НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ им. Р.Е.АЛЕКСЕЕВА

Институт радиоэлектроники и информационных технологий

Кафедра прикладной математики

Курсовая работа

“Компилятор модельного языка”

Руководитель:

Санников Николай Александрович

Студент:

Корноухов Илья Владимирович

Группа 23-ПМ-2

Нижний Новгород 2025

**Содержание**

[Содержание 2](#_Toc1533514480)

[Введение 3](#_Toc1533514480)

1 [Постановка задачи 4](#_Toc1533514480)

2 [Формальная модель задачи 5](#_Toc1533514480)

3 [Спецификация основных процедур и функций 6](#_Toc1533514480)

[3.1 Лексический анализатор 6](#_Toc1533514480)

3.2 [Синтаксический анализатор](#_Toc1533514480) 11

3.3 [Семантический анализатор 1](#_Toc1533514480)3

3.4 [Генерация внутреннего представления программы 1](#_Toc1533514480)5

3.5 [Интерпретатор программы 1](#_Toc1533514480)6

[7 Работа с программным средством 1](#_Toc1533514480)7

[Заключение 1](#_Toc1533514480)8

[Приложение А – Текст программы 1](#_Toc1533514480)9

[Приложение Б – Контрольный пример 1](#_Toc1533514480)23

**Введение**

Компиляторы являются неотъемлемой частью сферы информационных технологий. Они позволяют возможность разрабатывать программы на языках высокого уровня, переводя написанный человеком код в форму, понятную компьютеру.

Изучение устройства компилятора позволяет глубже понять внутреннюю структуру языков программирования, особенности синтаксиса и семантики. Этот процесс включает в себя работу с формальными грамматиками, конечными автоматами, деревьями разбора и другими концепциями теории вычислений. Такой опыт представляет ценность не только с образовательной точки зрения, но и с практической, поскольку формирует фундаментальные навыки, необходимые при разработке языков, написании парсеров, и других задач, связанных с анализом и преобразованием кода.

**Постановка задачи**

Разработать компилятор модельного языка, выполнив следующие действия.

В соответствии с номером варианта составить формальное описание модельного языка программирования с помощью РБНФ, диаграмм Вирта или формальных грамматик.

Написать пять содержательных примеров программ, раскрывающих особенности конструкций учебного языка программирования, отразив в этих примерах все его функциональные возможности.

Составить таблицы лексем и диаграмму состояний с действиями для распознавания и формирования лексем языка.

По диаграмме с действиями написать функцию сканирования текста входной программы на модельном языке.

Разработать программное средство, реализующее лексический анализ текста программы на входном языке.

Реализовать синтаксический анализатор текста программы на модельном языке методом рекурсивного спуска.

Построить цепочку вывода и дерево разбора простейшей программы на модельном языке из начального символа грамматики.

Дополнить синтаксический анализатор процедурами проверки семантической правильности программы на модельном языке в соответствии с контекстными условиями вашего варианта.

Распечатать пример таблиц идентификаторов и двуместных операций.

Показать динамику изменения содержимого стека при семантическом анализе программы на примере одного синтаксически правильного выражения.

Записать правила вывода грамматики с действиями по переводу в ПОЛИЗ программы на модельном языке.

Пополнить разработанное программное средство процедурами, реализующими генерацию внутреннего представления введенной программы в форме ПОЛИЗа.

Разработать интерпретатор ПОЛИЗа программы на модельном языке.

Составить набор контрольных примеров, демонстрирующих все возможные типы лексических, синтаксических и семантических ошибок в программах на модельном языке, перевод в ПОЛИЗ различных конструкций языка представить ход интерпретации синтаксически и семантически правильной программы с помощью таблицы.

**Формальная модель задачи**

Регулярной грамматикой G называется кортеж вида

<T, N, S, P>, где:

T - множество терминалов

N - множество нетерминалов

S - начальный символ грамматики из множества нетерминалов

P - правила грамматики, имеющие вид A -> b, A -> bB, A -> Bb, где A и B - нетерминал, b - терминал.

Конечным автоматом M называется кортеж вида

<Q, T, S, F, P>, где:

Q - множество состояний автомата

T - алфавит, состоящий из терминалов исходной грамматики

S - начальное состояние

F - множество конечных состояний

P - множество переходов вида <q1, a> -> q2, где q1 и q2 - состояния автомата, a - символ из алфавита

КС-грамматикой G1 называется кортеж вида <T, N, S, P>, где:

T - множество терминалов

N - множество нетерминалов

S - начальный символ грамматики из множества нетерминалов

P - правила грамматики, имеющие вид a -> b, где a - нетерминал, b - последовательность терминалов и нетерминалов

.

**Спецификация основных процедур и функций**

**Лексический анализатор**

Компилятор написан на языке программирования Java, исходный код представлен в Приложении А.Приложение Б содержит контрольные примеры программ для проверки работы компилятора.

Лексический анализатор (ЛА) – это первый этап процесса компиляции, на котором символы, составляющие исходную программу, группируются в лексемы.

На вход лексическому анализатору подаётся цепочка символов исходного кода компилируемой программы. ЛА проверяет цепочку на принадлежность к регулярной грамматике модельного языка и формирует таблицы лексем и идентификаторов.

Лексический анализатор содержит набор процедур и состояний, необходимых для составления таблиц лексем и идентификаторов.

При чтении символа из цепочки ЛА может переходить из одного состояния в другое и вызывать процедуры.

Лексический анализатор использует таблицы:

serviceTable - таблица служебных слов

delimiterTable - таблица ограничителей

identifierTable - таблица идентификаторов

numberTable - таблица числовых значений

lexemesSeqTable - таблица лексем исходной программы.

Лексический анализатор имеет следующие состояния:

READ - начальное состояние, чтение лексемы

IDENT - чтение идентификатора

NUM\_BIN, NUM\_BIN\_FIN - чтение двоичного числа

NUM\_OCT, NUM\_OCT\_FIN - чтение восьмеричного числа

NUM\_DEC, NUM\_DEC\_FIN - чтение десятичного числа

NUM\_HEX, NUM\_HEX\_FIN - чтение шестнадцатеричного числа

NUM\_REAL\_POINT\_1, NUM\_REAL\_POINT\_ORDER\_START\_1, NUM\_REAL\_ORDER, NUM\_REAL\_POINT\_ORDER\_START\_2, NUM\_REAL\_POINT\_2, NUM\_REAL\_ORDER\_OR\_HEX, NUM\_REAL\_POINT\_ORDER - чтение действительного числа

DELIMITER - чтение разделителя

COMMENT, COMMENT\_START, COMMENT\_END - чтение комметнария

MORE\_THEN\_EQUAL - чтение разделителя “>=”

LESS\_THEN\_EQUAL - чтение разделителя “<=”

END - конечное состояние

ERROR - состояние ошибки

Лексический анализатор содержит набор переменных:

lexBuffer - переменная для накопления символов текущей лексемы

currentState - текущее состояние лексического анализатора

currentChar - текущий прочитанный символ

curLexId - номер лексемы из таблицы

curTableId - номер таблицы лексемы

curRow - номер текущей строки программы

curCol - номер текущего символа в строке программы

Лексический анализатор может вызывать следующие процедуры:

isBinAllow - проверка текущего символа на принадлежность к символам двоичного числа

isOctAllow - проверка текущего символа на принадлежность к символам восьмеричного числа

isDecAllow - проверка текущего символа на принадлежность к символам десятичного числа

isHexAllow - проверка текущего символа на принадлежность к символам шестнадцатеричного числа

isNumber - проверка текущего символа на принадлежность к символам числовой строки

isLetter - проверка текущего символа на принадлежность к множеству букв

clean - очистка lexBuffer

add - добавление текущего символа к lexBuffer

read - чтение следующего символа в currentChar

check(t) - проверка содержимого lexBuffer на соответствие лексеме из таблицы t

put(t, nt, d) - добавление лексемы из lexBuffer в таблицу t с возможностью указания типа nt и системы счисления d при сохранении числа

write(t, l) - запись номера t таблицы и номера l лексемы в lexemesSeqTable.

Ниже представлен модельный язык в формате РБНФ:

<буква>::= (A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p q | r | s | t | u | v | w | x | y | z)

<цифра>::= (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)

<операции\_группы\_отношения>:: = (< > | = | < | <= | > | >=)

<операции\_группы\_сложения>:: =(+ | - | or)

<операции\_группы\_умножения>::=(\* | / | and)

<логическая\_константа>::= (true | false)

<унарная\_операция>::= not

<тип>::= (int | float | bool)

<двоичное>::= {/ 0 | 1 /} (B | b)

<восьмеричное>::= {/ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 /} (O | o)

<идентификатор>::= <буква> {<буква> | <цифра>}

<шестнадцатеричное>::= <цифра> {<цифра> | A | B | C | D | E | F | a | b | c | d | e | f} (H | h)

<десятичное>::= {/ <цифра> /} [D | d]

<числовая\_строка>::= {/ <цифра> /}

<целое>::= (<двоичное> | <восьмеричное> | <десятичное> | <шестнадцатеричное>)

<порядок>::= ( E | e )[+ | -] <числовая\_строка>

<ввода>::= read (<идентификатор> {, <идентификатор> })

<описание>::= <тип> <идентификатор> { , <идентификатор> }

<действительное>::= (<числовая\_строка> <порядок> | [<числовая\_строка>] . <числовая\_строка> [порядок])

<число>::= (<целое> | <действительное>)

<множитель>::= (<идентификатор> | <число> | <логическая\_константа> | <унарная\_операция> <множитель> | <выражение>)

<слагаемое>::= <множитель> {<операции\_группы\_умножения> <множитель>}

<операнд>::= <слагаемое> {<операции\_группы\_сложения> <слагаемое>}

<выражение>::= <операнд>{<операции\_группы\_отношения> <операнд>}

<составной>::= <оператор> { ( : | перевод строки) <оператор> }

<присваивания>::= <идентификатор> ass <выражение>

<условный>::= if <выражение> then <оператор> [ else <оператор>]

<фиксированного\_цикла>::= for <присваивания> to <выражение> do <оператор>

<условного\_цикла>::= while <выражение> do <оператор>

<вывода>::= write (<выражение> {, <выражение> })

<оператор>::= (<составной> | <присваивания> | <условный> | <фиксированного\_цикла> | <условного\_цикла> | <ввода> | <вывода>)

<программа>::= “{“ {/ (<описание> | <оператор>) ; /} “}”

**Синтаксический анализатор**

Синтаксический анализ - этап процесса компиляции, в ходе которого последовательность лексем исходной программы проверяется на принадлежность к КС-грамматике языка.

Алгоритм синтаксического анализатора основан на методе рекурсивного спуска.

Синтаксический анализатор содержит функции для проверки начала оставшейся цепочки лексем на принадлежность к соответствующему нетерминалу:

OGO - операции группы отношения

OGS - операции группы сложения

OGU - операции группы умножения

LC - логическая константа

UO - унарная операция

T - тип

MNOZH - множитель

EXPR - выражение

SLAG - слагаемое

OPRND - операнд

ENTER - оператор ввода

OUT - оператор вывода

PRISV - оператор присваивания

USLOV - оператор условный

FIXLOOP - оператор фиксированного цикла

USLLOOP - оператор условного цикла

OPERATOR - оператор

DESC - описание

PROG - программа

В качестве входных данных СиА принимает таблицы лексем и идентификаторов, сформированные на этапе лексического анализа.

Результатом работы СиА является заключение о принадлежности последовательности лексем программы к КС-грамматике языка.

**Семантический анализатор**

Семантический анализ - этап компиляции, необходимый для проверки программы на соответствие семантическим соглашениям, таким как правильность типов данных в выражениях и операторах, а так же проверка описаний переменных.

Семантический анализ выполняется параллельно с синтаксическим.

Для проверки выражений в СиА:

1. Вводится таблица binOperationTable
2. Добавляется exprStack - стек обработки операндов в выражениях
3. Функции EXPR, MNOZH, EXPR, SLAG и OPRND заполняют стек операндами и операторами.
4. В функции EXPR выполняется проверка соответствия оператора операндам из стека согласно таблице бинарных операций и запись в стек результата операции.

Для проверки описаний в СиА:

1. Таблица идентификаторов дополняется столбцами isInit и type.
2. В функции DESC выполняется проверка отсутствия описания переменной, отметка переменной как описанной и присвоение ей типа.

Для проверки операторов в функции PRISV, USLOV, USLLOOP добавляется проверка типа в exprStack после обработки выражения.

Для выполнения семантического анализа используются функции:

getType(l) - функция для определения типа лексемы l из очереди лексем

getExprType(s) - функция обработки стека выражения s для определения типа выражения.

Результатом работы СиА является заключение о соблюдении семантических соглашений.

**Генерация внутреннего представления программы**

В качестве внутреннего представления программы используется польская инверсная запись ПОЛИЗ. Генерация ПОЛИЗ выполняется параллельно с синтаксическим анализом с помощью синтаксически управляемого перевода.

Для этого в СиА:

1. Вводятся таблицы: polizTable - таблица для записи ПОЛИЗ программы, polizPointerItems - таблица меток для записи меток перехода
2. В таблицу разделителей добавляются операции: ‘!’ - безусловный переход к метке, ‘!F’ - условный переход по лжи к метке, ‘R’ - чтение в переменную, ‘W’ - вывод значения в консоль
3. В функциях EXPR, SLAG, OPRND, MNOZH, ENTER, OUT, PRISV, USLOV, FIXLOOP и USLLOOP СиА выполняется формирование ПОЛИЗ программы и запись в polizTable.

**Интерпретатор программы**

Для выполнения ПОЛИЗ программы используется интерпретатор. Интерпретатор читает последовательность ПОЛИЗ по порядку один раз. При чтении операнда он заносится в стек, при чтении оператора его операнды выводятся из стека, результат операции записывается в стек.

Для хранения значений логических констант, переменных и чисел используется хэш таблица memory.

Для интерпретации программы используются функции:

getTypeById(t, l) - функция определения типа лексемы l из таблицы t

getValueById(t, l) - функция определения значения для лексемы l из таблицы t

getMemoryAddress(t, l) - функция получения ключа memory для лексемы l из таблицы t

getOperandsCount(s) - функция для получения колличества операндов для оператора s

getBinOpResultType(optr, opnd1, opnd2) - функция для получения типа результата бинарной операции для оператора optr и операндов opnd1, opnd2.

Результатом работы интерпретатора является выполнение операций из ПОЛИЗ.

**Работа с программным средством**

Для запуска программы необходимо выполнить команду: java -jar <Название исполняемого файла >.

По умолчанию программа ищет файл main.prog в текущем каталоге, переводит в ПОЛИЗ и интерпретирует.

Программа может принимать в качестве первого аргумента:

‘-i’ - перевод в ПОЛИЗ и интерпретация

‘-l’ - вывод в консоль заполненных таблиц, используемых при компиляции без интерпретации

‘-p’ - вывод ПОЛИЗ программы без интерпретации.

В качестве второго аргумента программа может принимать путь к файлу с исходным кодом компилируемой программы.

**Заключение**

В ходе выполнения курсовой работы была проведена практическая реализация базового компилятора, включающего основные этапы обработки исходного кода: лексический, синтаксический и семантический анализ, построение внутреннего представления и генерацию выходного кода с последующей интерпретацией. Это позволило на практике познакомиться с фундаментальными принципами, лежащими в основе трансляции языков программирования.

