НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

ФАКУЛЬТЕТ ІНФОРМАТИКИ І ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

**РОЗРАХУНКОВО-ГРАФІЧНА РОБОТА**

з дисципліни **«**Системне програмування 2**»**

Варіант 11

Виконав:

студент 3 курсу гр. ІО-71

Костинюк О. А.

Перевірив:

Павлов В. Г.

Київ 2019 р.

**Завдання:** Створити компілятор на основі циклу лабораторних робіт. На основі лабораторної роботи №8 створити генератор ассемблерного коду даного варіанту завдання та перевірити правильність його виконання. У компіляторі повинен бути виконаний лексичний, синтаксичний та семантичний аналіз даних.

Для виконання завдання використовувались 4, 5, 6 лабораторні роботи, в яких виконувався лексичний, синтаксичний та сематичний аналізи відповідно.

Програма працює в наступному режимі:

1. Створення токенів з переданого виразу
2. Виконання лексичних перевірок, які визначають наявність лексичних помилок(наприклад, назви змінних, що починаються з числа)
3. Створення таблиці лексем з створених токенів
4. Виконання синтаксичного аналізу утвореної таблиці лексем
5. Виконання семантичного аналізу
6. Формування готового асемблерного коду

Для перевірки правильності отриманого коду, використовується мова високого рівня(С++), в функцію \_\_asm() якої передається згенерований код

**Варіант** - 11

**Завдання**:



**Лістинг програми:**

'use strict';

const fs = require('fs');

const getReserved = (str, arrOfReserved) => {

const reservedUsed = [];

for (const reserved in arrOfReserved) {

if (str.includes(arrOfReserved[reserved]))

reservedUsed.push(arrOfReserved[reserved]);

}

return reservedUsed;

};

const deleteCopies = (arr1, arr2) => {

for (let i = 0; i < arr1.length; i++) {

for (let j = 0; j < arr2.length; j++) {

if (arr1[i] === arr2[j]) {

arr1.splice(i, 1);

i--;

}

}

}

return arr1;

};

const regex2 = /\W+\d+[a-zA-Z\_]+[a-zA-Z0-9\_]\*/;

const cyrillicPattern = /[\u0400-\u04FF]/;

const checkRules = expression => {

let copy = expression;

let match;

try {

match = copy.match(regex2)[0];

notDetected.push(match);

copy = copy.replace(match, '');

} catch (err) {

return copy;

}

return checkRules(copy);

};

const isUnresolved = (table, str) => {

let unresolved = str;

const tokens = [];

for (const lexem of table) {

tokens.push(lexem.token);

unresolved = unresolved.replace(lexem.token, '').trim();

}

unresolved = unresolved.split(' ');

if (Array.isArray(unresolved)) {

unresolved = unresolved[0];

}

const index = str.indexOf(unresolved);

return [unresolved, index];

};

if (str.match(cyrillicPattern)) {

console.log('Something went wrong.\nTry another expression.');

process.exit(1);

}

str = checkRules(str);

if (notDetected[0]) {

console.log(notDetected);

console.log('Something went wrong.\nTry another expression.');

process.exit(1);

}

const getTokens = (expressions, string = '') => {

let ids = expressions.map(el => el.match(/[A-Za-z\_][A-Za-z0-9\_]\*/g));

let floats = expressions.map(el => el.match(/[0-9]+[.][0-9]+/g)).flat();

floats = floats.filter(obj => obj);

//let integers = expressions.map(el => el.match(/\d+/g)).flat();

let integers = expressions.map(el => el.match(/[-+]?\d+/gm)).flat();

integers = integers.filter(obj => obj);

integers = eliminateFloatsFromInts(floats, integers);

const keyWords = getReserved(str, reserved);

ids = deleteCopies(ids.flat(), keyWords);

const singleSymb = symbolsAll.filter(el => el.length === 1);

const multipleSymb = symbolsAll.filter(el => el.length === 2);

return { keyWords, ids, floats, integers, symbolsUsed };

};

const getIndexes = (str, hash) => {

const result = [];

const exec = (str, hash) => {

for (const tokens in hash) {

const arrLength = hash[tokens].length;

for (let i = 0; i < arrLength; i++) {

let name;

let type;

const token = hash[tokens][i];

const length = token.length;

const index = str.indexOf(token);

let indexToken = symbolsAll.indexOf(token);

if (indexToken !== -1) {

name = symbolsTokenNames[indexToken];

for (const op in tokenss.symbol) {

if (tokenss.symbol[op][token]) type = op;

