package syntatic;

import java.io.IOException;

import java.util.Map;

import java.util.HashMap;

import lexical.Lexeme;

import lexical.TokenType;

import lexical.LexicalAnalysis;

import interpreter.command.Command;

import interpreter.expr.ConstExpr;

import interpreter.expr.Expr;

import interpreter.expr.FunctionCallExpr;

import interpreter.expr.Rhs;

import interpreter.util.AccessPath;

import interpreter.value.IntegerValue;

import interpreter.value.StringValue;

public class SyntaticAnalysis {

private LexicalAnalysis lex;

private Lexeme current;

public SyntaticAnalysis(LexicalAnalysis lex) throws IOException {

this.lex = lex;

this.current = lex.nextToken();

}

public Command start() throws IOException {

procCode();

matchToken(TokenType.END\_OF\_FILE);

return null;

}

private void matchToken(TokenType type) throws IOException {

//System.out.println("Match token: " + current.type + " == " + type + "?");

if (type == current.type) {

current = lex.nextToken();

} else {

showError();

}

}

private void showError() {

System.out.printf("%02d: ", lex.getLine());

switch (current.type) {

case INVALID\_TOKEN:

System.out.printf("Lexema inválido [%s]\n", current.token);

break;

case UNEXPECTED\_EOF:

case END\_OF\_FILE:

System.out.printf("Fim de arquivo inesperado\n");

break;

default:

System.out.printf("Lexema não esperado [%s]\n", current.token);

break;

}

System.exit(1);

}

//<code> ::= { <statement> }

private void procCode() throws IOException {

while (current.type == TokenType.IF

|| current.type == TokenType.WHILE

|| current.type == TokenType.SELF

|| current.type == TokenType.SYSTEM

|| current.type == TokenType.ARGS

|| current.type == TokenType.NAME) {

procStatement();

}

}

//<statement> ::= <if> | <while> | <cmd>

private void procStatement() throws IOException {

if (current.type == TokenType.IF) {

procIf();

} else if (current.type == TokenType.WHILE) {

procWhile();

} else {

procCmd();

}

}

//<if> ::= if '(' <boolexpr> ')' '{' <code> '}' [else '{' <code> '}' ]

private void procIf() throws IOException {

matchToken(TokenType.IF);

matchToken(TokenType.OPEN\_PAR);

procBolExpr();

matchToken(TokenType.CLOSE\_PAR);

matchToken(TokenType.OPEN\_CUR);

procCode();

matchToken(TokenType.CLOSE\_CUR);

if (current.type == TokenType.ELSE) {

matchToken(TokenType.ELSE);

matchToken(TokenType.OPEN\_CUR);

procCode();

matchToken(TokenType.CLOSE\_CUR);

}

}

//<while> ::= while '(' <boolexpr> ')' '{' <code> '}'

private void procWhile() throws IOException {

matchToken(TokenType.WHILE);

matchToken(TokenType.OPEN\_PAR);

procBolExpr();

matchToken(TokenType.CLOSE\_PAR);

matchToken(TokenType.OPEN\_CUR);

procCode();

matchToken(TokenType.CLOSE\_CUR);

}

//<cmd> ::= <access> ( <assign> | <call> ) ';'

private void procCmd() throws IOException {

procAccess();

if (current.type == TokenType.ASSIGN) {

procAssign();

} else if (current.type == TokenType.OPEN\_PAR) {

procCall();

} else {

showError();

}

matchToken(TokenType.DOT\_COMMA);

}

//<access> ::= <var> { '.' <name> }

private AccessPath procAccess() throws IOException {

int line = lex.getLine();

String name = procVar();

AccessPath path = new AccessPath(name, line);

while (current.type == TokenType.DOT) {

matchToken(TokenType.DOT);

name = procName();

path.addName(name);

}

return path;

}

//<assign> ::= '=' <rhs>

private void procAssign() throws IOException {

matchToken(TokenType.ASSIGN);

procRhs();