Работа над компилятором способствовала углублению понимания работы современных инструментов разработки и как они интерпретируют написанный код. Полученные знания и навыки могут быть полезны в дальнейшей учебной и профессиональной деятельности, особенно в областях, связанных с разработкой языков, парсеров, трансляторов и системного программного обеспечения.

**Приложение А – Текст программы**

package com.ilyakrn.entities.items;

public class BinOperationItem {

private final String operation;

private final Type operand1;

private final Type operand2;

private final Type result;

public BinOperationItem(String operation, Type operand1, Type operand2, Type result) {

this.operation = operation;

this.operand1 = operand1;

this.operand2 = operand2;

this.result = result;

}

public String getOperation() {

return operation;

}

public Type getOperand1() {

return operand1;

}

public Type getOperand2() {

return operand2;

}

public Type getResult() {

return result;

}

}

package com.ilyakrn.exceptions.external;

public class CompilerExternalException extends RuntimeException {

public CompilerExternalException(String message) {

super(message);

}

}

package com.ilyakrn.exceptions.internal;

public class CompilerInternalException extends RuntimeException {

public CompilerInternalException(String message) {

super(message);

}

}

package com.ilyakrn.entities.items;

public class DelimiterItem {

private String lexeme;

public String getLexeme() {

return lexeme;

}

public void setLexeme(String lexeme) {

this.lexeme = lexeme;

}

public DelimiterItem(String lexeme) {

this.lexeme = lexeme;

}

}

package com.ilyakrn.entities.items;

public class IdentifierItem {

private final String lexeme;

private boolean isInit;

private Type type;

public void setInit(boolean init) {

isInit = init;

}

public void setType(Type type) {

this.type = type;

}

public String getLexeme() {

return lexeme;

}

public boolean isInit() {

return isInit;

}

public Type getType() {

return type;

}

public IdentifierItem(String lexeme, boolean isInit, Type type) {

this.lexeme = lexeme;

this.isInit = isInit;

this.type = type;

}

}package com.ilyakrn.exceptions.internal;

public class InternalInterpreterException extends CompilerInternalException {

public InternalInterpreterException(String message) {

super("Interpreter internal error: " + message);

}

}

package com.ilyakrn.exceptions.internal;

public class InternalLexerException extends CompilerInternalException {

public InternalLexerException(String message) {

super("Lexer internal error: " + message);

}

}

package com.ilyakrn.exceptions.internal;

public class InternalParserException extends CompilerInternalException {

public InternalParserException(String message) {

super("Parser internal error: " + message);

}

}

package com.ilyakrn.entities;

import com.ilyakrn.entities.items.\*;

import java.util.ArrayList;

public final class InternalProgramPresentation {

public static final int serviceTableId = 0;

public static final int delimiterTableId = 1;

public static final int identifierTableId = 2;

public static final int numberTableId = 3;

public static final int polizTableId = 4;

public static final int polizPointerTableId = 5;

private final ArrayList<ServiceItem> serviceTable;

private final ArrayList<DelimiterItem> delimiterTable;

private final ArrayList<IdentifierItem> identifierTable;

private final ArrayList<NumberItem> numberTable;

private final ArrayList<BinOperationItem> binOperationTable;

private final ArrayList<PolizItem> polizTable;

private final ArrayList<PolizPointerItem> polizPointerTable;

private final ArrayList<LexemesSeqItem> lexemesSeqTable;

public InternalProgramPresentation(ArrayList<ServiceItem> serviceTable, ArrayList<DelimiterItem> delimiterTable, ArrayList<IdentifierItem> identifierTable, ArrayList<NumberItem> numberTable, ArrayList<LexemesSeqItem> lexemesSeqTable, ArrayList<BinOperationItem> binOperationTable, ArrayList<PolizItem> polizTable, ArrayList<PolizPointerItem> polizPointerTable) {

this.serviceTable = serviceTable;

this.delimiterTable = delimiterTable;

this.identifierTable = identifierTable;

this.numberTable = numberTable;

this.lexemesSeqTable = lexemesSeqTable;

this.binOperationTable = binOperationTable;

this.polizTable = polizTable;

this.polizPointerTable = polizPointerTable;

}

public ArrayList<PolizItem> getPolizTable() {

return polizTable;

}

public ArrayList<BinOperationItem> getBinOperationTable() {

return binOperationTable;

}

public ArrayList<ServiceItem> getServiceTable() {

return serviceTable;

}

public ArrayList<DelimiterItem> getDelimiterTable() {

return delimiterTable;

}

public ArrayList<IdentifierItem> getIdentifierTable() {

return identifierTable;

}

public ArrayList<NumberItem> getNumberTable() {

return numberTable;

}

public ArrayList<LexemesSeqItem> getLexemesSeqTable() {

return lexemesSeqTable;

}

public ArrayList<PolizPointerItem> getPolizPointerTable() {

return polizPointerTable;

}

@Override

public String toString() {

StringBuilder sb = new StringBuilder();

sb.append("LexerOutput:\nTables:\n");

sb.append("=================BIN\_OPS=================\n");

for (int i = 0; i < binOperationTable.size(); i++) {

sb.append(String.format("%s\t%s\t%s\t%s\t%s\n", i, binOperationTable.get(i).getOperation(), binOperationTable.get(i).getOperand1(), binOperationTable.get(i).getOperand2(), binOperationTable.get(i).getResult()));

}

sb.append("=================SERVICE=================\n");

for (int i = 0; i < serviceTable.size(); i++) {

sb.append(String.format("%s\t%s\n", i, serviceTable.get(i).getLexeme()));

}

sb.append("==================DELIM==================\n");

for (int i = 0; i < delimiterTable.size(); i++) {

sb.append(String.format("%s\t%s\n", i, delimiterTable.get(i).getLexeme()));

}

sb.append("==================IDENT==================\n");

for (int i = 0; i < identifierTable.size(); i++) {

sb.append(String.format("%s\t%s\t\t\t\t%s\t%s\n", i, identifierTable.get(i).getLexeme(), identifierTable.get(i).getType() != null ? identifierTable.get(i).getType().name() : "null", identifierTable.get(i).isInit()));

}

sb.append("===================NUM===================\n");

for (int i = 0; i < numberTable.size(); i++) {

sb.append(String.format("%s\t%s\t%s\t%s\n", i, numberTable.get(i).getLexeme(), numberTable.get(i).getType().name(), numberTable.get(i).getDimensionCount()));

}

sb.append("=================POINTER=================\n");

for (int i = 0; i < polizPointerTable.size(); i++) {

sb.append(String.format("%s\t%s\n", i, polizPointerTable.get(i).getPolizIndex()));

}

sb.append("==================LEXEMES==================\n");

for (int i = 0; i < lexemesSeqTable.size(); i++) {

String lexemeName = "";

String tableName = "";

switch (lexemesSeqTable.get(i).getTableId()){

case serviceTableId:

lexemeName = serviceTable.get(lexemesSeqTable.get(i).getLexId()).getLexeme();

tableName = "serv";

break;

case delimiterTableId:

lexemeName = delimiterTable.get(lexemesSeqTable.get(i).getLexId()).getLexeme();

tableName = "delim";

break;

case identifierTableId:

lexemeName = identifierTable.get(lexemesSeqTable.get(i).getLexId()).getLexeme();

tableName = "ident";

break;

case numberTableId:

lexemeName = numberTable.get(lexemesSeqTable.get(i).getLexId()).getLexeme();

tableName = "num ";

break;

}

sb.append(String.format("%s\t%s\t%s\t%s\t\t\t\t%s\n", i, lexemesSeqTable.get(i).getLexId(), lexemesSeqTable.get(i).getTableId(), tableName, lexemeName ));

}

sb.append("===================POLIZ===================\n");

for (int i = 0; i < polizTable.size(); i++) {

String lexemeName = "";

String tableName = "";

switch (polizTable.get(i).getTableId()){

case serviceTableId:

lexemeName = serviceTable.get(polizTable.get(i).getLexId()).getLexeme();

tableName = "serv";

break;

case delimiterTableId:

lexemeName = delimiterTable.get(polizTable.get(i).getLexId()).getLexeme();

tableName = "delim";

break;

case identifierTableId:

lexemeName = identifierTable.get(polizTable.get(i).getLexId()).getLexeme();

tableName = "ident";

break;

case numberTableId:

lexemeName = numberTable.get(polizTable.get(i).getLexId()).getLexeme();

tableName = "num ";

break;

case polizPointerTableId:

lexemeName = String.valueOf(polizPointerTable.get(polizTable.get(i).getLexId()).getPolizIndex());

tableName = "point";

break;

}

sb.append(String.format("%s\t%s\t\t%s\n", i, tableName, lexemeName));

}

return sb.toString();

}

}

package com.ilyakrn.exceptions.external;

public class InterpretationException extends CompilerExternalException {

public InterpretationException(String message) {

super("Interpretation error: " + message);

}

}

package com.ilyakrn.interpreter;

import com.ilyakrn.entities.InternalProgramPresentation;

import com.ilyakrn.entities.items.\*;

import com.ilyakrn.exceptions.external.InterpretationException;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.Scanner;

import java.util.Stack;

public class Interpreter {

private HashMap<String, MemoryItem> memory;

private InternalProgramPresentation internalProgramPresentation;

public void interpret(InternalProgramPresentation input) {

memory = new HashMap<>();

internalProgramPresentation = input;

for (int i = 0; i < internalProgramPresentation.getIdentifierTable().size(); i++) {

IdentifierItem item = internalProgramPresentation.getIdentifierTable().get(i);

String defaultValue = "";

switch (item.getType()) {

case INT:

defaultValue="0";

break;

case FLOAT:

defaultValue="0.0";

break;

case BOOL:

defaultValue="false";

break;

default:

throw new InterpretationException("Unknown item type: '" + item.getType() + "'");

}

memory.put(getMemoryAddress(InternalProgramPresentation.identifierTableId, i), new MemoryItem(item.getType(), defaultValue));

}

for (int i = 0; i < internalProgramPresentation.getNumberTable().size(); i++) {

NumberItem item = internalProgramPresentation.getNumberTable().get(i);

try {

if (item.getType() == Type.INT){

String originValue = item.getLexeme();

if(originValue.charAt(originValue.length() - 1) == 'b' ||

originValue.charAt(originValue.length() - 1) == 'o' ||

originValue.charAt(originValue.length() - 1) == 'd' ||

originValue.charAt(originValue.length() - 1) == 'h'

){

originValue = originValue.substring(0, originValue.length() - 1);

}

String value = "" + Integer.parseInt(originValue, item.getDimensionCount());

memory.put(getMemoryAddress(InternalProgramPresentation.numberTableId, i), new MemoryItem(Type.INT, value));

}

if (item.getType() == Type.FLOAT){

String originValue = item.getLexeme();

String value = "" + Float.parseFloat(originValue);

memory.put(getMemoryAddress(InternalProgramPresentation.numberTableId, i), new MemoryItem(Type.FLOAT, value));

}

} catch (NumberFormatException e) {

throw new InterpretationException("can not convert '" + item.getLexeme() + "' to decimal int");

}

}

for (int i = 0; i < internalProgramPresentation.getServiceTable().size(); i++) {

ServiceItem item = internalProgramPresentation.getServiceTable().get(i);

if (item.getLexeme() == "true" || item.getLexeme() == "false") {

memory.put(getMemoryAddress(InternalProgramPresentation.serviceTableId, i), new MemoryItem(Type.BOOL, item.getLexeme()));

}

}

for (int i = 0; i < internalProgramPresentation.getPolizPointerTable().size(); i++) {

PolizPointerItem item = internalProgramPresentation.getPolizPointerTable().get(i);

memory.put(getMemoryAddress(InternalProgramPresentation.polizPointerTableId, i), new MemoryItem(Type.INT, String.valueOf(item.getPolizIndex())));

}

Stack<InterpreterStackItem> execStack = new Stack<>();

for (int i = 0; i < internalProgramPresentation.getPolizTable().size(); i++) {

PolizItem currentPolizItem = internalProgramPresentation.getPolizTable().get(i);

if (i == internalProgramPresentation.getPolizTable().size() - 1){

if (!execStack.isEmpty())

throw new InterpretationException("unexpected end of program");

if (currentPolizItem.getTableId() != InternalProgramPresentation.delimiterTableId || !internalProgramPresentation.getDelimiterTable().get(currentPolizItem.getLexId()).getLexeme().equals("}"))

throw new InterpretationException("program must be terminated by '}'");

return;

}

if (currentPolizItem.getTableId() == InternalProgramPresentation.delimiterTableId) {

String operator = internalProgramPresentation.getDelimiterTable().get(currentPolizItem.getLexId()).getLexeme();

int operandsCount = getOperandsCount(operator);

if (operandsCount == 1) {

if (execStack.size() < 1)

throw new InterpretationException("execution stack size < 1 for 1-placed operator '" + internalProgramPresentation.getDelimiterTable().get(currentPolizItem.getLexId()).getLexeme() + "'");

InterpreterStackItem stackOperand = execStack.pop();

switch (operator){

case "!":

try {

i = Integer.parseInt(stackOperand.getValue()) - 1;

if (i < 0 || i >= internalProgramPresentation.getPolizTable().size())

throw new InterpretationException("poliz pointer out of bounds");

} catch (NumberFormatException e){

throw new InterpretationException("invalid poliz pointer value");

}

break;

case "R":

if (stackOperand.getMemoryAddress() == null)

throw new InterpretationException("memory to read not allocated");

Scanner scanner = new Scanner(System.in);

String value = scanner.nextLine();

switch (stackOperand.getType()){

case INT:

try{

int integerValue = Integer.parseInt(value);

MemoryItem memoryItem = memory.get(stackOperand.getMemoryAddress());

memoryItem.setValue(String.valueOf(integerValue));

memory.put(stackOperand.getMemoryAddress(), memoryItem);

}catch (NumberFormatException e){

throw new InterpretationException("invalid integer value");

}

break;

case FLOAT:

try{

float floatValue = Float.parseFloat(value);

MemoryItem memoryItem = memory.get(stackOperand.getMemoryAddress());

memoryItem.setValue(String.valueOf(floatValue));

memory.put(stackOperand.getMemoryAddress(), memoryItem);

}catch (NumberFormatException e){

throw new InterpretationException("invalid float value");

}

break;

case BOOL:

if (!value.equals("true") && !value.equals("false"))

throw new InterpretationException("can not convert '" + value + "' to bool");

MemoryItem memoryItem = memory.get(stackOperand.getMemoryAddress());

memoryItem.setValue(value);

memory.put(stackOperand.getMemoryAddress(), memoryItem);

break;

default:

throw new InterpretationException("unknown type '" + stackOperand.getType().name() + "'");

}

break;

case "W":

System.out.println(stackOperand.getValue());

break;

case "not":

execStack.push(new InterpreterStackItem(stackOperand.getValue().equals("true") ? "false" : "true", Type.BOOL, null));

break;

default:

throw new InterpretationException("Invalid poliz operator '" + internalProgramPresentation.getDelimiterTable().get(currentPolizItem.getLexId()).getLexeme() + "'");

}

} else if (operandsCount == 2) {

if (execStack.size() < 2)

throw new InterpretationException("execution stack size < 2 for 2-placed operator '" + internalProgramPresentation.getDelimiterTable().get(currentPolizItem.getLexId()).getLexeme() + "'");

