**Лабораторна РОБОТА**

З дисципліни «Інженерія програмного забезпечення»

На тему: «Розробка генератора коду»

Варіант #8

|  |  |
| --- | --- |
| **Виконав**  Студент групи КВ-21  Комарницький О.Б.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | **Перевірила**  Доцент кафедри СП і СКС  Бояринова Ю. Є.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

**Варіант #8**

**Мета лабораторної роботи**

Метою синтаксичної роботи “Розробка генератора коду” є засвоєння теоретичного матеріалу та набуття практичного досвіду і практичних навичок розробки генераторів коду.

**Постановка задачі**

1. Розробити програму генератора коду для підмножини мови програмування SIGNAL згідно граматики за варіантом.
2. Програма має забезпечувати наступні дії:
   * + - Читання дерева розбору та таблиць, створених синтаксичним аналізатором, який було розроблено в розрахунково-графічній роботі.
       - Виявлення семантичних помилок
       - Генерацію коду та/або побудову внутрішніх таблиць.
3. Входом генератора коду (ГК) має бути наступне:
   * + - Дерево розбору;
       - Таблиці ідентифікаторів та констант повною інформацією, необхідною для генерації коду.
       - Вхідна програма підмножини мови програмування SIGNAL згідно з варіантом (необхідна для формування лістингу програми)
4. Виходом ГК має бути наступне:

* Асемблерний код, згенерований для вхідної програми та/або внутрішні таблиці для генерації коду;
* Внутрішні таблиці для генерації коду;

1. Зкомпонувати повний компілятор, що складається із розроблених раніше лексичного та синтаксичного аналізаторів і генератора коду, який забезпечує наступне:

* Генерацію коду та/або побудову внутрішніх таблиць;
* Формування лістингу вхідної програми з повідомленнями про лексичні, синтаксичні та семантичні помилки;

1. Входом компілятора має бути програма мовою програмування підмножини SIGNAL згідно із варіантом;
2. Виходом компілятора мають бути:

* Асемблерний код згенерований код для вхідної програми та/або внутрішні таблиці для генерації коду;
* Лістинг вхідної програми з повідомленнями про лексичні, синтаксичні та семантичні помилки;

1. Для програмування може бути використана довільна алгоритмічна мова програмування високого рівня. Якщо обрана мова програмування має конструкції або бібліотеки для роботи з **регулярними виразами**, то використання цих конструкцій та/або бібліотек **строго заборонено**.

**Зміст звіту**

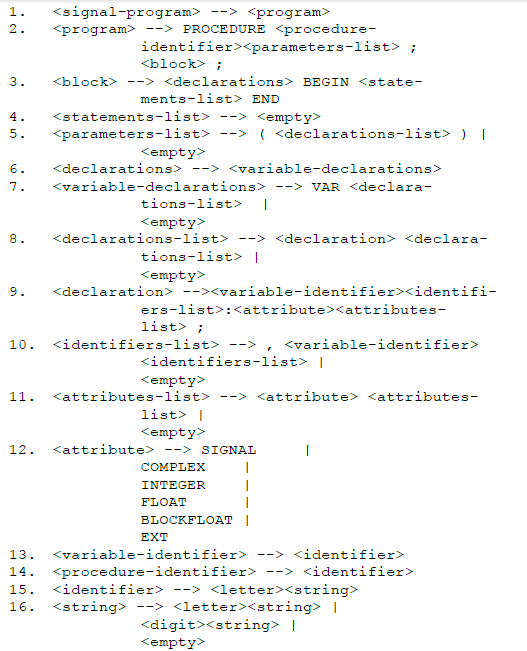
Зміст оформлюється згідно вимог до розрахунково-графічної роботи і має мати наступне:

* + - Титульний аркуш;
    - Індивідуальне завдання згідно до варіанту;
    - Лістинг програми ГК;
    - Контрольні приклади, необхідні для демонстрації всіх конструкцій заданої граматики, а також всіх можливих помилкових ситуацій. Опис кожного контрольного прикладу має містити вхідне дерево розбору і таблиці сформовані синтаксичним аналізатором, а також згенерований код та додаткові внутрішні таблиці ГК (якщо необхідно).

До звіту на довільному носії інформації мають бути додані:

* + - Файл з текстом звіту;
    - Проект працюючої програми мовою програмування (компілятора, що включає ЛА, СА та ГК) з вхідними кодами;
    - Завантажувальний модуль програми з необхідними файлами даних.