}

} else {

indexToken = keywordsAll.indexOf(token);

name = keywordsTokenNames[indexToken];

for (const op in tokenss.keyword) {

if (tokenss.keyword[op][token]) type = op;

}

}

if (!name) {

for (const arr in hash) {

const array = hash[arr];

if (array.includes(token)) {

name = arr;

}

}

name = name || 'ID';

type = 'ID';

}

result.push({ token, index, length, type, name });

if (typeof str === 'string') {

str = str.replace(token, '.'.repeat(length));

} else {

str = str.map(el => el.replace(token, '.'.repeat(length)));

}

}

}

};

exec(str, hash);

return result;

};

const main = () => {

const unresolved = isUnresolved(lexems, original);

if (unresolved[0]) {

errorLogging(`ERROR: Found unresolved symbol.

Symbol '${unresolved[0]}' at ${unresolved[1]} index`);

process.exit(1);

}

const getAssignments = lexTable => {

const lexemsAll = lexTable;

let semicolonBef = 0;

let semicolonAft = 0;

const indexesOfAssign = [];

lexTable.forEach((lexem, i, lexems) => {

if (lexems[i].name === 'ids') {

if (lexems[i - 1]) {

if (lexems[i - 1].type === 'type-operator') {

lexemsAll[i].Type = lexems[i - 1].token;

}

if (lexems[i - 1].name === 'T\_COMA\_SEPARATOR') {

for (let j = i; j > 0; j--) {

if (lexTable[j].name === 'T\_SEMICOLON') {

semicolonBef = lexTable[j].index + 1;

break;

} else semicolonBef = 0;

}

for (let j = i; j < lexems.length; j++) {

if (lexTable[j].name === 'T\_SEMICOLON') {

semicolonAft = lexTable[j].index;

break;

}

}

indexesOfAssign.push([semicolonBef, semicolonAft]);

}

}

}

});

return [...new Set(indexesOfAssign)];

};

const assignRanges = getAssignments(lexems);

const setTypes = (assignRanges, lexems) => {

const divorcedExp = [];

for (let i = 0; i < assignRanges.length; i++) {

const start = assignRanges[i][0];

const end = assignRanges[i][1];

divorcedExp[i] = [];

for (const lexem of lexems) {

if (i === 0 && lexem === lexems[0]) {

divorcedExp[i].push(lexems[0]);

}

if (lexem.index < end && lexem.index >= start)

divorcedExp[i].push(lexem);

}

}

divorcedExp.map(arr => {

arr.map((lexem, indexP, parent) => {

if (lexem.name === 'ids') {

lexem.Type = parent[0].token;

const index = lexems.indexOf(lexem);

lexems[index] = lexem;

}

});

});

return [lexems, divorcedExp];

};

const setArrayTypes = lexems => {

const original = lexems;

lexems.forEach((lexem, i, table) => {

if (lexem.token === ']') {

if (table[i - 1].type === 'ID' && table[i - 2].token === '[') {

if (table[i - 3].Type) {

original[i - 3].Type += `[${table[i - 1].token}]`;

}

} else {

errorLogging('ERROR: Array assignment at index', lexem.index + '');

}

}

});

return original;

};

const getOnlyAssignments = lexems => {

const assignments = [];

let indexes = [];

const indexesBeforeFirst = [];

lexems.forEach((lexem, i, table) => {

if (lexem.token === ';') {

if (

table[i - 1].name === 'integers' ||

table[i - 1].name === 'floats'

) {

indexes.push(lexem.index);

// }

}

}

});

return assignments;

};

const inizializationWithTypes = lexems => {

const variables = {};

for (const lexem of lexems) {

if (lexem.Type) {

const type = lexem.Type;

const typeName = type.replace(/[^A-Za-z]/g, '');

const keywords = Object.keys(tokenss.keyword['type-operator']);

if (!keywords.includes(typeName)) {

errorLogging(`ERROR: Wrong type: ${type} at ${lexem.index} index.`);

}

variables[lexem.token] = { type };

}

}

return variables;

};

const separationOfNamesAndValues = arr => {

for (let i = 0; i < arr.length; i++) {

indexes[i] = [];

for (let j = 0; j < arr[i].length; j++) {

if (arr[i][j] === '[') start = j;

if (arr[i][j] === ']') end = j;

}

}

return result;