InterpreterStackItem stackOperand2 = execStack.pop();

InterpreterStackItem stackOperand1 = execStack.pop();

switch (operator){

case ">":

try{

float floatValue1 = Float.parseFloat(stackOperand1.getValue());

float floatValue2 = Float.parseFloat(stackOperand2.getValue());

execStack.push(new InterpreterStackItem(floatValue1 > floatValue2 ? "true" : "false", getBinOpResultType(operator, stackOperand1.getType(), stackOperand2.getType()), null));

}catch (NumberFormatException e){

throw new InterpretationException("can apply '>' to '" + stackOperand1.getValue() + "' and '" + stackOperand2.getValue() + "'");

}

break;

case "<":

try{

float floatValue1 = Float.parseFloat(stackOperand1.getValue());

float floatValue2 = Float.parseFloat(stackOperand2.getValue());

execStack.push(new InterpreterStackItem(floatValue1 < floatValue2 ? "true" : "false", getBinOpResultType(operator, stackOperand1.getType(), stackOperand2.getType()), null));

}catch (NumberFormatException e){

throw new InterpretationException("can apply '<' to '" + stackOperand1.getValue() + "' and '" + stackOperand2.getValue() + "'");

}

break;

case "<>":

try{

String value1;

String value2;

if (stackOperand1.getType() == Type.BOOL && stackOperand2.getType() == Type.BOOL){

value1 = String.valueOf(Boolean.parseBoolean(stackOperand1.getValue()));

value2 = String.valueOf(Boolean.parseBoolean(stackOperand2.getValue()));

}

else {

value1 = String.valueOf(Float.parseFloat(stackOperand1.getValue()));

value2 = String.valueOf(Float.parseFloat(stackOperand2.getValue()));

}

execStack.push(new InterpreterStackItem(!value1.equals(value2) ? "true" : "false", getBinOpResultType(operator, stackOperand1.getType(), stackOperand2.getType()), null));

}catch (NumberFormatException e){

throw new InterpretationException("can apply '<>' to '" + stackOperand1.getValue() + "' and '" + stackOperand2.getValue() + "'");

}

break;

case ">=":

try{

float floatValue1 = Float.parseFloat(stackOperand1.getValue());

float floatValue2 = Float.parseFloat(stackOperand2.getValue());

execStack.push(new InterpreterStackItem(floatValue1 >= floatValue2 ? "true" : "false", getBinOpResultType(operator, stackOperand1.getType(), stackOperand2.getType()), null));

}catch (NumberFormatException e){

throw new InterpretationException("can apply '>=' to '" + stackOperand1.getValue() + "' and '" + stackOperand2.getValue() + "'");

}

break;

case "<=":

try{

float floatValue1 = Float.parseFloat(stackOperand1.getValue());

float floatValue2 = Float.parseFloat(stackOperand2.getValue());

execStack.push(new InterpreterStackItem(floatValue1 <= floatValue2 ? "true" : "false", getBinOpResultType(operator, stackOperand1.getType(), stackOperand2.getType()), null));

}catch (NumberFormatException e){

throw new InterpretationException("can apply '<=' to '" + stackOperand1.getValue() + "' and '" + stackOperand2.getValue() + "'");

}

break;

case "=":

try{

String value1;

String value2;

if (stackOperand1.getType() == Type.BOOL && stackOperand2.getType() == Type.BOOL){

value1 = String.valueOf(Boolean.parseBoolean(stackOperand1.getValue()));

value2 = String.valueOf(Boolean.parseBoolean(stackOperand2.getValue()));

}

else {

value1 = String.valueOf(Float.parseFloat(stackOperand1.getValue()));

value2 = String.valueOf(Float.parseFloat(stackOperand2.getValue()));

}

execStack.push(new InterpreterStackItem(value1.equals(value2) ? "true" : "false", getBinOpResultType(operator, stackOperand1.getType(), stackOperand2.getType()), null));

}catch (NumberFormatException e){

throw new InterpretationException("can apply '=' to '" + stackOperand1.getValue() + "' and '" + stackOperand2.getValue() + "'");

}

break;

case "+":

try{

float floatValue1 = Float.parseFloat(stackOperand1.getValue());

float floatValue2 = Float.parseFloat(stackOperand2.getValue());

String res = String.valueOf(floatValue1 + floatValue2);

if (getBinOpResultType(operator, stackOperand1.getType(), stackOperand2.getType()) == Type.INT) {

res = res.substring(0, res.length() - 2);

}

execStack.push(new InterpreterStackItem(res, getBinOpResultType(operator, stackOperand1.getType(), stackOperand2.getType()), null));

}catch (NumberFormatException e){

throw new InterpretationException("can apply '+' to '" + stackOperand1.getValue() + "' and '" + stackOperand2.getValue() + "'");

}

break;

case "-":

try{

float floatValue1 = Float.parseFloat(stackOperand1.getValue());

float floatValue2 = Float.parseFloat(stackOperand2.getValue());

String res = String.valueOf(floatValue1 - floatValue2);

if (getBinOpResultType(operator, stackOperand1.getType(), stackOperand2.getType()) == Type.INT) {

res = res.substring(0, res.length() - 2);

}

execStack.push(new InterpreterStackItem(res, getBinOpResultType(operator, stackOperand1.getType(), stackOperand2.getType()), null));

}catch (NumberFormatException e){

throw new InterpretationException("can apply '-' to '" + stackOperand1.getValue() + "' and '" + stackOperand2.getValue() + "'");

}

break;

case "\*":

try{

float floatValue1 = Float.parseFloat(stackOperand1.getValue());

float floatValue2 = Float.parseFloat(stackOperand2.getValue());

String res = String.valueOf(floatValue1 \* floatValue2);

if (getBinOpResultType(operator, stackOperand1.getType(), stackOperand2.getType()) == Type.INT) {

res = res.substring(0, res.length() - 2);

}

execStack.push(new InterpreterStackItem(res, getBinOpResultType(operator, stackOperand1.getType(), stackOperand2.getType()), null));

}catch (NumberFormatException e){

throw new InterpretationException("can apply '\*' to '" + stackOperand1.getValue() + "' and '" + stackOperand2.getValue() + "'");

}

break;

case "/":

try{

float floatValue1 = Float.parseFloat(stackOperand1.getValue());

float floatValue2 = Float.parseFloat(stackOperand2.getValue());

String res = String.valueOf(floatValue1 / floatValue2);

execStack.push(new InterpreterStackItem(res, getBinOpResultType(operator, stackOperand1.getType(), stackOperand2.getType()), null));

}catch (NumberFormatException e){

throw new InterpretationException("can apply '/' to '" + stackOperand1.getValue() + "' and '" + stackOperand2.getValue() + "'");

}

break;

case "or":

try{

boolean booleanValue1 = Boolean.parseBoolean(stackOperand1.getValue());

boolean booleanValue2 = Boolean.parseBoolean(stackOperand2.getValue());

String res = String.valueOf(booleanValue1 || booleanValue2);

execStack.push(new InterpreterStackItem(res, getBinOpResultType(operator, stackOperand1.getType(), stackOperand2.getType()), null));

}catch (NumberFormatException e){

throw new InterpretationException("can apply 'or' to '" + stackOperand1.getValue() + "' and '" + stackOperand2.getValue() + "'");

}

break;

case "and":

try{

boolean booleanValue1 = Boolean.parseBoolean(stackOperand1.getValue());

boolean booleanValue2 = Boolean.parseBoolean(stackOperand2.getValue());

String res = String.valueOf(booleanValue1 && booleanValue2);

execStack.push(new InterpreterStackItem(res, getBinOpResultType(operator, stackOperand1.getType(), stackOperand2.getType()), null));

}catch (NumberFormatException e){

throw new InterpretationException("can apply 'and' to '" + stackOperand1.getValue() + "' and '" + stackOperand2.getValue() + "'");

}

break;

case "ass":

if (stackOperand1.getMemoryAddress() == null)

throw new InterpretationException("memory to read not allocated");

MemoryItem memoryItem = memory.get(stackOperand1.getMemoryAddress());

memoryItem.setValue(String.valueOf(stackOperand2.getValue()));

memory.put(stackOperand1.getMemoryAddress(), memoryItem);

break;

case "!F":

try {

if (stackOperand1.getType() == Type.BOOL){

if (stackOperand1.getValue().equals("false")){

i = Integer.parseInt(stackOperand2.getValue()) - 1;

if (i < 0 || i >= internalProgramPresentation.getPolizTable().size())

throw new InterpretationException("poliz pointer out of bounds");

}

} else

throw new InterpretationException("'!F' can not apply to " + stackOperand1.getValue() + "' and '" + stackOperand2.getValue() + "'");

} catch (NumberFormatException e){

throw new InterpretationException("invalid poliz pointer value");

}

break;

default:

throw new InterpretationException("Invalid poliz operator " + internalProgramPresentation.getDelimiterTable().get(currentPolizItem.getLexId()).getLexeme());

}

} else {

throw new InterpretationException("operator with '" + operandsCount + "' operands are not allowed. operator '" + internalProgramPresentation.getDelimiterTable().get(currentPolizItem.getLexId()).getLexeme() + "'");

}

} else {

execStack.push(new InterpreterStackItem(getValueById(currentPolizItem.getTableId(), currentPolizItem.getLexId()), getTypeById(currentPolizItem.getTableId(), currentPolizItem.getLexId()), getMemoryAddress(currentPolizItem.getTableId(), currentPolizItem.getLexId())));

}

}

}

Type getTypeById(int tableId, int lexId){

switch (tableId){

case InternalProgramPresentation.identifierTableId:

if (lexId >= internalProgramPresentation.getIdentifierTable().size())

throw new InterpretationException("identifier table id out of bounds");

return internalProgramPresentation.getIdentifierTable().get(lexId).getType();

case InternalProgramPresentation.numberTableId:

if (lexId >= internalProgramPresentation.getNumberTable().size())

throw new InterpretationException("number table id out of bounds");

return internalProgramPresentation.getNumberTable().get(lexId).getType();

case InternalProgramPresentation.serviceTableId:

if (lexId >= internalProgramPresentation.getServiceTable().size())

throw new InterpretationException("servide table id out of bounds");

if (!internalProgramPresentation.getServiceTable().get(lexId).getLexeme().equals("true") && !internalProgramPresentation.getServiceTable().get(lexId).getLexeme().equals("false"))

throw new InterpretationException("can not ger type of '" + internalProgramPresentation.getServiceTable().get(lexId).getLexeme() + "'");

return Type.BOOL;

case InternalProgramPresentation.polizPointerTableId:

return Type.INT;

default:

throw new InterpretationException("can not get type of lexeme");

}

}

String getValueById(int tableId, int lexId){

return memory.get(getMemoryAddress(tableId, lexId)).getValue();

}

String getMemoryAddress(int tableId, int lexemeId) {

return tableId + ":" + lexemeId;

}

int getOperandsCount(String operator) {

switch (operator) {

case ">":

case "<":

case "<>":

case ">=":

case "<=":

case "=":

case "+":

case "-":

case "\*":

case "/":

case "or":

case "and":

case "ass":

case "!F":

return 2;

case "!":

case "R":

case "W":

case "not":

return 1;

default:

throw new InterpretationException("Invalid poliz operator '" + operator + "'");

}

}

Type getBinOpResultType(String operator, Type type1, Type type2) {

for (BinOperationItem binOperationItem : internalProgramPresentation.getBinOperationTable()) {

if (binOperationItem.getOperation().equals(operator) && binOperationItem.getOperand1() == type1 && binOperationItem.getOperand2() == type2)

return binOperationItem.getResult();

}

throw new InterpretationException("can not get result type for '" + type1 + "' '" + operator + "' '" + type2 + "'");

}

}

package com.ilyakrn.entities.items;

public class InterpreterStackItem {

private final String value;

private final Type type;

private final String memoryAddress;

public InterpreterStackItem(String value, Type type, String memoryAddress) {

this.value = value;

this.type = type;

this.memoryAddress = memoryAddress;

}

public String getValue() {

return value;

}

public Type getType() {

return type;

}

public String getMemoryAddress() {

return memoryAddress;

}

}

package com.ilyakrn.entities.items;

public class LexemesSeqItem {

private final int tableId;

private final int lexId;

private final int row;

private final int col;

public LexemesSeqItem(int tableId, int lexId, int row, int col) {

this.tableId = tableId;

this.lexId = lexId;

this.row = row;

this.col = col;

}

public int getTableId() {

return tableId;

}

public int getLexId() {

return lexId;

}

public int getRow() {

return row;

}

public int getCol() {

return col;

}

}

package com.ilyakrn.lexer;

import com.ilyakrn.entities.InternalProgramPresentation;

import com.ilyakrn.entities.items.\*;

import com.ilyakrn.exceptions.external.LexicalException;

import com.ilyakrn.exceptions.internal.InternalLexerException;

import java.util.ArrayList;

import java.util.LinkedList;

import java.util.Queue;

public class Lexer {

private enum STATE {

READ,

IDENT,

NUM\_BIN, NUM\_BIN\_FIN,

NUM\_OCT, NUM\_OCT\_FIN,

NUM\_DEC, NUM\_DEC\_FIN,

NUM\_HEX, NUM\_HEX\_FIN,

NUM\_REAL\_POINT\_1, NUM\_REAL\_POINT\_2, NUM\_REAL\_POINT\_ORDER\_START\_1, NUM\_REAL\_POINT\_ORDER\_START\_2, NUM\_REAL\_POINT\_ORDER,

NUM\_REAL\_ORDER\_OR\_HEX, NUM\_REAL\_ORDER,

DELIMITER,

COMMENT,

COMMENT\_START,

COMMENT\_END,

MORE\_THEN\_EQUAL,

LESS\_THEN\_EQUAL,

END,

ERROR,

}

private String lexBuffer;

private STATE currentState;

private Character currentChar;

private int curLexId;

private int curTableId;

private int curRow;

private int curCol;

private final ArrayList<ServiceItem> serviceTable;

private final ArrayList<DelimiterItem> delimiterTable;

private final ArrayList<IdentifierItem> identifierTable;

private final ArrayList<NumberItem> numberTable;

private final ArrayList<LexemesSeqItem> lexemesSeqTable;

private final Queue<Character> input;

public Lexer() {

input = new LinkedList<>();

serviceTable = new ArrayList<>();

delimiterTable = new ArrayList<>();

identifierTable = new ArrayList<>();

numberTable = new ArrayList<>();

lexemesSeqTable = new ArrayList<>();

curRow = 1;

curCol = 1;

serviceTable.add(new ServiceItem("true"));

serviceTable.add(new ServiceItem("false"));

serviceTable.add(new ServiceItem("int"));

serviceTable.add(new ServiceItem("float"));

serviceTable.add(new ServiceItem("bool"));

serviceTable.add(new ServiceItem("if"));

serviceTable.add(new ServiceItem("then"));

serviceTable.add(new ServiceItem("else"));

serviceTable.add(new ServiceItem("for"));

serviceTable.add(new ServiceItem("while"));

serviceTable.add(new ServiceItem("to"));

serviceTable.add(new ServiceItem("do"));

serviceTable.add(new ServiceItem("read"));

serviceTable.add(new ServiceItem("write"));

delimiterTable.add(new DelimiterItem("\n"));

delimiterTable.add(new DelimiterItem(";"));

delimiterTable.add(new DelimiterItem(":"));

delimiterTable.add(new DelimiterItem(","));

delimiterTable.add(new DelimiterItem(">"));

delimiterTable.add(new DelimiterItem("<"));

delimiterTable.add(new DelimiterItem("<>"));

delimiterTable.add(new DelimiterItem(">="));

delimiterTable.add(new DelimiterItem("<="));

delimiterTable.add(new DelimiterItem("="));

delimiterTable.add(new DelimiterItem("+"));

delimiterTable.add(new DelimiterItem("-"));

delimiterTable.add(new DelimiterItem("\*"));

delimiterTable.add(new DelimiterItem("/"));

delimiterTable.add(new DelimiterItem("or"));

delimiterTable.add(new DelimiterItem("and"));

delimiterTable.add(new DelimiterItem("not"));

delimiterTable.add(new DelimiterItem("ass"));

delimiterTable.add(new DelimiterItem("("));

delimiterTable.add(new DelimiterItem(")"));

delimiterTable.add(new DelimiterItem("{"));

delimiterTable.add(new DelimiterItem("}"));