**Граматика мови**



**Лістинг програми ГК**

**Оголошення класу Generator:**

#include "Parser.h"

using namespace std;

struct \_identifier\_ /\* Сутність структури, якщо просто змінна: тип \*/

{ /\* описує будь-яку змінну \*/

string name;

string type;

};

struct \_proc\_structure\_ /\* Структура, що описує дерево розбору для мого варіанту \*/

{

\_identifier\_ identifier; /\* Ідентифікатор головної (єдиної) процедури \*/

list<\_identifier\_> parameters; /\* Список параметрів процедури \*/

list<\_identifier\_> declarations; /\* Список оголошених змінних \*/

};

class Generator

{

public:

Generator();

Generator(\_proc\_structure\_ );

~Generator();

void buildAsmCode();

private:

int getTypeSize(string );

int getStackSize();

string getAsmType(unsigned short int);

\_proc\_structure\_ TREE;

};

**Опис функцій класу Generator:**

#include "Generator.h"

Generator::Generator()

{}

Generator::Generator(\_proc\_structure\_ tree)

{

TREE = tree;

}

Generator::~Generator()

{

}

int Generator::getTypeSize(string type)

{

if (type == "SIGNAL") // Тип для виведення на порт

return 1;

else if (type == "COMPLEX") // Вважаємо, що ціла та уявна частина - float

return 8;

else if (type == "FLOAT") // як в С++

return sizeof(float);

else if (type == "INTEGER") // як в С++

return sizeof(int);

else if (type == "BLOCKFLOAT") // масиви у вигляді [n] не дозволені граматикою

return 4;

else if (type == "EXT") /\* Оскільки у цьому варіанті немає адресації до змінних \*/

return 4; /\* і лише одна можлива процедура, то приймемо як float \*/

else if (type == "PROCEDURE") // Тому що дального виклику

return 4;

return 0;

}

int Generator::getStackSize()

{

unsigned short int stack\_size = 0;

for each (\_identifier\_ identifier in TREE.declarations)

{

stack\_size += getTypeSize(identifier.type);

}

return stack\_size;

}

string Generator::getAsmType(unsigned short int bytes)

{

switch (bytes)

{

case 1:

return "db";

case 2:

return "dw";

case 4:

return "dd";

case 8:

return "dq";

case 16:

return "df";

case 32:

return "dp";

case 64:

return "dt";

default:

return "err";

}

}

void Generator::buildAsmCode()

{

std::ofstream f("Code.asm");

// Описуємо сегмент даних

f << ".data" << endl;

// Оголошуємо змінні, які передамо в нашу програму

for each(\_identifier\_ identifier in TREE.parameters)

{

f<<'\t'<<identifier.name<<'\t'<<getAsmType(getTypeSize(identifier.type))<<'\t'<<'?'<<endl;

}

// Описуємо сегмент коду

f << endl << ".code" << endl;

// Починаємо опис коду процедури

f << "PROC" << '\t' << TREE.identifier.name << "\tFAR" << endl;

// Зчитуємо параметри із стеку

f << "\t;Тут ми зчитуємо параметри із стеку" << endl;

//Визначаємо розмір параметрів, записаних у стек

int short unsigned stack\_size\_tmp = 0;

for each(\_identifier\_ identifier in TREE.parameters)

{

f << '\t' << identifier.name << '\t' << "equ \t" << "[EBP + " << 8 + stack\_size\_tmp << ']' << endl;

stack\_size\_tmp += getTypeSize(identifier.type);

}

f << "\n\tPUSH\tEBP;" << endl;

f << "\tMOV \tEBP, ESP;\t\t" << endl;

if (stack\_size\_tmp = getStackSize())

f << "\tSUB \tESP, " << stack\_size\_tmp << "; зарезервувати місце в стеку для локальних змінних" << endl;

f << "\t;тут команди можуть використовувати стек" << endl;

f << "\t;тут знаходиться розділ опису локальних змінних" << endl;

// Оголошуємо локальні змінні

stack\_size\_tmp = 0;

for each(\_identifier\_ identifier in TREE.declarations)

{

f << '\t' << identifier.name << "\t equ \t" << "[BP - " << getTypeSize(identifier.type) + stack\_size\_tmp << ']' << endl;

stack\_size\_tmp += getTypeSize(identifier.type);

//f << '\t' << identifier.name << '\t' << getAsmType(getTypeSize(identifier.type)) << "\t?" << endl;