};

const addValues = (arrOfExpr, obj, lexems) => {

const keys = Object.keys(obj);

const data = [];

arrOfExpr.forEach((expression, i) => {

data[i] = [];

expression.forEach(lexem => {

data[i].push(lexem.token);

});

});

const namesVar = [];

data.forEach(el => namesVar.push(el[0]));

for (let i = 0; i < keys.length; i++) {

if (!namesVar.includes(keys[i])) {

//const random10000 = randomIntFromInterval(-10000, 10000);

data.push([keys[i].toString(), '=', '1003' + '']);

}

}

let arrOfVar = separationOfNamesAndValues(data);

for (let i = 0; i < lexems.length; i++) {

for (let j = 0; j < arrOfVar.length; j++) {

if (lexems[i].token === arrOfVar[j].name) {

if (lexems[i].Type) {

if (lexems[i].Type[lexems[i].Type.length - 1] === ']') {

const num = lexems[i].Type.match(/\d+/)[0];

arrayIndex = num - 1

if (num - 1 < arrOfVar[j].index) {

//console.log(num - 1)

const variable = arrOfVar[j].name;

const index = arrOfVar[j].index;

errorLogging(

`ERROR: Index out of range: ${variable}[${index}]`

);

}

}

arrOfVar[j].type = lexems[i].Type;

}

}

}

}

const getOutput = table => {

for (const variable of table) {

const { name, type, value } = variable;

if (variable.type === 'double') {

const output = `'${name}': ${type} {${value}.0}`;

} else {

const output = `'${name}': ${type} {${value}}`;

}

}

}

const evalContentOfBrackets = exp => {

let str = exp;

let flag = true;

const results = [];

while (flag) {

let start = str.indexOf('[');

let end = str.indexOf(']');

const expression = str.substring(start + 1, end);

results.push({ from: start, to: end, expression });

str = str.replace('[', ';');

str = str.replace(']', ';');

start = str.indexOf('[');

end = str.indexOf(']');

if (start === -1) flag = false;

}

for (let i = 0; i < results.length; i++) {

const beginning = exp.slice(0, results[i].from + 1);

const last = exp.slice(results[i].to);

exp = `${beginning}${eval(results[i].expression)}${last}`;

}

return exp;

};

const calculationAsmCodeGen = (str, lexems) => {

lexems = lexems.map((lexem, i, arr) => {

if (lexem.value && arr[i + 1].token === ']') {

lexem.token = lexem.value;

}

if (lexem.name === 'integers' || lexem.name === 'floats') {

numbers.push(lexem.token);

}

return lexem;

});

const exps = str

.split(';')

.filter(el => !!el)

.map(el => el.trim());

//console.log(exps, lexems);

const before = [];

const after = [];

let scIndexes = [0];

const expsArr = [];

lexems.forEach(el => {

if (el.token === ';') {

scIndexes.push(el.index);

}

});

scIndexes = scIndexes

.map((el, i, arr) => {

if (i !== 0) {

return [arr[i - 1], arr[i]];

}

})

.filter(el => !!el);

for (let i = 0; i < scIndexes.length; i++) {

expsArr[i] = [];

}

lexems.forEach(el => {

for (let i = 0; i < scIndexes.length; i++) {

if (el.index >= scIndexes[i][0] && el.index <= scIndexes[i][1]) {

expsArr[i].push(el);

}

}

});

const parts = {

'\*': [],

'/': [],

'+': [],

'-': []

};

const signs = Object.keys(parts);

operations = {

multiplication: parts['\*'],

divide: parts['/'],

add: parts['+'],

substract: parts['-']

};

const calculations = (expression, table, variables) => {

let names = [];

for (const obj of table) {

names.push(obj.name);

}

names = Array.from(new Set(names));

let expressions = expression.split(';');

expressions.pop();

expressions = expressions.map(el => el.trim());

const ids = expressions.map(el => el.match(/[A-Za-z\_][A-Za-z0-9\_]\*/g));

const idsFlatted = Array.from(new Set(ids.flat()));

for (let i = 0; i < idsFlatted.length; i++) {

let includes = false;

for (let j = 0; j < names.length; j++) {

if (idsFlatted[i] === names[j]) includes = true;

}

if (!includes) {

console.log(`You haven't created the variable, ${idsFlatted[i]}`);

process.exit(0);

}

}

idLexems.forEach(el => {

if (el.token === ';') {

scIndexes.push(el.index);

}

});

scIndexes = scIndexes

.map((el, i, arr) => {

if (i !== 0) {

return [arr[i - 1], arr[i]];

}

})