}

private boolean isBinAllow(){

return currentChar >= '0' && currentChar <= '1';

}

private boolean isOctAllow(){

return currentChar >= '0' && currentChar <= '7';

}

private boolean isDecAllow(){

return currentChar >= '0' && currentChar <= '9';

}

private boolean isHexAllow(){

return currentChar >= '0' && currentChar <= '9' || currentChar >= 'A' && currentChar <= 'F' || currentChar >= 'a' && currentChar <= 'f';

}

private boolean isNumber(){

return currentChar >= '0' && currentChar <= '9';

}

private boolean isLetter(){

return currentChar >= 'A' && currentChar <= 'Z' || currentChar >= 'a' && currentChar <= 'z';

}

private void clean(){

lexBuffer = "";

}

private void add(){

lexBuffer += currentChar;

}

private void read() {

currentChar = input.poll();

if (currentChar == '\n'){

curRow++;

curCol = 1;

}

else {

curCol++;

}

}

private void check(int tableId) throws InternalLexerException {

switch (tableId){

case InternalProgramPresentation.serviceTableId:

curTableId = tableId;

for (int i = 0; i < serviceTable.size(); i++) {

if(serviceTable.get(i).getLexeme().equals(lexBuffer)){

curLexId = i;

return;

}

}

curLexId = -1;

break;

case InternalProgramPresentation.delimiterTableId:

curTableId = tableId;

for (int i = 0; i < delimiterTable.size(); i++) {

if(delimiterTable.get(i).getLexeme().equals(lexBuffer)){

curLexId = i;

return;

}

}

curLexId = -1;

break;

case InternalProgramPresentation.identifierTableId:

curTableId = tableId;

for (int i = 0; i < identifierTable.size(); i++) {

if(identifierTable.get(i).getLexeme().equals(lexBuffer)){

curLexId = i;

return;

}

}

curLexId = -1;

break;

case InternalProgramPresentation.numberTableId:

curTableId = tableId;

for (int i = 0; i < numberTable.size(); i++) {

if(numberTable.get(i).getLexeme().equals(lexBuffer)){

curLexId = i;

return;

}

}

curLexId = -1;

break;

default:

throw new InternalLexerException("table with id '" + tableId + "' not exists");

}

}

private void put(int tableId, Type numType, int numDimensionCount) throws InternalLexerException {

switch (tableId){

case InternalProgramPresentation.serviceTableId:

case InternalProgramPresentation.delimiterTableId:

throw new InternalLexerException("table with id '" + tableId + "' can not modify");

case InternalProgramPresentation.identifierTableId:

curTableId = tableId;

int findId = -1;

for (int i = 0; i < identifierTable.size(); i++) {

if(identifierTable.get(i).getLexeme().equals(lexBuffer)){

findId = i;

break;

}

}

if(findId == -1){

identifierTable.add(new IdentifierItem(lexBuffer, false, null));

curLexId = identifierTable.size() - 1;

}

else {

curLexId = findId;

}

break;

case InternalProgramPresentation.numberTableId:

curTableId = tableId;

findId = -1;

for (int i = 0; i < numberTable.size(); i++) {

if(numberTable.get(i).getLexeme().equals(lexBuffer)){

findId = i;

break;

}

}

if(findId == -1){

numberTable.add(new NumberItem(lexBuffer, numDimensionCount, numType));

curLexId = numberTable.size() - 1;

}

else {

curLexId = findId;

}

break;

default:

throw new InternalLexerException("table with id '" + tableId + "' not exists");

}

}

private void write(int tableId, int lexId) throws InternalLexerException {

switch (tableId){

case InternalProgramPresentation.serviceTableId:

if (lexId >= serviceTable.size() || lexId < 0)

throw new InternalLexerException("lexeme in table '" + tableId + "' with id '" + lexId + "' not exists");

lexemesSeqTable.add(new LexemesSeqItem(tableId, lexId, curRow, curCol));

break;

case InternalProgramPresentation.delimiterTableId:

if (lexId >= delimiterTable.size() || lexId < 0)

throw new InternalLexerException("lexeme in table '" + tableId + "' with id '" + lexId + "' not exists");

lexemesSeqTable.add(new LexemesSeqItem(tableId, lexId, curRow, curCol));

break;

case InternalProgramPresentation.identifierTableId:

if (lexId >= identifierTable.size() || lexId < 0)

throw new InternalLexerException("lexeme in table '" + tableId + "' with id '" + lexId + "' not exists");

lexemesSeqTable.add(new LexemesSeqItem(tableId, lexId, curRow, curCol));

break;

case InternalProgramPresentation.numberTableId:

if (lexId >= numberTable.size() || lexId < 0)

throw new InternalLexerException("lexeme in table '" + tableId + "' with id '" + lexId + "' not exists");

lexemesSeqTable.add(new LexemesSeqItem(tableId, lexId, curRow, curCol));

break;

default:

throw new InternalLexerException("table with id '" + tableId + "' not exists");

}

}

public InternalProgramPresentation analyze(String progText) throws InternalLexerException, LexicalException {

identifierTable.clear();

numberTable.clear();

lexemesSeqTable.clear();

input.clear();

lexBuffer = "";

currentChar = null;

String message = "";

char[] progChars = progText.toCharArray();

for (int i = 0; i < progChars.length; i++) {

input.add(progChars[i]);

}

currentState = STATE.READ;

while (currentState != STATE.END && currentState != STATE.ERROR) {

switch (currentState) {

case READ:

read();

if(currentChar == null){

message = "program end reached before '}'";

currentState = STATE.ERROR;

} else if (currentChar == ' ') {

clean();

} else if (isBinAllow()) {

currentState = STATE.NUM\_BIN;

clean();

add();

} else if (isOctAllow()) {

currentState = STATE.NUM\_OCT;

clean();

add();

} else if (isDecAllow()) {

currentState = STATE.NUM\_DEC;

clean();

add();

} else if (isLetter()) {

currentState = STATE.IDENT;

clean();

add();

} else if (currentChar == '.') {

currentState = STATE.NUM\_REAL\_POINT\_1;

clean();

add();

} else {

currentState = STATE.DELIMITER;

clean();

add();

}

break;

case NUM\_BIN:

read();

if(currentChar == null){

message = "program end reached before '}'";

currentState = STATE.ERROR;

} else if (isBinAllow()) {

add();

} else if (currentChar == '.') {

currentState = STATE.NUM\_REAL\_POINT\_1;

add();

} else if (currentChar == 'E' || currentChar == 'e') {

currentState = STATE.NUM\_REAL\_ORDER\_OR\_HEX;

add();

} else if (currentChar == 'B' || currentChar == 'b') {

currentState = STATE.NUM\_BIN\_FIN;

add();

} else if (currentChar == 'O' || currentChar == 'o') {

currentState = STATE.NUM\_OCT\_FIN;

add();

} else if (currentChar == 'D' || currentChar == 'd') {

currentState = STATE.NUM\_DEC\_FIN;

add();

} else if (currentChar == 'H' || currentChar == 'h') {

currentState = STATE.NUM\_HEX\_FIN;

add();

} else if (isOctAllow()) {

currentState = STATE.NUM\_OCT;

add();

} else if (isDecAllow()) {

currentState = STATE.NUM\_DEC;

add();

} else if (isHexAllow()) {

currentState = STATE.NUM\_HEX;

add();

} else if (isLetter()) {

message = "unresolved character '" + currentChar + "'";

currentState = STATE.ERROR;

} else {

currentState = STATE.DELIMITER;

put(InternalProgramPresentation.numberTableId, Type.INT, 10);

write(InternalProgramPresentation.numberTableId, curLexId);

clean();

add();

}

break;

case NUM\_OCT:

read();

if(currentChar == null){

message = "program end reached before '}'";

currentState = STATE.ERROR;

} else if (isOctAllow()) {

add();

} else if (currentChar == '.') {

currentState = STATE.NUM\_REAL\_POINT\_1;

add();

} else if (currentChar == 'E' || currentChar == 'e') {

currentState = STATE.NUM\_REAL\_ORDER\_OR\_HEX;

add();

} else if (currentChar == 'O' || currentChar == 'o') {

currentState = STATE.NUM\_OCT\_FIN;

add();

} else if (currentChar == 'D' || currentChar == 'd') {

currentState = STATE.NUM\_DEC\_FIN;

add();

} else if (currentChar == 'H' || currentChar == 'h') {

currentState = STATE.NUM\_HEX\_FIN;

add();

} else if (isDecAllow()) {

currentState = STATE.NUM\_DEC;

add();

} else if (isHexAllow()) {

currentState = STATE.NUM\_HEX;

add();

} else if (isLetter()) {

message = "unresolved character '" + currentChar + "'";

currentState = STATE.ERROR;

} else {

currentState = STATE.DELIMITER;

put(InternalProgramPresentation.numberTableId, Type.INT, 10);

write(InternalProgramPresentation.numberTableId, curLexId);

clean();

add();

}

break;

case NUM\_DEC:

read();

if(currentChar == null){

message = "program end reached before '}'";

currentState = STATE.ERROR;

} else if (isDecAllow()) {

add();

} else if (currentChar == '.') {

currentState = STATE.NUM\_REAL\_POINT\_1;

add();

} else if (currentChar == 'E' || currentChar == 'e') {

currentState = STATE.NUM\_REAL\_ORDER\_OR\_HEX;

add();

} else if (currentChar == 'D' || currentChar == 'd') {

currentState = STATE.NUM\_DEC\_FIN;

add();

} else if (currentChar == 'H' || currentChar == 'h') {

currentState = STATE.NUM\_HEX\_FIN;

add();

} else if (isHexAllow()) {

currentState = STATE.NUM\_HEX;

add();

} else if (isLetter()) {

message = "unresolved character '" + currentChar + "'";

currentState = STATE.ERROR;

} else {

currentState = STATE.DELIMITER;

put(InternalProgramPresentation.numberTableId, Type.INT, 10);

write(InternalProgramPresentation.numberTableId, curLexId);

clean();

add();

}

break;

case NUM\_HEX:

read();

if(currentChar == null){

message = "program end reached before '}'";

currentState = STATE.ERROR;

} else if (isHexAllow()) {

add();

} else if (currentChar == 'H' || currentChar == 'h') {

currentState = STATE.NUM\_HEX\_FIN;

add();

} else {

message = "unresolved character '" + currentChar + "'";

currentState = STATE.ERROR;

}

break;

case IDENT:

read();

if(currentChar == null){

message = "program end reached before '}'";

currentState = STATE.ERROR;

} else if (isLetter()) {

add();

} else if (isNumber()) {

add();

} else {

currentState = STATE.DELIMITER;

check(InternalProgramPresentation.serviceTableId);

if (curLexId == -1){

check(InternalProgramPresentation.delimiterTableId);

if (curLexId == -1){

put(InternalProgramPresentation.identifierTableId, null, 0);

write(InternalProgramPresentation.identifierTableId, curLexId);

}

else {

write(InternalProgramPresentation.delimiterTableId, curLexId);

}

}

else {

write(InternalProgramPresentation.serviceTableId, curLexId);

}

clean();

add();

}

break;

case NUM\_BIN\_FIN:

read();

if(currentChar == null){

message = "program end reached before '}'";

currentState = STATE.ERROR;

} else if (currentChar == 'H' || currentChar == 'h') {

currentState = STATE.NUM\_HEX\_FIN;

add();

} else if (isHexAllow()) {

currentState = STATE.NUM\_HEX;

add();

} else if (isLetter() || isNumber()) {

message = "unresolved character '" + currentChar + "'";

currentState = STATE.ERROR;

} else {

currentState = STATE.DELIMITER;

put(InternalProgramPresentation.numberTableId, Type.INT, 2);

write(InternalProgramPresentation.numberTableId, curLexId);

clean();

add();

}

break;

case NUM\_DEC\_FIN:

read();

if(currentChar == null){

message = "program end reached before '}'";

currentState = STATE.ERROR;

} else if (currentChar == 'H' || currentChar == 'h') {

currentState = STATE.NUM\_HEX\_FIN;

add();

} else if (isHexAllow()) {

currentState = STATE.NUM\_HEX;

add();

} else if (isLetter() || isNumber()) {

message = "unresolved character '" + currentChar + "'";

currentState = STATE.ERROR;

} else {

currentState = STATE.DELIMITER;

put(InternalProgramPresentation.numberTableId, Type.INT, 10);

write(InternalProgramPresentation.numberTableId, curLexId);

clean();

add();

}

break;

case NUM\_OCT\_FIN:

read();

if(currentChar == null){

message = "program end reached before '}'";

currentState = STATE.ERROR;

} else if (isLetter() || isNumber()) {

message = "unresolved character '" + currentChar + "'";

currentState = STATE.ERROR;

} else {

currentState = STATE.DELIMITER;

put(InternalProgramPresentation.numberTableId, Type.INT, 8);

write(InternalProgramPresentation.numberTableId, curLexId);

clean();

add();

}

break;

case NUM\_HEX\_FIN:

read();

if(currentChar == null){

message = "program end reached before '}'";

currentState = STATE.ERROR;

} else if (isLetter() || isNumber()) {

message = "unresolved character '" + currentChar + "'";

currentState = STATE.ERROR;

} else {

currentState = STATE.DELIMITER;

put(InternalProgramPresentation.numberTableId, Type.INT, 16);

write(InternalProgramPresentation.numberTableId, curLexId);

clean();

add();

}

break;

case NUM\_REAL\_POINT\_1:

read();

if(currentChar == null){

message = "program end reached before '}'";

currentState = STATE.ERROR;

} else if (isNumber()) {

currentState = STATE.NUM\_REAL\_POINT\_2;

add();

} else {

message = "unresolved character '" + currentChar + "'";

currentState = STATE.ERROR;

}

break;

case NUM\_REAL\_POINT\_2:

read();

if(currentChar == null){

message = "program end reached before '}'";

currentState = STATE.ERROR;

} else if (isNumber()) {

add();

} else if (currentChar == 'E' || currentChar == 'e') {

currentState = STATE.NUM\_REAL\_POINT\_ORDER\_START\_1;

add();

} else if (isLetter() || currentChar == '.') {

message = "unresolved character '" + currentChar + "'";

currentState = STATE.ERROR;