}

//

private FunctionCallExpr procCall(AccessPath path) throws IOException {

FunctionCallExpr fce = new FunctionCallExpr(path, lex.getLine());

matchToken(TokenType.OPEN\_PAR);

if (current.type == TokenType.FUNCTION

|| current.type == TokenType.NUMBER

|| current.type == TokenType.STRING

|| current.type == TokenType.OPEN\_PAR

|| current.type == TokenType.SYSTEM

|| current.type == TokenType.SELF

|| current.type == TokenType.ARGS

|| current.type == TokenType.NAME) {

Rhs rhs = procRhs();

fce.addParam(rhs);

while (current.type == TokenType.COMMA) {

matchToken(TokenType.COMMA);

rhs = procRhs();

fce.addParam(rhs);

}

}

matchToken(TokenType.CLOSE\_PAR);

return fce;

}

//<boolexpr> ::= [ '!' ] <cmpexpr> [ ('&' | '|') <boolexpr> ]

private void procBolExpr() throws IOException {

}

//<cmpexpr> ::= <expr> <relop> <expr>

private void procCmpExpr() throws IOException {

}

//<relop> ::= '==' | '!=' | '<' | '>' | '<=' | '>='

private void procRelop() throws IOException {

}

private ConstExpr procNumber() throws IOException {

int line = lex.getLine();

String tmp = current.token;

matchToken(TokenType.NUMBER);

int n = Integer.parseInt(tmp);

IntegerValue iv = new IntegerValue(n);

ConstExpr ce = new ConstExpr(iv, line);

return ce;

}

private String procName() throws IOException {

String name = current.token;

matchToken(TokenType.NAME);

return name;

}

private ConstExpr procString() throws IOException {

int line = lex.getLine();

String tmp = current.token;

matchToken(TokenType.STRING);

StringValue sv = new StringValue(tmp);

ConstExpr ce = new ConstExpr(sv, line);

return ce;

}

//<rhs> ::= <function> | <expr>

private Rhs procRhs() throws IOException {

Rhs rhs = null;

if (current.type == TokenType.FUNCTION) {

procFunction();

}

else{

rhs = procExpr();

}

return rhs;

}

//<function> ::= function '{' <code> [ return <rhs> ';' ] '}'

private Rhs procFunction() throws IOException {

Rhs rhs = null;

return rhs;

}

//<expr> ::= <term> { ('+' | '-') <term> }

private Expr procExpr() throws IOException {

Expr e = procTerm();

while (current.type == TokenType.ADD || current.type == TokenType.SUB) {

if (current.type == TokenType.ADD) {

matchToken(TokenType.ADD);

} else {

matchToken(TokenType.SUB);

}

procTerm();

}

return e;

}

//<term> ::= <factor> { ('\*' | '/' | '%') <factor> }

private Expr procTerm() throws IOException {

//ExprprocFactor();

while (current.type == TokenType.MULT

|| current.type == TokenType.DIV

|| current.type == TokenType.MOD) {

if (current.type == TokenType.MULT) {

matchToken(TokenType.MULT);

} else if (current.type == TokenType.DIV) {

matchToken(TokenType.DIV);

} else {

matchToken(TokenType.MOD);

}

}

return null;

}

//<factor> ::= <number> | <string> | <access> [ <call> ]

private void procFactor() throws IOException {

if (current.type == TokenType.NUMBER) {

procNumber();

} else if (current.type == TokenType.STRING) {

procString();

} else if (current.type == TokenType.OPEN\_PAR) {

matchToken(TokenType.OPEN\_PAR);

procExpr();

matchToken(TokenType.CLOSE\_PAR);

} else {

procAccess();

if (current.type == TokenType.OPEN\_PAR) {

procCall();

}

}

}

//<var> ::= system | self | args | <name>

private String procVar() throws IOException {

String var = null;

if (current.type == TokenType.SYSTEM) {

var = current.token;

matchToken(TokenType.SYSTEM);

} else if (current.type == TokenType.SELF) {

var = current.token;

matchToken(TokenType.SELF);

} else if (current.type == TokenType.ARGS) {

var = current.token;

matchToken(TokenType.ARGS);

} else{

var = procName();

}

return var;

}

}