.filter(el => !!el);

for (let i = 0; i < scIndexes.length; i++) {

expsArr[i] = [];

}

idLexems.forEach(el => {

for (let i = 0; i < scIndexes.length; i++) {

if (el.index >= scIndexes[i][0] && el.index <= scIndexes[i][1]) {

expsArr[i].push(el);

}

}

});

let divided = expsArr;

divided = divided.map(() => ({ before: [], after: [] }));

divided = divided.map(obj => {

obj.before.map(lexem => {

lexem.Type = variables[lexem.token].type;

return lexem;

});

obj.after.map(lexem => {

lexem.Type = variables[lexem.token].type;

return lexem;

});

return obj;

}

calculationAsmCodeGen(expression, lexems);

expForAssembly = expression;

const evalled = evalContentOfBrackets(expression);

expression = evalled;

const shorts = table

.filter(obj => obj.type === 'short')

.map(obj => {

if (obj.value.includes('.')) {

console.log(

`ERROR: You can't assign float to short variable. Variable ${obj.name}`

);

process.exit(1);

}

});

const floatInsBrac = expression.match(floatFinder);

if (floatInsBrac) {

console.log(

`ERROR: Index of array should be integer, find ${

floatInsBrac[0]

} at index ${actions.indexOf('[')}`

);

process.exit(1);

}

expression = expression.toLowerCase();

const hash = {};

for (let i = 0; i < splitted.length; i++) {

hash[splitted[i][0]] = splitted[i][1];

}

const results = [];

for (const el in hash) {

try {

const value = eval(hash[el]);

results.push({ name: el, value });

} catch (e) {

console.log(

`ERROR: You haven't define the variable to assign to ${el}`

);

process.exit(1);

}

}

const actions = process.argv[3];

lexems = setArrayTypes(lexems);

const isEndGood = lexems => {

if (!lexems[lexems.length - 1].token === ';') {

console.log("ERROR: Expression should be followed with ';'");

process.exit(1);

}

};

const arrOfAssign = getOnlyAssignments(lexems);

const variables = inizializationWithTypes(lexems);

const info = addValues(arrOfAssign, variables, lexems);

calculations(actions, info, variables);

// rgr

creationStage = JSON.parse(fs.readFileSync('help.json', 'utf-8')).data;

const helper = require('./doublepoint');

const makeIEE754Hex = (int, float) => {

const iee754 = helper.makeIEE754(int, float);

const hexIee754 = helper.binToHex(iee754);

const decIee754 = helper.hexToDec(hexIee754);

return decIee754;

};

splitted.forEach((arr, i, table) => {

if (arr[1].toString().length === 1) {

movupss.push('movups xmm' + counter + ', ' + arr[1][0]);

xmms.push({ ['xmm' + counter]: arr[1][0] });

counter++;

if (arr[0].toString().length === 1) {

const values = [];

xmms.forEach(obj => {

values.push(Object.values(obj)[0]);

});

if (!values.includes(arr[0].toString())) {

movupss.push('movups xmm' + counter + ', ' + arr[0][0]);

xmms.push({ ['xmm' + counter]: arr[0][0] });

counter++;

}

if (

arr[1].toString() !== str[str.length - 1] &&

arr[1].toString().length === 4

) {

//console.log({ar: arr[1]})

const arrName = arr[1][0].slice(0, arr[1][0].indexOf('['));

const arrIndex = arr[1][0].slice(

arr[1][0].indexOf('[') + 1,

arr[1][0].indexOf(']')

);

}

});

} else if (arr[0].toString().length > 1) {

movupss.forEach((str, index) => {

if (arr[0].toString() !== str[str.length - 1] || str === '') {

const arrName = arr[0][0].slice(0, arr[0][0].indexOf('['));

const arrIndex = arr[0][0].slice(

arr[0][0].indexOf('[') + 1,

arr[0][0].indexOf(']')

);

movupss[i] = 'mov esi, ' + arrIndex;

movupss[i + 1] =

'movups xmm' + counter + ', ' + '[4 \* esi] + ' + arrName;

counter++;

index--;

}

});

}

});

const sseTransform = {

'+': 'addss',

'-': 'subss',

'\*': 'mulss',

'/': 'divss'

};

const movHandlerCreation = (obj, i, table) => {

if (obj.index) {

return `

mov eax, ${makeIEE754Hex(obj.intPart, obj.floatPart)}

mov dword ptr[a+${obj.index \* 4}], eax // ${obj.name}[${obj.index}] = ${

obj.value

}

`;