} else {

currentState = STATE.DELIMITER;

put(InternalProgramPresentation.numberTableId, Type.FLOAT, -1);

write(InternalProgramPresentation.numberTableId, curLexId);

clean();

add();

}

break;

case NUM\_REAL\_POINT\_ORDER\_START\_1:

read();

if(currentChar == null){

message = "can not resolve real order";

currentState = STATE.ERROR;

} else if (isNumber()) {

currentState = STATE.NUM\_REAL\_POINT\_ORDER;

add();

} else if (currentChar == '+' || currentChar == '-') {

currentState = STATE.NUM\_REAL\_POINT\_ORDER\_START\_2;

add();

} else if (isLetter() || currentChar == '.') {

message = "unresolved character '" + currentChar + "'";

currentState = STATE.ERROR;

} else {

currentState = STATE.DELIMITER;

put(InternalProgramPresentation.numberTableId, Type.FLOAT, -1);

write(InternalProgramPresentation.numberTableId, curLexId);

clean();

add();

}

break;

case NUM\_REAL\_POINT\_ORDER\_START\_2:

read();

if(currentChar == null){

message = "can not resolve real order";

currentState = STATE.ERROR;

} else if (isNumber()) {

currentState = STATE.NUM\_REAL\_POINT\_ORDER;

add();

} else if (isLetter() || currentChar == '.') {

message = "unresolved character '" + currentChar + "'";

currentState = STATE.ERROR;

} else {

currentState = STATE.DELIMITER;

put(InternalProgramPresentation.numberTableId, Type.FLOAT, -1);

write(InternalProgramPresentation.numberTableId, curLexId);

clean();

add();

}

break;

case NUM\_REAL\_POINT\_ORDER:

read();

if(currentChar == null){

message = "program end reached before '}'";

currentState = STATE.ERROR;

} else if (isNumber()) {

add();

} else if (isLetter() || currentChar == '.') {

message = "unresolved character '" + currentChar + "'";

currentState = STATE.ERROR;

} else {

currentState = STATE.DELIMITER;

put(InternalProgramPresentation.numberTableId, Type.FLOAT, -1);

write(InternalProgramPresentation.numberTableId, curLexId);

clean();

add();

}

break;

case NUM\_REAL\_ORDER\_OR\_HEX:

read();

if(currentChar == null){

message = "program end reached before '}'";

currentState = STATE.ERROR;

} else if (isNumber()) {

add();

} else if (isHexAllow()) {

currentState = STATE.NUM\_HEX;

} else if (currentChar == '+' || currentChar == '-') {

currentState = STATE.NUM\_REAL\_ORDER;

add();

} else if (currentChar == 'H' || currentChar == 'h') {

currentState = STATE.NUM\_HEX\_FIN;

add();

} else if (isLetter() || currentChar == '.') {

message = "unresolved character '" + currentChar + "'";

currentState = STATE.ERROR;

} else {

currentState = STATE.DELIMITER;

put(InternalProgramPresentation.numberTableId, Type.FLOAT, -1);

write(InternalProgramPresentation.numberTableId, curLexId);

clean();

add();

}

break;

case NUM\_REAL\_ORDER:

read();

if(currentChar == null){

message = "program end reached before '}'";

currentState = STATE.ERROR;

} else if (isNumber()) {

add();

} else if (isLetter() || currentChar == '.' || currentChar == '+' || currentChar == '-') {

message = "unresolved character '" + currentChar + "'";

currentState = STATE.ERROR;

} else {

currentState = STATE.DELIMITER;

put(InternalProgramPresentation.numberTableId, Type.FLOAT, -1);

write(InternalProgramPresentation.numberTableId, curLexId);

clean();

add();

}

break;

case DELIMITER:

if(currentChar == null){

message = "program end reached before '}'";

currentState = STATE.ERROR;

}

else if(currentChar == ' '){

currentState = STATE.READ;

clean();

}

else if (currentChar == '>') {

currentState = STATE.MORE\_THEN\_EQUAL;

}

else if (currentChar == '<') {

currentState = STATE.LESS\_THEN\_EQUAL;

}

else if (currentChar == '/') {

currentState = STATE.COMMENT\_START;

}

else {

check(InternalProgramPresentation.delimiterTableId);

if (curLexId == -1){

currentState = STATE.ERROR;

message = "unresolved character '" + currentChar + "'";

}

else {

if (currentChar == '}'){

currentState = STATE.END;

}

else {

currentState = STATE.READ;

}

write(InternalProgramPresentation.delimiterTableId, curLexId);

clean();

}

}

break;

case MORE\_THEN\_EQUAL:

case LESS\_THEN\_EQUAL:

read();

if(currentChar == null){

message = "program end reached before '}'";

currentState = STATE.ERROR;

} else if(currentChar == '='){

add();

check(InternalProgramPresentation.delimiterTableId);

if(curLexId == -1){

currentState = STATE.ERROR;

message = "unresolved character '" + currentChar + "'";

}

else {

currentState = STATE.READ;

write(InternalProgramPresentation.delimiterTableId, curLexId);

clean();

}

} else if(currentChar == '>'){

add();

check(InternalProgramPresentation.delimiterTableId);

if(curLexId == -1){

currentState = STATE.ERROR;

message = "unresolved character '" + currentChar + "'";

}

else {

currentState = STATE.READ;

write(InternalProgramPresentation.delimiterTableId, curLexId);

clean();

}

} else if (isBinAllow()) {

currentState = STATE.NUM\_BIN;

check(InternalProgramPresentation.delimiterTableId);

if(curLexId == -1){

currentState = STATE.ERROR;

message = "unresolved character '" + currentChar + "'";

}

else {

write(InternalProgramPresentation.delimiterTableId, curLexId);

}

clean();

add();

} else if (isOctAllow()) {

currentState = STATE.NUM\_OCT;

check(InternalProgramPresentation.delimiterTableId);

if(curLexId == -1){

currentState = STATE.ERROR;

message = "unresolved character '" + currentChar + "'";

}

else {

write(InternalProgramPresentation.delimiterTableId, curLexId);

}

clean();

add();

} else if (isDecAllow()) {

currentState = STATE.NUM\_DEC;

check(InternalProgramPresentation.delimiterTableId);

if(curLexId == -1){

currentState = STATE.ERROR;

message = "unresolved character '" + currentChar + "'";

}

else {

write(InternalProgramPresentation.delimiterTableId, curLexId);

}

clean();

add();

} else if (isLetter()) {

currentState = STATE.IDENT;

check(InternalProgramPresentation.delimiterTableId);

if(curLexId == -1){

currentState = STATE.ERROR;

message = "unresolved character '" + currentChar + "'";

}

else {

write(InternalProgramPresentation.delimiterTableId, curLexId);

}

clean();

add();

} else if (currentChar == '.') {

currentState = STATE.NUM\_REAL\_POINT\_1;

if(curLexId == -1){

currentState = STATE.ERROR;

message = "unresolved character '" + currentChar + "'";

}

else {

write(InternalProgramPresentation.delimiterTableId, curLexId);

}

clean();

add();

} else {

check(InternalProgramPresentation.delimiterTableId);

if(curLexId == -1){

currentState = STATE.ERROR;

message = "unresolved character '" + currentChar + "'";

}

else {

currentState = STATE.DELIMITER;

write(InternalProgramPresentation.delimiterTableId, curLexId);

clean();

add();

}

}

break;

case COMMENT\_START:

read();

if(currentChar == null){

message = "program end reached before '}'";

currentState = STATE.ERROR;

} else if(currentChar == ' '){

currentState = STATE.READ;

check(InternalProgramPresentation.delimiterTableId);

if(curLexId == -1){

currentState = STATE.ERROR;

message = "unresolved character '" + currentChar + "'";

}

else {

write(InternalProgramPresentation.delimiterTableId, curLexId);

}

clean();

} else if(currentChar == '\*'){

currentState = STATE.COMMENT;

clean();

} else if (isBinAllow()) {

currentState = STATE.NUM\_BIN;

check(InternalProgramPresentation.delimiterTableId);

if(curLexId == -1){

currentState = STATE.ERROR;

message = "unresolved character '" + currentChar + "'";

}

else {

write(InternalProgramPresentation.delimiterTableId, curLexId);

}

clean();

add();

} else if (isOctAllow()) {

currentState = STATE.NUM\_OCT;

check(InternalProgramPresentation.delimiterTableId);

if(curLexId == -1){

currentState = STATE.ERROR;

message = "unresolved character '" + currentChar + "'";

}

else {

write(InternalProgramPresentation.delimiterTableId, curLexId);

}

clean();

add();

} else if (isDecAllow()) {

currentState = STATE.NUM\_DEC;

check(InternalProgramPresentation.delimiterTableId);

if(curLexId == -1){

currentState = STATE.ERROR;

message = "unresolved character '" + currentChar + "'";

}

else {

write(InternalProgramPresentation.delimiterTableId, curLexId);

}

clean();

add();

} else if (isLetter()) {

currentState = STATE.IDENT;

check(InternalProgramPresentation.delimiterTableId);

if(curLexId == -1){

currentState = STATE.ERROR;

message = "unresolved character '" + currentChar + "'";

}

else {

write(InternalProgramPresentation.delimiterTableId, curLexId);

}

clean();

add();

} else if (currentChar == '.') {

currentState = STATE.NUM\_REAL\_POINT\_1;

if(curLexId == -1){

currentState = STATE.ERROR;

message = "unresolved character '" + currentChar + "'";

}

else {

write(InternalProgramPresentation.delimiterTableId, curLexId);

}

clean();

add();

} else {

check(InternalProgramPresentation.delimiterTableId);

if(curLexId == -1){

currentState = STATE.ERROR;

message = "unresolved character '" + currentChar + "'";

}

else {

currentState = STATE.DELIMITER;

write(InternalProgramPresentation.delimiterTableId, curLexId);

clean();

add();

}

}

break;

case COMMENT:

read();

add();

if(currentChar == null){

message = "program end reached before '}'";

currentState = STATE.ERROR;

} else if(currentChar == '\*') {

currentState = STATE.COMMENT\_END;

}

break;

case COMMENT\_END:

read();

if(currentChar == null){

message = "program end reached before '}'";

currentState = STATE.ERROR;

} else if(currentChar == '/') {

currentState = STATE.READ;

clean();

}

else {

currentState = STATE.COMMENT;

add();

}

break;

default:

throw new InternalLexerException("can not resolve state '" + currentState.name() + "'");

}

}

if (currentState == STATE.ERROR) {

throw new LexicalException(curRow + ":" + (curCol - 1) + "\t" + message);

}

return new InternalProgramPresentation(serviceTable, delimiterTable, identifierTable, numberTable, lexemesSeqTable, new ArrayList<>(), new ArrayList(), new ArrayList());

}

}

package com.ilyakrn.exceptions.external;

public class LexicalException extends CompilerExternalException {

public LexicalException(String message) {

super("Lexical error: " + message);

}

}

package com.ilyakrn;

import com.ilyakrn.exceptions.external.CompilerExternalException;

import com.ilyakrn.exceptions.internal.CompilerInternalException;

import com.ilyakrn.exceptions.internal.InternalLexerException;

import com.ilyakrn.interpreter.Interpreter;

import com.ilyakrn.lexer.Lexer;

import com.ilyakrn.entities.InternalProgramPresentation;

import com.ilyakrn.parser.Parser;

import java.nio.charset.StandardCharsets;

import java.nio.file.Files;

import java.nio.file.Paths;

import java.util.ArrayList;

import java.util.Arrays;

public class Main {

public static void main(String[] args) throws Exception {

String filePath = "main.prog";

String compileFlag = "-i";

if (args.length > 0) {

switch (args[0]){

case "-i":

compileFlag = "-i";

break;

case "-l":

compileFlag = "-l";

break;

case "-p":

compileFlag = "-p";

break;

default:

System.err.println("Invalid compiler flag '" + args[0] + "'\nallowed flags:\n'-i' - compile and interpret\n'-l' - print compiled program presentation\n'-p' - print POLIZ of program presentation\nalso you can set file to compile as second argument");

return;

}

}

if (args.length > 1) {

filePath = args[1];

}

String prog = new String(Files.readAllBytes(Paths.get(filePath)), StandardCharsets.UTF\_8);

Lexer lexer = new Lexer();

Parser parser = new Parser();

Interpreter interpreter = new Interpreter();

InternalProgramPresentation internalProgramPresentation = null;

try {

System.out.println("==================COMPILER==================");

internalProgramPresentation = lexer.analyze(prog);

System.out.println("LEXER COMPLETE SUCCESSFUL");

internalProgramPresentation = parser.analyze(internalProgramPresentation);

System.out.println("PARSER COMPLETE SUCCESSFUL");

if (compileFlag.equals("-l")) {

System.out.println(internalProgramPresentation);

}

if (compileFlag.equals("-p")) {

System.out.println("===================POLIZ===================");

for (int i = 0; i < internalProgramPresentation.getPolizTable().size(); i++) {

String lexemeName = "";

String tableName = "";

switch (internalProgramPresentation.getPolizTable().get(i).getTableId()){

case InternalProgramPresentation.serviceTableId:

lexemeName = internalProgramPresentation.getServiceTable().get(internalProgramPresentation.getPolizTable().get(i).getLexId()).getLexeme();

tableName = "serv";

break;

case InternalProgramPresentation.delimiterTableId:

lexemeName = internalProgramPresentation.getDelimiterTable().get(internalProgramPresentation.getPolizTable().get(i).getLexId()).getLexeme();

tableName = "delim";

break;

case InternalProgramPresentation.identifierTableId:

lexemeName = internalProgramPresentation.getIdentifierTable().get(internalProgramPresentation.getPolizTable().get(i).getLexId()).getLexeme();

tableName = "ident";

break;

case InternalProgramPresentation.numberTableId:

lexemeName = internalProgramPresentation.getNumberTable().get(internalProgramPresentation.getPolizTable().get(i).getLexId()).getLexeme();

tableName = "num ";

break;

case InternalProgramPresentation.polizPointerTableId:

lexemeName = String.valueOf(internalProgramPresentation.getPolizPointerTable().get(internalProgramPresentation.getPolizTable().get(i).getLexId()).getPolizIndex());

tableName = "point";

break;

}

System.out.println(String.format("%s\t%s\t\t%s", i, tableName, lexemeName));

}

}

if (compileFlag.equals("-i")) {

System.out.println("INTERPRETING PROGRAM...");

interpreter.interpret(internalProgramPresentation);

System.out.println("INTERPRETATION COMPLETE SUCCESSFUL");

}

System.out.println("====================END=====================");

} catch (CompilerExternalException e){

System.out.println("COMPILATION FAILED:\n" + e.getMessage());

System.out.println("====================END=====================");

} catch (CompilerInternalException e){

System.out.println("COMPILER INTERNAL ERROR:\n" + e.getMessage());

System.out.println("====================END=====================");

} catch (Exception e){

System.out.println("COMPILER FATAL ERROR:\n" + e.getMessage());

System.out.println("====================END=====================");

e.printStackTrace();

}

}

}package com.ilyakrn.entities.items;

public class MemoryItem {

private final Type type;

String value;

public MemoryItem(Type type, String value) {

this.type = type;

this.value = value;

}

public Type getType() {

return type;

}

public String getValue() {

return value;