}

f << endl << "\tPOP \tEBP;" << endl;

f << "\tRET \t" << getStackSize() << endl;

f << TREE.identifier.name << '\t' << "ENDP" << endl;

// Мітка початку коду

f << endl << "BEGIN:" << endl;

// Передаємо параметри в процедуру через стек

for each(\_identifier\_ identifier in TREE.parameters)

{

f << "PUSH " << identifier.name << ';' << endl;

}

// Викликаємо процедуру

f << "CALL " << TREE.identifier.name << ';' << endl;

// Мітка завершення програми

f << endl << "END BEGIN" << endl;

f.close();

}

**Програма компілятора:**

#include "Scanner.h"

#include "Parser.h"

#include "Generator.h"

int main()

{

Scanner scanner = Scanner();

if (!scanner.setFile("Program.txt"))

{

cout << "File can't be opened" << endl;

return 0;

}

list<int> codedLine = scanner.processFile();

cout << "All identifiers:" << endl;

scanner.showIdentifers();

cout << "Encoded line:" << endl;

for each (int symbol in codedLine)

cout << symbol << " ";

cout << endl;

Parser parser = Parser(codedLine, scanner.getIDENTIFIERSTable(), scanner.getKEYSTable());

parser.PRG();

cout << endl << "Compiled successfully ..." << endl;

Generator generator = Generator(parser.getTree());

generator.buildAsmCode();

return 0;

}

**Тестування компілятора:**

* Перевіримо коректний код програми:

PROCEDURE Proc1(myvar1: COMPLEX; var1, var2: SIGNAL);

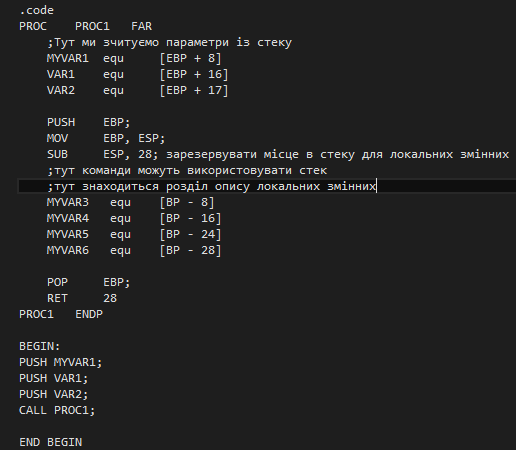
VAR myvar3, myvar4, myvar5: COMPLEX;

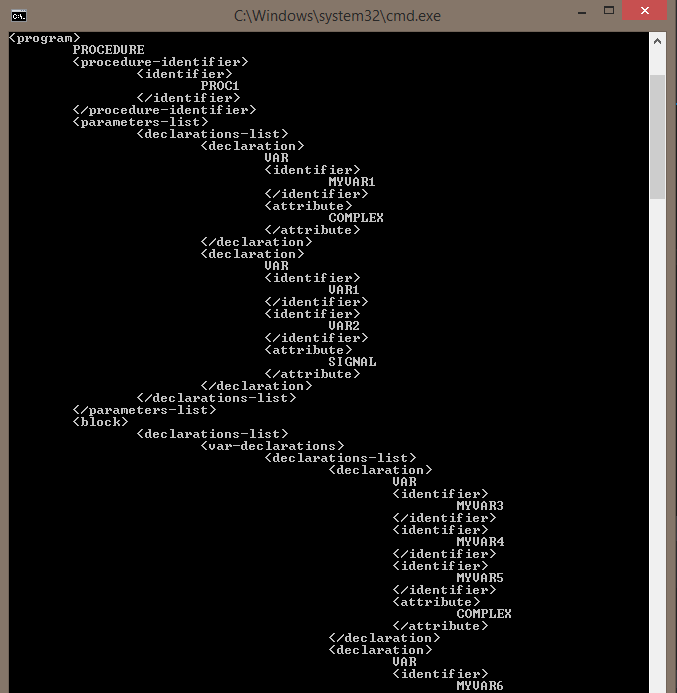
myvar6: INTEGER

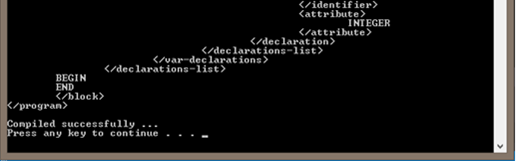
BEGIN

END;

**Код мовою асемблер**

 **Дерево розбору**





* Інші випадки не тестуються, бо в цьому варіанті неможливо припуститись семантичної помилки.