}

return `

mov ${obj.name}, ${makeIEE754Hex(obj.intPart, obj.floatPart)} //${obj.name} = ${

obj.value

}`;

};

const output = creationStage.map((obj, i, arr) =>

movHandlerCreation(obj, i, arr)

);

output.push('\n\n');

let final = [output.join(''), movupss.join('\n')];

final = final.join('');

const time = new Date().getMinutes()ж

const path = './rgr-tests/generatedCodeAt' + time + '.asm';

fs.writeFileSync(path, final);

};

**Результат роботи програми:**

1. PS D:\js\systemProgramming> node rgr2.js 'double a[4], b; short n, d; a[1]= 15; a[1]= -3; a[3]= 5.3; a[1] = -10; n =3; d = -20;' 'b = 2 \* a[n]; b = d;'

Lexical analyze is correct

Syntax analyze is correct

Semantic anazyle is correct

{ results: [ { name: 'b', value: -20 } ] }

Assembly code generated successfully.

*Згенерований код:*

mov eax, 1097859072

mov dword ptr[a+4], eax // a[1] = 15

mov eax, 3225419776

mov dword ptr[a+4], eax // a[1] = -3

mov eax, 1084856729

mov dword ptr[a+12], eax // a[3] = 5.3

mov eax, 3240099840

mov dword ptr[a+4], eax // a[1] = -10

mov n, 1077936128 //n = 3

mov d, 3248488448 //d = -20

mov CONSTANT\_1, 1073741824 //CONSTANT\_1 = 2

movups xmm0, b

movups xmm1, d

movups xmm2, CONSTANT\_1

mov esi, 3

movups xmm3, [4 \* esi] + a

mulss xmm2, xmm3

movups b, xmm2

movups b, xmm1

*Результат виконання згенерованого коду:*



2. PS D:\js\systemProgramming> node rgr2.js 'double a[4], b; short n, d; a[1]= 15; a[2]= -3; a[3]= 5.3; a[1] = -10; n =3; d = -20;' 'b = 2 - a[n]; d = n;'

Lexical analyze is correct

Syntax analyze is correct

Semantic anazyle is correct

{ results: [ { name: 'b', value: -3.3 }, { name: 'd', value: 3 } ] }

Assembly code generated successfully.

*Згенерований код:*

mov eax, 1097859072

mov dword ptr[a+4], eax // a[1] = 15

mov eax, 3225419776

mov dword ptr[a+8], eax // a[2] = -3

mov eax, 1084856729

mov dword ptr[a+12], eax // a[3] = 5.3

mov eax, 3240099840

mov dword ptr[a+4], eax // a[1] = -10

mov n, 1077936128 //n = 3

mov d, 3248488448 //d = -20

mov CONSTANT\_1, 1073741824 //CONSTANT\_1 = 2

movups xmm0, b

movups xmm1, n

movups xmm2, d

movups xmm3, CONSTANT\_1

mov esi, 3

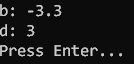
movups xmm4, [4 \* esi] + a

subss xmm3, xmm4

movups b, xmm3

movups d, xmm1

*Результат виконання згенерованого коду:*



3. PS D:\js\systemProgramming> node rgr2.js 'double a[4], b; short n, d; a[1]= 15; a[2]= -3; a[3]= 5.3; a[1] = -10; n =3; d = -20;' 'a[0] = 2 - a[2]; d = 6 / n;'

Lexical analyze is correct

Syntax analyze is correct

Semantic anazyle is correct

found a[0]

{ results: [ { name: 'a[0]', value: 5 }, { name: 'd', value: 2 } ] }

Assembly code generated successfully.

*Згенерований код:*

mov eax, 1097859072

mov dword ptr[a+4], eax // a[1] = 15

mov eax, 3225419776

mov dword ptr[a+8], eax // a[2] = -3

mov eax, 1084856729

mov dword ptr[a+12], eax // a[3] = 5.3

mov eax, 3240099840

mov dword ptr[a+4], eax // a[1] = -10

mov n, 1077936128 //n = 3

mov d, 3248488448 //d = -20

mov CONSTANT\_1, 1065353216 //CONSTANT\_1 = 0

mov CONSTANT\_2, 1073741824 //CONSTANT\_2 = 2

mov CONSTANT\_3, 1073741824 //CONSTANT\_3 = 2

mov CONSTANT\_4, 1086324736 //CONSTANT\_4 = 6

mov esi, 0

movups xmm1, d

movups xmm2, CONSTANT\_3

mov esi, 2

movups xmm3, [4 \* esi] + a

movups xmm4, CONSTANT\_4

movups xmm5, n

subss xmm2, xmm3

divss xmm4, xmm5

movups [4 \* 0] + a, xmm2

movups d, xmm4

*Результат виконання згенерованого коду:*



4. PS D:\js\systemProgramming> node rgr2.js 'double a[4], b; short n, d; a[1]= 20; a[2]= -20; a[3]= 5.3; a[0] = 10; n =3; d = -18;' 'a[3] = a[0] + a[1] - a[2];'