}

public void setValue(String value) {

this.value = value;

}

}

package com.ilyakrn.entities.items;

public class NumberItem {

private final String lexeme;

private Type type;

private final int dimensionCount;

public NumberItem(String lexeme, int dimensionCount, Type type) {

this.lexeme = lexeme;

this.dimensionCount = dimensionCount;

this.type = type;

}

public void setType(Type type) {

this.type = type;

}

public String getLexeme() {

return lexeme;

}

public Type getType() {

return type;

}

public int getDimensionCount() {

return dimensionCount;

}

}

package com.ilyakrn.parser;

import com.ilyakrn.entities.items.\*;

import com.ilyakrn.entities.InternalProgramPresentation;

import com.ilyakrn.exceptions.external.SemanticException;

import com.ilyakrn.exceptions.external.SyntaxException;

import com.ilyakrn.exceptions.internal.InternalParserException;

import java.util.\*;

public class Parser {

private ArrayList<ServiceItem> serviceTable;

private ArrayList<DelimiterItem> delimiterTable;

private ArrayList<IdentifierItem> identifierTable;

private ArrayList<NumberItem> numberTable;

private ArrayList<PolizItem> polizTable;

private ArrayList<PolizPointerItem> polizPointerItems;

private ArrayList<BinOperationItem> binOperationTable;

private Stack<String> exprStack;

private String message;

private int errorCol;

private int errorRow;

private HashMap<String, Integer> operatorsIndexes;

private int oneIndex;

public InternalProgramPresentation analyze(InternalProgramPresentation internalProgramPresentation) throws InternalParserException, SyntaxException, SemanticException {

serviceTable = internalProgramPresentation.getServiceTable();

delimiterTable = internalProgramPresentation.getDelimiterTable();

identifierTable = internalProgramPresentation.getIdentifierTable();

numberTable = internalProgramPresentation.getNumberTable();

binOperationTable = new ArrayList<>();

polizTable = new ArrayList<>();

polizPointerItems = new ArrayList<>();

exprStack = new Stack<>();

operatorsIndexes = new HashMap<>();

message = "";

delimiterTable.add(new DelimiterItem("!"));

delimiterTable.add(new DelimiterItem("!F"));

delimiterTable.add(new DelimiterItem("R"));

delimiterTable.add(new DelimiterItem("W"));

binOperationTable.add(new BinOperationItem(">", Type.INT, Type.INT, Type.BOOL));

binOperationTable.add(new BinOperationItem(">", Type.INT, Type.FLOAT, Type.BOOL));

binOperationTable.add(new BinOperationItem(">", Type.FLOAT, Type.INT, Type.BOOL));

binOperationTable.add(new BinOperationItem(">", Type.FLOAT, Type.FLOAT, Type.BOOL));

binOperationTable.add(new BinOperationItem("<", Type.INT, Type.INT, Type.BOOL));

binOperationTable.add(new BinOperationItem("<", Type.INT, Type.FLOAT, Type.BOOL));

binOperationTable.add(new BinOperationItem("<", Type.FLOAT, Type.INT, Type.BOOL));

binOperationTable.add(new BinOperationItem("<", Type.FLOAT, Type.FLOAT, Type.BOOL));

binOperationTable.add(new BinOperationItem("<>", Type.INT, Type.INT, Type.BOOL));

binOperationTable.add(new BinOperationItem("<>", Type.INT, Type.FLOAT, Type.BOOL));

binOperationTable.add(new BinOperationItem("<>", Type.FLOAT, Type.INT, Type.BOOL));

binOperationTable.add(new BinOperationItem("<>", Type.FLOAT, Type.FLOAT, Type.BOOL));

binOperationTable.add(new BinOperationItem("<>", Type.BOOL, Type.BOOL, Type.BOOL));

binOperationTable.add(new BinOperationItem(">=", Type.INT, Type.INT, Type.BOOL));

binOperationTable.add(new BinOperationItem(">=", Type.INT, Type.FLOAT, Type.BOOL));

binOperationTable.add(new BinOperationItem(">=", Type.FLOAT, Type.INT, Type.BOOL));

binOperationTable.add(new BinOperationItem(">=", Type.FLOAT, Type.FLOAT, Type.BOOL));

binOperationTable.add(new BinOperationItem("<=", Type.INT, Type.INT, Type.BOOL));

binOperationTable.add(new BinOperationItem("<=", Type.INT, Type.FLOAT, Type.BOOL));

binOperationTable.add(new BinOperationItem("<=", Type.FLOAT, Type.INT, Type.BOOL));

binOperationTable.add(new BinOperationItem("<=", Type.FLOAT, Type.FLOAT, Type.BOOL));

binOperationTable.add(new BinOperationItem("=", Type.INT, Type.INT, Type.BOOL));

binOperationTable.add(new BinOperationItem("=", Type.INT, Type.FLOAT, Type.BOOL));

binOperationTable.add(new BinOperationItem("=", Type.FLOAT, Type.INT, Type.BOOL));

binOperationTable.add(new BinOperationItem("=", Type.FLOAT, Type.FLOAT, Type.BOOL));

binOperationTable.add(new BinOperationItem("=", Type.BOOL, Type.BOOL, Type.BOOL));

binOperationTable.add(new BinOperationItem("+", Type.INT, Type.INT, Type.INT));

binOperationTable.add(new BinOperationItem("+", Type.INT, Type.FLOAT, Type.FLOAT));

binOperationTable.add(new BinOperationItem("+", Type.FLOAT, Type.INT, Type.FLOAT));

binOperationTable.add(new BinOperationItem("+", Type.FLOAT, Type.FLOAT, Type.FLOAT));

binOperationTable.add(new BinOperationItem("-", Type.INT, Type.INT, Type.INT));

binOperationTable.add(new BinOperationItem("-", Type.INT, Type.FLOAT, Type.FLOAT));

binOperationTable.add(new BinOperationItem("-", Type.FLOAT, Type.INT, Type.FLOAT));

binOperationTable.add(new BinOperationItem("-", Type.FLOAT, Type.FLOAT, Type.FLOAT));

binOperationTable.add(new BinOperationItem("\*", Type.INT, Type.INT, Type.INT));

binOperationTable.add(new BinOperationItem("\*", Type.INT, Type.FLOAT, Type.FLOAT));

binOperationTable.add(new BinOperationItem("\*", Type.FLOAT, Type.INT, Type.FLOAT));

binOperationTable.add(new BinOperationItem("\*", Type.FLOAT, Type.FLOAT, Type.FLOAT));

binOperationTable.add(new BinOperationItem("/", Type.INT, Type.INT, Type.FLOAT));

binOperationTable.add(new BinOperationItem("/", Type.INT, Type.FLOAT, Type.FLOAT));

binOperationTable.add(new BinOperationItem("/", Type.FLOAT, Type.INT, Type.FLOAT));

binOperationTable.add(new BinOperationItem("/", Type.FLOAT, Type.FLOAT, Type.FLOAT));

binOperationTable.add(new BinOperationItem("or", Type.BOOL, Type.BOOL, Type.BOOL));

binOperationTable.add(new BinOperationItem("and", Type.BOOL, Type.BOOL, Type.BOOL));

for (int i = 0; i < delimiterTable.size(); i++) {

operatorsIndexes.put(delimiterTable.get(i).getLexeme(), i);

}

oneIndex = -1;

for (int i = 0; i < numberTable.size(); i++) {

if (numberTable.get(i).getLexeme().equals("1")) {

oneIndex = i;

break;

}

}

if (oneIndex == -1) {

oneIndex = numberTable.size();

numberTable.add(new NumberItem("1", 10, Type.INT));

}

Queue<ParserQueueItem> input = new LinkedList<>();

for (int i = 0; i < internalProgramPresentation.getLexemesSeqTable().size(); i++) {

String lexemeText = "";

switch (internalProgramPresentation.getLexemesSeqTable().get(i).getTableId()){

case InternalProgramPresentation.serviceTableId:

lexemeText = serviceTable.get(internalProgramPresentation.getLexemesSeqTable().get(i).getLexId()).getLexeme();

break;

case InternalProgramPresentation.delimiterTableId:

lexemeText = delimiterTable.get(internalProgramPresentation.getLexemesSeqTable().get(i).getLexId()).getLexeme();

break;

case InternalProgramPresentation.identifierTableId:

lexemeText = identifierTable.get(internalProgramPresentation.getLexemesSeqTable().get(i).getLexId()).getLexeme();

break;

case InternalProgramPresentation.numberTableId:

lexemeText = numberTable.get(internalProgramPresentation.getLexemesSeqTable().get(i).getLexId()).getLexeme();

break;

default:

throw new InternalParserException("Invalid table id: '" + internalProgramPresentation.getLexemesSeqTable().get(i).getTableId() + "'");

}

input.add(new ParserQueueItem(internalProgramPresentation.getLexemesSeqTable().get(i).getLexId(), internalProgramPresentation.getLexemesSeqTable().get(i).getTableId(), lexemeText, internalProgramPresentation.getLexemesSeqTable().get(i).getRow(), internalProgramPresentation.getLexemesSeqTable().get(i).getCol()));

}

int result = PROG(input);

if(result != input.size())

throw new SyntaxException(errorRow + ":" + errorCol + " " + message);

return new InternalProgramPresentation(serviceTable, delimiterTable, identifierTable, numberTable, internalProgramPresentation.getLexemesSeqTable(), binOperationTable, polizTable, polizPointerItems);

}

private int nextLexemeIs(Queue<ParserQueueItem> input, String lexeme){

Queue<ParserQueueItem> tempInput = new LinkedList<>(input);

if (tempInput.isEmpty()) {

errorCol = -1;

errorRow = -1;

message = "unexpected end of program";

return -1;

}

else{

ParserQueueItem next = tempInput.poll();

int skipped = 0;

while (next.getLexeme().equals("\n")){

next = tempInput.poll();

skipped++;

}

if (skipped > 0 && lexeme.equals("\n"))

return skipped;

if (next.getLexeme().equals(lexeme)){

return 1 + skipped;

}

else {

errorCol = next.getCol() - next.getLexeme().length() - 1;

errorRow = next.getRow();

message = "error at or near lexeme '" + next.getLexeme() + "'";

return -1;

}

}

}

private int isIdentifier(Queue<ParserQueueItem> input){

Queue<ParserQueueItem> tempInput = new LinkedList<>(input);

if (tempInput.isEmpty()) {

errorCol = -1;

errorRow = -1;

message = "unexpected end of program";

return -1;

}

else{

ParserQueueItem next = tempInput.poll();

int skipped = 0;

while (next.getLexeme().equals("\n")){

next = tempInput.poll();

skipped++;

}

if (next.getTableId() == InternalProgramPresentation.identifierTableId){

return 1 + skipped;

}

else {

errorCol = next.getCol() - next.getLexeme().length() - 1;

errorRow = next.getRow();

message = "error at or near lexeme '" + next.getLexeme() + "', identifier expected";

return -1;

}

}

}

private int isNumber(Queue<ParserQueueItem> input) {

Queue<ParserQueueItem> tempInput = new LinkedList<>(input);

if (tempInput.isEmpty()) {

errorCol = -1;

errorRow = -1;

message = "unexpected end of program";

return -1;

}

else{

ParserQueueItem next = tempInput.poll();

int skipped = 0;

while (next.getLexeme().equals("\n")){

next = tempInput.poll();

skipped++;

}

if (next.getTableId() == InternalProgramPresentation.numberTableId){

return 1 + skipped;

}

else {

errorCol = next.getCol() - next.getLexeme().length() - 1;

errorRow = next.getRow();

message = "error at or near lexeme '" + next.getLexeme() + "', number expected";

return -1;

}

}

}

private int OGO(Queue<ParserQueueItem> input){

Queue<ParserQueueItem> tempInput = new LinkedList<>(input);

if (tempInput.isEmpty()) {

errorCol = -1;

errorRow = -1;

message = "unexpected end of program";

return -1;

}

else{

ParserQueueItem next = tempInput.poll();

int skipped = 0;

while (next.getLexeme().equals("\n")){

next = tempInput.poll();

skipped++;

}

if (next.getLexeme().equals("<>") ||

next.getLexeme().equals("=") ||

next.getLexeme().equals("<") ||

next.getLexeme().equals("<=") ||

next.getLexeme().equals(">") ||

next.getLexeme().equals(">=")

){

return 1 + skipped;

}

else {

errorCol = next.getCol() - next.getLexeme().length() - 1;

errorRow = next.getRow();

message = "error at or near lexeme '" + next.getLexeme() + "', relation operation expected";

return -1;

}

}

}

private int OGS(Queue<ParserQueueItem> input){

Queue<ParserQueueItem> tempInput = new LinkedList<>(input);

if (tempInput.isEmpty()) {

errorCol = -1;

errorRow = -1;

message = "unexpected end of program";

return -1;

}

else{

ParserQueueItem next = tempInput.poll();

int skipped = 0;

while (next.getLexeme().equals("\n")){

next = tempInput.poll();

skipped++;

}

if (next.getLexeme().equals("+") ||

next.getLexeme().equals("-") ||

next.getLexeme().equals("or")

){

return 1 + skipped;

}

else {

errorCol = next.getCol() - next.getLexeme().length() - 1;

errorRow = next.getRow();

message = "error at or near lexeme '" + next.getLexeme() + "', addition operation expected";

return -1;

}

}

}

private int OGU(Queue<ParserQueueItem> input){

Queue<ParserQueueItem> tempInput = new LinkedList<>(input);

if (tempInput.isEmpty()) {

errorCol = -1;

errorRow = -1;

message = "unexpected end of program";

return -1;

}

else{

ParserQueueItem next = tempInput.poll();

int skipped = 0;

while (next.getLexeme().equals("\n")){

next = tempInput.poll();

skipped++;

}

if (next.getLexeme().equals("\*") ||

next.getLexeme().equals("/") ||

next.getLexeme().equals("and")

){

return 1 + skipped;

}

else {

errorCol = next.getCol() - next.getLexeme().length() - 1;

errorRow = next.getRow();

message = "error at or near lexeme '" + next.getLexeme() + "', multiplication operation expected";

return -1;

}

}

}

private int LC(Queue<ParserQueueItem> input){

Queue<ParserQueueItem> tempInput = new LinkedList<>(input);

if (tempInput.isEmpty()) {

errorCol = -1;

errorRow = -1;

message = "unexpected end of program";

return -1;

}

else{

ParserQueueItem next = tempInput.poll();

int skipped = 0;

while (next.getLexeme().equals("\n")){

next = tempInput.poll();

skipped++;

}

if (next.getLexeme().equals("true") ||

next.getLexeme().equals("false")

){

return 1 + skipped;

}

else {

errorCol = next.getCol() - next.getLexeme().length() - 1;

errorRow = next.getRow();

message = "error at or near lexeme '" + next.getLexeme() + "', boolean const expected";

return -1;

}

}

}

private int UO(Queue<ParserQueueItem> input){

Queue<ParserQueueItem> tempInput = new LinkedList<>(input);

if (tempInput.isEmpty()) {

errorCol = -1;

errorRow = -1;

message = "unexpected end of program";

return -1;

}

else{

ParserQueueItem next = tempInput.poll();

int skipped = 0;

while (next.getLexeme().equals("\n")){

next = tempInput.poll();

skipped++;

