Int>>	Массив параметризованного типа
var a: Array<A<B>>	Та же самая структура
var a: Array<Int?>	Массив обнуляемого типа
var a: Array<A?>	Та же самая структура
var a : Array<Int>?	Обнуляемый массив
var a : Array<A>?	Та же самая структура
var a : Array<Int?>?	Обнуляемый массив обнуляемого типа
var a : Array<A?>?	Та же самая структура
var a: Array<Set<Int?>?>	Массив параметризованного обнуляемого типа
var a: Array<A<B?>?>	Та же самая структура
var a : Array<Array<String?>>?	Обнуляемый массив массивов обнуляемого типа
var a : Array<Array<A?>>?	Та же самая структура
var a : Map<Int, String>	Примитивное отображение
var a : Map<A, B>	Та же самая структура
var a : Map<Array<Int>, Map<Int, Int>>	Отображение из массива в отображение
var a : Map<Array<A>, Map<B, C>>	Та же самая структура
var a : Map<Set<Int>, String>	Отображение из параметризованного типа
var a : Map<A<B>, C>	Та же самая структура
var a : Map<Int, Map<String?, String>>?	Обнуляемое отображение в отображение обнуляемого типа
var a : Map<A, Map<B?, B>>?	Та же самая структура
var	Некорректное объявление массива
var a	Некорректное объявление массива
var a :	Некорректное объявление массива
var a : A var a : Array	Некорректное объявление массива
var a : Array<	Некорректное объявление массива
var a : Array<>	Некорректное объявление массива
var a : Array<Int	Некорректное объявление массива
var a : Array<Int<	Некорректное объявление массива
var a : Array>Int>	Некорректное объявление массива
val a : Array<Int>	Некорректное объявление массива
var : Array<Int>	Некорректное объявление массива
var a Array<Int>	Некорректное объявление массива
var a : array<Int>	Некорректное объявление массива

var a : A<Int>	Некорректное объявление массива
var a : Array<Array>	Некорректное объявление массива
var : Array<Int>	Некорректное объявление массива
a : Array<Int>	Некорректное объявление массива
var a : <Int>	Некорректное объявление массива
var a : <Int>Array	Некорректное объявление массива
var a Array : Int	Некорректное объявление массива
var a : Array<Array<>	Некорректное объявление массива
var a : Array<Array<>>	Некорректное объявление массива
var a : Array<Array<>Int>	Некорректное объявление массива
var a : Array<Int<>>	Некорректное объявление массива
var a : Array<Int??>	Некорректное объявление массива
var a : Array<?Int>	Некорректное объявление массива
var a : Array?<Int>	Некорректное объявление массива
var a : ?Array<Int>	Некорректное объявление массива
var a? : Array<Int>	Некорректное объявление массива
var ?a : Array<Int>	Некорректное объявление массива
var? a : Array<Int>	Некорректное объявление массива
var ? : Array<Int>	Некорректное объявление массива
var a : Array<?>	Некорректное объявление массива
var a : Map	Некорректное объявление отображения
var a : Map<	Некорректное объявление отображения
var a : Map<Int	Некорректное объявление отображения
var a : Map<Int,	Некорректное объявление отображения
var a : Map<Int,Int	Некорректное объявление отображения
var a : Map<Int,Int>>	Некорректное объявление отображения
var a : Map<<Int,Int>	Некорректное объявление отображения
var a : Map<<Int,Int>>	Некорректное объявление отображения
var a : Map<,Int,Int>	Некорректное объявление отображения
var a : Map<Int,Int,>	Некорректное объявление отображения
var a : Map<Int Int>	Некорректное объявление отображения
var a : Map<Int>	Некорректное объявление отображения
var a : Map<Int ? Int>	Некорректное объявление отображения
var a : Map<?Int, Int>	Некорректное объявление отображения
var a : Map<Int, ?Int>	Некорректное объявление отображения