Міністерство освіти і науки України
Національний університет «Одеська політехніка»
Навчально-науковий інститут комп’ютерних систем
Кафедра системного програмного забезпечення

Агафонов Олег Олегович,
студент групи АС-161

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

Програмний продукт для навчання алгоритмізації

Спеціальність:
121 – Інженерія програмного забезпечення

Освітня програма:
Інженерія програмного забезпечення

Керівник:
Пригожев Олександр Сергійович,
канд. техн. наук, доцент

Одеса – 2021

ЗМІСТ

Міністерство освіти і науки України
Національний університет «Одеська політехніка»
Навчально-науковий інститут комп'ютерних систем
Кафедра системного програмного забезпечення

Рівень вищої освіти: другий (магістерський)
Спеціальність: 121 – Інженерія програмного забезпечення
Освітня програма: Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____
Любченко В. В.
«___» _____ 2021 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Агафонов Олег Олегович, студент групи АС-161

1. Тема роботи: Програмний продукт для навчання алгоритмізації

Керівник роботи: Пригожев Олександр Сергійович, канд. техн. наук, доцент

затверджені наказом ректора від «25» жовтня 2021 р. № 374-в

2. Зміст роботи:
   1) Проаналізувати існуючі рішення.
   2) Розробити алгоритм для пошуку помилок у програмі.
   3) Визначити вимоги до програмного продукту.
   4) Спроектувати структуру програмного продукту.
   5) Програмно реалізувати алгоритм та описати процес реалізації.
   6) Провести тестування та показати покриття тестами розроблені класи.

7) Провести розрахунок ефективності програмного продукту.

3. Перелік ілюстративного матеріалу: 1 слайд - титульний, 2 - предметна область, проблема, та мета, 3 - аналіз аналогів, 4 - процес роботи програми, 5 - алгоритм пошуку помилок, 6 - процес синтаксичного аналізу, 7 - варіанти використання, 8 - архітектура системи, 9 - класи для роботи з даними, 10 - діаграма програмних класів, 11 - тестування, 12 - показники ефективності, 13 - висновки, 14 - завершальний слайд.

4. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-6	Пригожев О. С. , канд. техн. наук, доцент		

5. Дата видачі завдання: «30» серпня 2021 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання	Примітка
1	Аналіз аналогів	12.09.21	виконано
2	Розробка алгоритму	20.09.21	виконано
3	Специфікація вимог до системи	29.09.21	виконано
4	Проектування системи	08.10.21	виконано
5	Розробка програмного продукту	16.10.21	виконано
6	Тестування	04.22.21	виконано
7	Розрахунок ефективності	06.22.21	виконано

Здобувач вищої освіти О. О. Агафонов

Керівник роботи О. С. Пригожев

АНОТАЦІЯ

Метою роботи є прискорення процесу навчання розробників таким поняттям як команди, оператори, змінні та цикли. Технологіями розробки є Intellij Idea як середовище розробки, Java як мова програмування, розроблений графічний редактор як базовий фреймворк та бібліотека JUnit для проведеного тестування.

Як результат роботи виконано програмну реалізацію продукту для навчання алгоритмізації, пошуку помилок та генерації програмного коду мовою C++.

Ключові слова: схема алгоритмів, блок-схема, програмування, візуалізація процесів всередині системи, алгоритм, пошук помилок, генерація коду.

ABSTRACT

The purpose of the work is to accelerate the process of training developers with such concepts as commands, operators, variables and loops. The development technologies are Intellij Idea as a development environment, Java as a programming language, developed graphic editor as a basic framework and the JUnit library for testing.

As a result of the work, the software implementation of the product for learning algorithmization, error searching, and C++ code generation.

Keywords: algorithm scheme, flowchart diagram, programming, visualization of processes within the system, algorithm, error search, code generation.

ВСТУП

Нестача кваліфікованих IT-фахівців в Україні спостерігається вже кілька років. Згідно з дослідженнями, проведеними компанією GlobalLogic [1], в середньому кількість вакансій для IT-фахівців зростає майже на 30% в рік, коли кількість самих інженерів лише на 18%. На рисунку 1.1 вказано статистику одного з сайтів для пошуку вакансій та кандидатів.