}

if (next.getLexeme().equals("not")){

return 1 + skipped;

}

else {

errorCol = next.getCol() - next.getLexeme().length() - 1;

errorRow = next.getRow();

message = "unexpected lexeme '" + next.getLexeme() + "', 'not' expected";

return -1;

}

}

}

private int T(Queue<ParserQueueItem> input){

Queue<ParserQueueItem> tempInput = new LinkedList<>(input);

if (tempInput.isEmpty()) {

errorCol = -1;

errorRow = -1;

message = "unexpected end of program";

return -1;

}

else{

ParserQueueItem next = tempInput.poll();

int skipped = 0;

while (next.getLexeme().equals("\n")){

next = tempInput.poll();

skipped++;

}

if (next.getLexeme().equals("int") ||

next.getLexeme().equals("float") ||

next.getLexeme().equals("bool")

){

return 1 + skipped;

}

else {

errorCol = next.getCol() - next.getLexeme().length() - 1;

errorRow = next.getRow();

message = "unexpected lexeme '" + next.getLexeme() + "', type expected";

return -1;

}

}

}

private int EXPR(Queue<ParserQueueItem> input){

Queue<ParserQueueItem> tempInput = new LinkedList<>(input);

int currentPolizPos = polizTable.size();

exprStack.clear();

int result = OPRND(tempInput);

if (result != -1){

for (int i = 0; i < result; i++) {

tempInput.poll();

}

int operIndex = polizTable.size();

int result1 = OGO(tempInput);

while (result1 != -1) {

for (int i = 0; i < result1 - 1; i++) {

tempInput.poll();

}

exprStack.add(tempInput.peek().getLexeme());

polizTable.add(new PolizItem(tempInput.peek().getLexId(),tempInput.poll().getTableId()));

result += result1;

int result2 = OPRND(tempInput);

if (result2 != -1) {

for (int i = 0; i < result2; i++) {

tempInput.poll();

}

polizTable.add(polizTable.get(operIndex));

polizTable.remove(operIndex);

operIndex = polizTable.size();

result1 = OGO(tempInput);

result += result2;

}

else {

for (int i = 0; i < polizTable.size() - currentPolizPos; i++) {

polizTable.remove(currentPolizPos);

}

result = -1;

break;

}

}

}

Type t = getExprType(exprStack);

return result;

}

private int SLAG(Queue<ParserQueueItem> input){

Queue<ParserQueueItem> tempInput = new LinkedList<>(input);

int currentPolizPos = polizTable.size();

int result = MNOZH(tempInput);

if (result != -1){

for (int i = 0; i < result; i++) {

tempInput.poll();

}

int operIndex = polizTable.size();

int result1 = OGU(tempInput);

while (result1 != -1) {

for (int i = 0; i < result1 - 1; i++) {

tempInput.poll();

}

exprStack.add(tempInput.peek().getLexeme());

polizTable.add(new PolizItem(tempInput.peek().getLexId(),tempInput.poll().getTableId()));

result += result1;

int result2 = MNOZH(tempInput);

if (result2 != -1) {

for (int i = 0; i < result2; i++) {

tempInput.poll();

}

polizTable.add(polizTable.get(operIndex));

polizTable.remove(operIndex);

operIndex = polizTable.size();

result1 = OGU(tempInput);

result += result2;

}

else {

for (int i = 0; i < polizTable.size() - currentPolizPos; i++) {

polizTable.remove(currentPolizPos);

}

result = -1;

break;

}

}

}

return result;

}

private int OPRND(Queue<ParserQueueItem> input){

Queue<ParserQueueItem> tempInput = new LinkedList<>(input);

int currentPolizPos = polizTable.size();

int result = SLAG(tempInput);

if (result != -1){

for (int i = 0; i < result; i++) {

tempInput.poll();

}

int operIndex = polizTable.size();

int result1 = OGS(tempInput);

while (result1 != -1) {

for (int i = 0; i < result1 - 1; i++) {

tempInput.poll();

}

exprStack.add(tempInput.peek().getLexeme());

polizTable.add(new PolizItem(tempInput.peek().getLexId(),tempInput.poll().getTableId()));

result += result1;

int result2 = SLAG(tempInput);

if (result2 != -1) {

for (int i = 0; i < result2; i++) {

tempInput.poll();

}

polizTable.add(polizTable.get(operIndex));

polizTable.remove(operIndex);

operIndex = polizTable.size();

result1 = OGS(tempInput);

result += result2;

}

else {

for (int i = 0; i < polizTable.size() - currentPolizPos; i++) {

polizTable.remove(currentPolizPos);

}

result = -1;

break;

}

}

}

return result;

}

private int MNOZH(Queue<ParserQueueItem> input){

Queue<ParserQueueItem> tempInput = new LinkedList<>(input);

int currentPolizPos = polizTable.size();

int result = -1;

if (result == -1) {

result = UO(tempInput);

if (result != -1) {

for (int i = 0; i < result - 1; i++) {

tempInput.poll();

}

polizTable.add(new PolizItem(tempInput.peek().getLexId(),tempInput.poll().getTableId()));

int result1 = MNOZH(tempInput);

if(result1 != -1){

for (int i = 0; i < result1 - 1; i++) {

tempInput.poll();

}

Type type = getType(tempInput.peek());

if(type != Type.BOOL)

throw new SemanticException("'not' can apply to bool only");

if (exprStack.isEmpty() || (!exprStack.isEmpty() && !exprStack.peek().equals(Type.BOOL.name())))

exprStack.add(type.name());

polizTable.add(polizTable.get(currentPolizPos));

polizTable.remove(currentPolizPos);

result += result1;

return result;

}

else {

throw new SemanticException("'not' statement not completed");

}

}

}

if (result == -1)

result = isIdentifier(tempInput);

if (result == -1)

result = isNumber(tempInput);

if (result == -1)

result = LC(tempInput);

// if (result == -1)

// result = MNOZH(tempInput);

if(result != -1){

for (int i = 0; i < result - 1; i++) {

tempInput.poll();

}

Type type = getType(tempInput.peek());

exprStack.add(type.name());

polizTable.add(new PolizItem(tempInput.peek().getLexId(),tempInput.poll().getTableId()));

}

return result;

}

private int ENTER(Queue<ParserQueueItem> input){

Queue<ParserQueueItem> tempInput = new LinkedList<>(input);

int currentPolizPos = polizTable.size();

int result = nextLexemeIs(tempInput,"read");

if (result == -1)

return -1;

for (int i = 0; i < result; i++) {

tempInput.poll();

}

int result1 = nextLexemeIs(tempInput,"(");

if (result1 == -1)

return -1;

for (int i = 0; i < result1; i++) {

tempInput.poll();

}

result += result1;

int result2 = isIdentifier(tempInput);

if (result2 == -1)

return -1;

for (int i = 0; i < result2 - 1; i++) {

tempInput.poll();

}

getType(tempInput.peek());

result += result2;

polizTable.add(new PolizItem(tempInput.peek().getLexId(),tempInput.poll().getTableId()));

polizTable.add(new PolizItem(operatorsIndexes.get("R"), InternalProgramPresentation.delimiterTableId));

int result3 = nextLexemeIs(tempInput,",");

while (result3 != -1) {

for (int i = 0; i < result3; i++) {

tempInput.poll();

}

result += result3;

int result4 = isIdentifier(tempInput);

if (result4 == -1) {

for (int i = 0; i < polizTable.size() - currentPolizPos; i++) {

polizTable.remove(currentPolizPos);

}

return -1;

}

for (int i = 0; i < result4 - 1; i++) {

tempInput.poll();

}

getType(tempInput.peek());

result += result4;

polizTable.add(new PolizItem(tempInput.peek().getLexId(),tempInput.poll().getTableId()));

polizTable.add(new PolizItem(operatorsIndexes.get("R"), InternalProgramPresentation.delimiterTableId));

result3 = nextLexemeIs(tempInput,",");

}

int result5 = nextLexemeIs(tempInput,")");

if (result5 == -1) {

for (int i = 0; i < polizTable.size() - currentPolizPos; i++) {

polizTable.remove(currentPolizPos);

}

return -1;

}

for (int i = 0; i < result5; i++) {

tempInput.poll();

}

result += result5;

return result;

}

private int OUT(Queue<ParserQueueItem> input){

Queue<ParserQueueItem> tempInput = new LinkedList<>(input);

int currentPolizPos = polizTable.size();

int result = nextLexemeIs(tempInput,"write");

if (result == -1)

return -1;

for (int i = 0; i < result; i++) {

tempInput.poll();

}

int result1 = nextLexemeIs(tempInput,"(");

if (result1 == -1)

return -1;

for (int i = 0; i < result1; i++) {

tempInput.poll();

}

result += result1;

int result2 = EXPR(tempInput);

if (result2 == -1)

return -1;

for (int i = 0; i < result2; i++) {

tempInput.poll();

}

result += result2;

polizTable.add(new PolizItem(operatorsIndexes.get("W"), InternalProgramPresentation.delimiterTableId));

int result3 = nextLexemeIs(tempInput,",");

while (result3 != -1) {

for (int i = 0; i < result3; i++) {

tempInput.poll();

}

result += result3;

int result4 = EXPR(tempInput);

if (result4 == -1) {

for (int i = 0; i < polizTable.size() - currentPolizPos; i++) {

polizTable.remove(currentPolizPos);

}

return -1;

}

for (int i = 0; i < result4; i++) {

tempInput.poll();

}

result += result4;

polizTable.add(new PolizItem(operatorsIndexes.get("W"), InternalProgramPresentation.delimiterTableId));

result3 = nextLexemeIs(tempInput,",");

}

int result5 = nextLexemeIs(tempInput,")");

if (result5 == -1) {

for (int i = 0; i < polizTable.size() - currentPolizPos; i++) {

polizTable.remove(currentPolizPos);

}

return -1;

}

for (int i = 0; i < result5; i++) {

tempInput.poll();

}

result += result5;

return result;

}

private int PRISV(Queue<ParserQueueItem> input){

Queue<ParserQueueItem> tempInput = new LinkedList<>(input);

int currentPolizPos = polizTable.size();

int result = isIdentifier(tempInput);

if (result == -1)

return -1;

for (int i = 0; i < result - 1; i++) {

tempInput.poll();

}

Type typeVar = getType(tempInput.peek());

polizTable.add(new PolizItem(tempInput.peek().getLexId(),tempInput.poll().getTableId()));

int result1 = nextLexemeIs(tempInput, "ass");

if (result1 == -1) {

polizTable.remove(currentPolizPos);

return -1;

}

for (int i = 0; i < result1 - 1; i++) {

tempInput.poll();

}

int operIndex = polizTable.size();

polizTable.add(new PolizItem(tempInput.peek().getLexId(),tempInput.poll().getTableId()));

result += result1;

int result2 = EXPR(tempInput);

if (result2 == -1) {

for (int i = 0; i < polizTable.size() - currentPolizPos; i++) {

polizTable.remove(currentPolizPos);

}

return -1;

}

for (int i = 0; i < result2; i++) {

tempInput.poll();

}

polizTable.add(polizTable.get(operIndex));

polizTable.remove(operIndex);

if (!exprStack.peek().equals(typeVar.name()))

throw new SemanticException(typeVar.name() + " can not be assigned as " + exprStack.peek());

result += result2;

return result;

}

private int USLOV(Queue<ParserQueueItem> input){

Queue<ParserQueueItem> tempInput = new LinkedList<>(input);

int currentPolizPos = polizTable.size();

int result = nextLexemeIs(tempInput, "if");

if (result == -1)

return -1;

for (int i = 0; i < result; i++) {

tempInput.poll();

}

int result1 = EXPR(tempInput);

if (result1 == -1)

return -1;

for (int i = 0; i < result1; i++) {

tempInput.poll();

}

if (!exprStack.peek().equals(Type.BOOL.name())) {

for (int i = 0; i < polizTable.size() - currentPolizPos; i++) {

polizTable.remove(currentPolizPos);

}

throw new SemanticException("can not use " + exprStack.peek() + " as " + Type.BOOL.name());

}

result += result1;

int result2 = nextLexemeIs(tempInput, "then");

if (result2 == -1) {

for (int i = 0; i < polizTable.size() - currentPolizPos; i++) {

polizTable.remove(currentPolizPos);

}

return -1;

}

for (int i = 0; i < result2; i++) {

tempInput.poll();

}

result += result2;

int gotoIfFalseIndex = polizTable.size();

polizTable.add(null);

polizTable.add(new PolizItem(operatorsIndexes.get("!F"), InternalProgramPresentation.delimiterTableId));

int result3 = OPERATOR(tempInput);

if (result3 == -1) {

for (int i = 0; i < polizTable.size() - currentPolizPos; i++) {

polizTable.remove(currentPolizPos);

}

return -1;

}

for (int i = 0; i < result3; i++) {

tempInput.poll();

}

result += result3;

int gotoIfFalse = polizTable.size();

int result4 = nextLexemeIs(tempInput, "else");

if (result4 != -1) {

for (int i = 0; i < result4; i++) {

tempInput.poll();

}

result += result4;

int gotoNextIndex = polizTable.size();

polizTable.add(null);

polizTable.add(new PolizItem(operatorsIndexes.get("!"), InternalProgramPresentation.delimiterTableId));

gotoIfFalse = polizTable.size();

int result5 = OPERATOR(tempInput);

if (result5 == -1) {

for (int i = 0; i < polizTable.size() - currentPolizPos; i++) {

polizTable.remove(currentPolizPos);

}

return -1;

}

for (int i = 0; i < result5; i++) {

tempInput.poll();

}

polizTable.set(gotoNextIndex, new PolizItem(polizPointerItems.size(), InternalProgramPresentation.polizPointerTableId));

polizPointerItems.add(new PolizPointerItem(polizTable.size()));

result += result5;

}

polizTable.set(gotoIfFalseIndex, new PolizItem(polizPointerItems.size(), InternalProgramPresentation.polizPointerTableId));

polizPointerItems.add(new PolizPointerItem(gotoIfFalse));

return result;

}

private int FIXLOOP(Queue<ParserQueueItem> input){

Queue<ParserQueueItem> tempInput = new LinkedList<>(input);

int currentPolizPos = polizTable.size();

int result = nextLexemeIs(tempInput, "for");

if (result == -1)

return -1;

for (int i = 0; i < result; i++) {

tempInput.poll();

}

int result1 = PRISV(tempInput);

if (result1 == -1) {

for (int i = 0; i < polizTable.size() - currentPolizPos; i++) {

polizTable.remove(currentPolizPos);

}

return -1;

}

for (int i = 0; i < result1; i++) {

tempInput.poll();

}

result += result1;

int gotoExpr = polizTable.size();

polizTable.add(polizTable.get(currentPolizPos));

int result2 = nextLexemeIs(tempInput, "to");

if (result2 == -1) {

for (int i = 0; i < polizTable.size() - currentPolizPos; i++) {

polizTable.remove(currentPolizPos);

}

return -1;

}

for (int i = 0; i < result2; i++) {

tempInput.poll();

}

result += result2;

int result3 = EXPR(tempInput);

if (result3 == -1){

for (int i = 0; i < polizTable.size() - currentPolizPos; i++) {

polizTable.remove(currentPolizPos);

}

return -1;

}

for (int i = 0; i < result3; i++) {

tempInput.poll();