Lexical analyze is correct

Syntax analyze is correct

Semantic anazyle is correct

found a[3]

{ results: [ { name: 'a[3]', value: 50 } ] }

Assembly code generated successfully.

*Згенерований код:*

mov eax, 1101004800

mov dword ptr[a+4], eax // a[1] = 20

mov eax, 3248488448

mov dword ptr[a+8], eax // a[2] = -20

mov eax, 1084856729

mov dword ptr[a+12], eax // a[3] = 5.3

mov eax, 1092616192

mov dword ptr[a+0], eax // a[0] = 10

mov n, 1077936128 //n = 3

mov d, 3247439872 //d = -18

mov CONSTANT\_1, 1077936128 //CONSTANT\_1 = 3

mov CONSTANT\_2, 1065353216 //CONSTANT\_2 = 0

mov CONSTANT\_3, 1065353216 //CONSTANT\_3 = 1

mov CONSTANT\_4, 1073741824 //CONSTANT\_4 = 2

mov esi, 3

movups xmm0, [4 \* esi] + a

mov esi, 0

movups xmm1, [4 \* esi] + a

mov esi, 1

movups xmm2, [4 \* esi] + a

mov esi, 2

movups xmm3, [4 \* esi] + a

addss xmm1, xmm2

subss xmm1, xmm3

movups [4 \* 0] + a, xmm1

*Результат виконання згенерованого коду:*



5. PS D:\js\systemProgramming> node rgr2.js 'double a[4], b; short n, d; a[1]= 15; a[1]= -3; a[3]= 5.3; a[1] = -10; n =-10; d = -20;' 'b = 2 \* a[n]; b = d;'

Lexical analyze is correct

Syntax analyze is correct

ERROR: Index of a array should be >= 0 and <= 37/

6. PS D:\js\systemProgramming> node rgr2.js 'double a[4], b; short n, d; a[1]= 15; a[1]= -3; a[3]= 5.3; a[1] = -10; n =10; d = -20;' 'b = 2 \* a[n];'

Lexical analyze is correct

Syntax analyze is correct

ERROR: Index out of range: 2\*a[10]

7. PS D:\js\systemProgramming> node rgr2.js 'double a[4], b; short n, d; a[1]= 15; a[1]= -3; a[3]= 5.3; a[1] = -10; n =10; d = -20;' 'b = 2 \* a[4];'

Lexical analyze is correct

Syntax analyze is correct

ERROR: Index out of range: 2\*a[4]

8. PS D:\js\systemProgramming> node rgr2.js 'double a[4], b; short n, d; a[1]= 15; a[1]= -3; a[3]= 5.3; a[1] = -10; n =10; d = -20;' 'b = 2 \* c;'

Lexical analyze is correct

Syntax analyze is correct

You haven't created the variable, c

9. PS D:\js\systemProgramming> node rgr2.js 'double a[4], b; short n, d; a[1]= 15; a[1]= -3; a[3]= 5.3; a[1] = -10; n =10; d = -20;' 'd = 2 \* a[3];'

Lexical analyze is correct

Syntax analyze is correct

ERROR: Wrong expression.

You can't assign 'a' with 'double[4]' type

to variable 'd' with 'short' type.

10. PS D:\js\systemProgramming> node rgr2.js 'double a[4], b; short n, b; a[1]= 15; a[1]= -3; a[3]= 5.3; a[1] = -10; n =10; d = -20;' 'd = 2 \* a[3];'

Lexical analyze is correct

Syntax analyze is correct

ERROR: Variable b has already been declared

11. PS D:\js\systemProgramming> node rgr2.js 'double a[4], b; short n, d; a[1]= 15; a[1]= -3; a[3]= 5.3; a[1] = -10; n =10; d = -20;' 'b = 2 \* a[[n]]; b = d;'

ERROR: Something wrong with your brackets,

lexem '[' at index 10