}

result += result3;

polizTable.add(new PolizItem(operatorsIndexes.get("<"), InternalProgramPresentation.delimiterTableId));

int gotoNextIndex = polizTable.size();

polizTable.add(null);

polizTable.add(new PolizItem(operatorsIndexes.get("!F"), InternalProgramPresentation.delimiterTableId));

int result4 = nextLexemeIs(tempInput, "do");

if (result4 == -1) {

for (int i = 0; i < polizTable.size() - currentPolizPos; i++) {

polizTable.remove(currentPolizPos);

}

return -1;

}

for (int i = 0; i < result4; i++) {

tempInput.poll();

}

result += result4;

int result5 = OPERATOR(tempInput);

if (result5 == -1) {

for (int i = 0; i < polizTable.size() - currentPolizPos; i++) {

polizTable.remove(currentPolizPos);

}

return -1;

}

for (int i = 0; i < result5; i++) {

tempInput.poll();

}

result += result5;

polizTable.add(polizTable.get(currentPolizPos));

polizTable.add(polizTable.get(currentPolizPos));

polizTable.add(new PolizItem(oneIndex, InternalProgramPresentation.numberTableId));

polizTable.add(new PolizItem(operatorsIndexes.get("+"), InternalProgramPresentation.delimiterTableId));

polizTable.add(new PolizItem(operatorsIndexes.get("ass"), InternalProgramPresentation.delimiterTableId));

polizTable.add(new PolizItem(polizPointerItems.size(), InternalProgramPresentation.polizPointerTableId));

polizPointerItems.add(new PolizPointerItem(gotoExpr));

polizTable.add(new PolizItem(operatorsIndexes.get("!"), InternalProgramPresentation.delimiterTableId));

polizTable.set(gotoNextIndex, new PolizItem(polizPointerItems.size(), InternalProgramPresentation.polizPointerTableId));

polizPointerItems.add(new PolizPointerItem(polizTable.size()));

return result;

}

private int USLLOOP(Queue<ParserQueueItem> input){

Queue<ParserQueueItem> tempInput = new LinkedList<>(input);

int currentPolizPos = polizTable.size();

int result = nextLexemeIs(tempInput, "while");

if (result == -1)

return -1;

for (int i = 0; i < result; i++) {

tempInput.poll();

}

int gotoExpr = polizTable.size();

int result1 = EXPR(tempInput);

if (result1 == -1)

return -1;

for (int i = 0; i < result1; i++) {

tempInput.poll();

}

if (!exprStack.peek().equals(Type.BOOL.name())) {

for (int i = 0; i < polizTable.size() - currentPolizPos; i++) {

polizTable.remove(currentPolizPos);

}

throw new SemanticException("can not use " + exprStack.peek() + " as " + Type.BOOL.name());

}

result += result1;

int gotoNextIndex = polizTable.size();

polizTable.add(null);

polizTable.add(new PolizItem(operatorsIndexes.get("!F"), InternalProgramPresentation.delimiterTableId));

int result2 = nextLexemeIs(tempInput, "do");

if (result2 == -1) {

for (int i = 0; i < polizTable.size() - currentPolizPos; i++) {

polizTable.remove(currentPolizPos);

}

return -1;

}

for (int i = 0; i < result2; i++) {

tempInput.poll();

}

result += result2;

int result3 = OPERATOR(tempInput);

if (result3 == -1) {

for (int i = 0; i < polizTable.size() - currentPolizPos; i++) {

polizTable.remove(currentPolizPos);

}

return -1;

}

for (int i = 0; i < result3; i++) {

tempInput.poll();

}

result += result3;

polizTable.add(new PolizItem(polizPointerItems.size(), InternalProgramPresentation.polizPointerTableId));

polizPointerItems.add(new PolizPointerItem(gotoExpr));

polizTable.add(new PolizItem(operatorsIndexes.get("!"), InternalProgramPresentation.delimiterTableId));

polizTable.set(gotoNextIndex, new PolizItem(polizPointerItems.size(), InternalProgramPresentation.polizPointerTableId));

polizPointerItems.add(new PolizPointerItem(polizTable.size()));

return result;

}

private int OPERATOR(Queue<ParserQueueItem> input){

Queue<ParserQueueItem> tempInput = new LinkedList<>(input);

int result = -1;

if (result == -1)

result = PRISV(tempInput);

if (result == -1)

result = USLOV(tempInput);

if (result == -1)

result = FIXLOOP(tempInput);

if (result == -1)

result = USLLOOP(tempInput);

if (result == -1)

result = ENTER(tempInput);

if (result == -1)

result = OUT(tempInput);

if (result != -1){

for (int i = 0; i < result; i++) {

tempInput.poll();

}

boolean isLn = false;

int result1 = nextLexemeIs(tempInput, ":");

if (result1 == -1) {

result1 = nextLexemeIs(tempInput, "\n");

if (result1 != -1)

isLn = true;

}

while (result1 != -1) {

for (int i = 0; i < result1; i++) {

tempInput.poll();

}

result += result1;

int result2 = OPERATOR(tempInput);

if (result2 == -1 && !isLn)

return -1;

else if (result2 == -1 && isLn){

result -= result1;

break;

}

for (int i = 0; i < result2; i++) {

tempInput.poll();

}

result += result2;

isLn = false;

result1 = nextLexemeIs(tempInput, ":");

if (result1 == -1) {

result1 = nextLexemeIs(tempInput, "\n");

if (result1 != -1)

isLn = true;

}

}

}

return result;

}

private int DESC(Queue<ParserQueueItem> input) throws InternalParserException {

Queue<ParserQueueItem> tempInput = new LinkedList<>(input);

int result = T(tempInput);

if (result == -1)

return -1;

Type type = null;

for (int i = 0; i < result - 1; i++) {

tempInput.poll();

}

switch (tempInput.poll().getLexeme()){

case "int":

type = Type.INT;

break;

case "float":

type = Type.FLOAT;

break;

case "bool":

type = Type.BOOL;

break;

default:

throw new InternalParserException("unresolved type");

}

ArrayList<Integer> ids = new ArrayList<>();

int result1 = isIdentifier(tempInput);

if (result1 == -1)

return -1;

for (int i = 0; i < result1 - 1; i++) {

tempInput.poll();

}

ids.add(tempInput.poll().getLexId());

result += result1;

int result2 = nextLexemeIs(tempInput, ",");

while (result2 != -1) {

for (int i = 0; i < result2; i++) {

tempInput.poll();

}

result += result1;

int result3 = isIdentifier(tempInput);

if (result3 == -1)

return -1;

for (int i = 0; i < result3 - 1; i++) {

tempInput.poll();

}

ids.add(tempInput.poll().getLexId());

result += result3;

result2 = nextLexemeIs(tempInput, ",");

}

for (Integer id : ids) {

if(identifierTable.get(id).isInit())

throw new SemanticException(id + " has already been initialized");

identifierTable.get(id).setInit(true);

identifierTable.get(id).setType(type);

}

return result;

}

private int PROG(Queue<ParserQueueItem> input) throws InternalParserException, SemanticException {

Queue<ParserQueueItem> tempInput = new LinkedList<>(input);

int result = nextLexemeIs(tempInput, "{");

if (result == -1)

return -1;

for (int i = 0; i < result; i++) {

tempInput.poll();

}

int result1 = DESC(tempInput);

if (result1 == -1)

result1 = OPERATOR(tempInput);

if (result1 == -1)

return -1;

for (int i = 0; i < result1; i++) {

tempInput.poll();

}

result += result1;

int result2 = nextLexemeIs(tempInput, ";");

if (result2 == -1)

return -1;

for (int i = 0; i < result2; i++) {

tempInput.poll();

}

result += result2;

int result3 = DESC(tempInput);

if (result3 == -1)

result3 = OPERATOR(tempInput);

while (result3 != -1) {

for (int i = 0; i < result3; i++) {

tempInput.poll();

}

result += result3;

int result4 = nextLexemeIs(tempInput, ";");

if (result4 == -1)

return -1;

for (int i = 0; i < result4; i++) {

tempInput.poll();

}

result += result4;

result3 = DESC(tempInput);

if (result3 == -1)

result3 = OPERATOR(tempInput);

}

int result5 = nextLexemeIs(tempInput, "}");

if (result5 == -1)

return -1;

for (int i = 0; i < result5 - 1; i++) {

tempInput.poll();

}

polizTable.add(new PolizItem(tempInput.peek().getLexId(),tempInput.poll().getTableId()));

result += result5;

return result;

}

Type getType(ParserQueueItem parserQueueItem) {

Type result = null;

switch (parserQueueItem.getTableId()){

case InternalProgramPresentation.serviceTableId:

switch (parserQueueItem.getLexeme()){

case "true":

case "false":

result = Type.BOOL;

break;

default:

throw new InternalParserException("can not get type of service word or delimiter");

}

break;

case InternalProgramPresentation.delimiterTableId:

throw new InternalParserException("can not get type of service word or delimiter");

case InternalProgramPresentation.numberTableId:

if (parserQueueItem.getLexId() >= numberTable.size())

throw new InternalParserException("can not find number with id '" + parserQueueItem.getLexId() + "'");

result = numberTable.get(parserQueueItem.getLexId()).getType();

break;

case InternalProgramPresentation.identifierTableId:

if (parserQueueItem.getLexId() >= identifierTable.size())

throw new InternalParserException("can not find number with id '" + parserQueueItem.getLexId() + "'");

result = identifierTable.get(parserQueueItem.getLexId()).getType();

if (result == null)

throw new SemanticException("variable '" + parserQueueItem.getLexeme() + "' not defined");

break;

default:

throw new InternalParserException("can not find table with id '" + parserQueueItem.getTableId() + "'");

}

if (result == null)

throw new InternalParserException("can not get type of table id: '" + parserQueueItem.getTableId() + "', lexeme id: '" + parserQueueItem.getLexId() + "'");

return result;

}

Type getExprType(Stack<String> stack) {

while (stack.size() >= 3) {

String operand2 = stack.pop();

String operator = stack.pop();

String operand1 = stack.pop();

Type result = null;

for (int i = 0; i < binOperationTable.size(); i++) {

if (binOperationTable.get(i).getOperation().equals(operator)

&& binOperationTable.get(i).getOperand1().name().equals(operand1)

&& binOperationTable.get(i).getOperand2().name().equals(operand2)) {

result = binOperationTable.get(i).getResult();

break;

}

}

if (result == null)

throw new InternalParserException("can not get type of operand: '" + operand1 + "', operator: '" + operator + "', operand: '" + operand2 + "'");

stack.push(result.name());

}

if (stack.isEmpty())

throw new SemanticException("can not get type of expression. (possibly you try to use numbers < 0? use 0-<number>)");

return Type.valueOf(stack.peek());

}

/// ///////////////////////\*/

/\*\* OGO операции\_группы\_отношения \*\*/

/\*\* OGS операции\_группы\_сложения \*\*/

/\*\* OGU операции\_группы\_умножения \*\*/

/\*\* LC логическая\_константа \*\*/

/\*\* UO унарная\_операция \*\*/

/\*\* T тип \*\*/

/\*\* MNOZH множитель \*\*/

/\*\* EXPR выражение \*\*/

/\*\* SLAG слагаемое \*\*/

/\*\* OPRND операнд \*\*/

/\*\* ENTER ввода \*\*/

/\*\* OUT вывода \*\*/

/\*\* PRISV присваивания \*\*/

/\*\* USLOV условный \*\*/

/\*\* FIXLOOP фиксированного\_цикла \*\*/

/\*\* USLLOOP условного\_цикла \*\*/

/\*\* SOSTAV составной \*\*/

/\*\* OPERATOR оператор \*\*/

/\*\* DESC описание \*\*/

/\*\* PROG программа \*\*/

// <операнд>::= <слагаемое> {<операции\_группы\_сложения> <слагаемое>}

// <выражение>::= <операнд>{<операции\_группы\_отношения> <операнд>}

// <множитель>::= (<идентификатор> | <число> | <логическая\_константа> | <унарная\_операция> <множитель> | <выражение>)

// <слагаемое>::= <множитель> {<операции\_группы\_умножения> <множитель>}

// <ввода>::= read (<идентификатор> {, <идентификатор> })

// <вывода>::= write (<выражение> {, <выражение> })

// <присваивания>::= <идентификатор> ass <выражение>

// <условный>::= if <выражение> then <оператор> [ else <оператор>]

// <фиксированного\_цикла>::= for <присваивания> to <выражение> do <оператор>

// <условного\_цикла>::= while <выражение> do <оператор>

// ---------<составной>::= <оператор> { ( : | перевод строки) <оператор> }

// <оператор>::= (<присваивания> | <условный> | <фиксированного\_цикла> | <условного\_цикла> | <ввода> | <вывода>) { ( : | перевод строки) <оператор> }

// <описание>::= <тип> <идентификатор> { , <идентификатор> }

// <программа>::= { {/ (<описание> | <оператор>) ; /} }

}

package com.ilyakrn.entities.items;

public class ParserQueueItem {

private final int lexId;

private final int tableId;

private final String lexeme;

private final int row;

private final int col;

public ParserQueueItem(int lexId, int tableId, String lexeme, int row, int col) {

this.lexId = lexId;

this.tableId = tableId;

this.lexeme = lexeme;

this.row = row;

this.col = col;

}

public int getLexId() {

return lexId;

}

public int getTableId() {

return tableId;

}

public String getLexeme() {

return lexeme;

}

public int getRow() {

return row;

}

public int getCol() {

return col;

}

}

package com.ilyakrn.entities.items;

public class PolizItem {

private final int lexId;

private final int tableId;

public PolizItem(int lexId, int tableId) {

this.lexId = lexId;

this.tableId = tableId;

}

public int getLexId() {

return lexId;

}

public int getTableId() {

return tableId;

}

}

package com.ilyakrn.entities.items;

public class PolizPointerItem {

private final int polizIndex;

public int getPolizIndex() {

return polizIndex;

}

public PolizPointerItem(int polizIndex) {

this.polizIndex = polizIndex;

}

}

package com.ilyakrn.exceptions.external;

public class SemanticException extends CompilerExternalException {

public SemanticException(String message) {

super("Semantic error: " + message);

}

}

package com.ilyakrn.entities.items;

public class ServiceItem {

private String lexeme;

public String getLexeme() {

return lexeme;

}

public void setLexeme(String lexeme) {

this.lexeme = lexeme;

}

public ServiceItem(String lexeme) {

this.lexeme = lexeme;

}

}

package com.ilyakrn.exceptions.external;

public class SyntaxException extends CompilerExternalException {

public SyntaxException(String message) {

super("Syntax error: " + message);

}

}

package com.ilyakrn.entities.items;

public enum Type {

INT, FLOAT, BOOL

}

**Приложение Б – Контрольный пример**

{

/\* чтение чисел пока не прочитается отрицательное \*/

float num;

read(num);

while num >= 0 do

read(num);

}

{

/\* факториал числа \*/

int n, i, fact;

read(n);

fact ass 1;

for i ass 2 to n + 1 do

fact ass fact \* i;

write(fact);

}

{

/\* сортировка трех чисел по возрастанию \*/

int x, y, z, t;

read(x, y, z);

if x > y then

t ass x

x ass y

y ass t;

if x > z then

t ass x

x ass z

z ass t;

if y > z then

t ass y

y ass z

z ass t;

write(x, y, z);

}