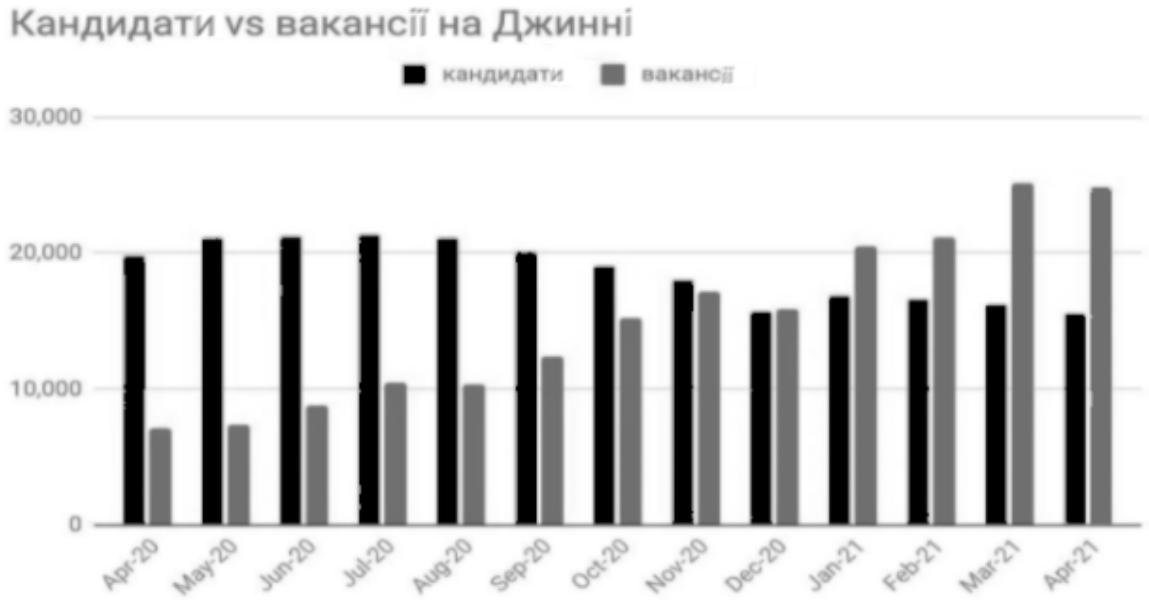


Рисунок 1.1 - Кількість кандидатів і вакансій на djinni.co в IT розділі

Це явище має складну природу з безліччю причин, але як найголовнішу з них варто виділити процес *діджиталізації* бізнес-компаній України, а також темп цього процесу.

Діджиталізація - це процес впровадження сучасних технологій у основні трудові процеси підприємства з метою підвищення їхньої швидкості, надійності, захищеності від зовнішніх факторів, або здешевлення цих процесів, що матиме для компанії в майбутньому позитивні наслідки.

Цей процес торкається майже усіх галузей підприємства - зберігання даних, маркетинг та реклама, комунікація з клієнтами, постачальниками та партнерами - і займає якийсь час на втілення. Враховуючи багатогранність процесу, для його реалізації необхідні кваліфіковані ІТ-фахівці з різних сфер: маркетинг, WEB-дизайн та WEB-аналітика, проектування, розробка та тестування спеціалізованого для підприємства ПЗ, дизайн, а також фахівці, які забезпечують безпеку та надійність впроваджених технологій.

Конкуренція спонукає компанії розвиватися, і процес діджиталізації – один із проявів такого розвитку – відкриває вакансії для найму спеціалістів із широкого ІТ-спектру.

Пандемія зробила цей процес необхідністю для виживання багатьох бізнес-фірм і стала потужним каталізатором для розвитку ІТ-ринку України через щонайменше необхідність створення нових або вдосконалення існуючих каналів зв'язку з клієнтами. Зростаюча необхідність нових сучасних засобів для ведення бізнесу з дому ще більше розширила ринок вакансій для ІТ-фахівців, що й призвело до сьогоднішньої ситуації на ринку.

Грунтуючись на даних про курси, присвячені програмуванню [2] [3], на думці інших людей [4], та на особистому досвіді, можна зробити висновок, що вивчення нової мови програмування займає в середньому 2-4 місяці за умови, що людина знає будь-яку іншу мову програмування, і 5-7 місяців, якщо людина стикається з програмуванням вперше [5]. Числа можуть сильно змінюватись в залежності від складності мови та темпу вивчення, тому надалі будуть використані середні значення. Так чи інакше, досвід, набутий людиною іншою мовою програмування, може у кілька разів скоротити час навчання нової мови. Очевидно, саме з цієї причини навчання людини, яка не знайома з програмуванням, триває в середньому на 3 місяці довше, адже людині потрібно вивчити основні концепції програмування, які так чи інакше зустрічаються в різних мовах програмування, хоч і в різній імплементації.

На даний момент програмування охоплює великий спектр задач, які можна за його допомогою вирішити, і пропонує відповідний інструментарій у вигляді мов програмування, кількість яких за різними оцінками варіюється від 150 [6] до 9000 [7]. Кожна мова має власну мету, власну область завдань, які можна за допомогою цієї мови вирішити, і свій набір інструментів, пропонованих програмісту для їх вирішення. Не зважаючи на велику різноманітність мов, існують деякі поняття і концепції, які є загальними для багатьох мов програмування, хоч і мають в них різну програмну реалізацію, і які програміст повинен знати.

Отже, програміст-початківець витрачає десь 3 місяці на вивчення таких концепцій програмування, як:

- команди послідовність їх виконання;
- типи даних і можливі операції з ними;
- змінні і оператори;
- розгалуження і цикли;
- функції та їх виклик.

Проблема - програмісти-початківці витрачають багато часу на вивчення основ, які зустрічаються в більшості мов програмування.

Мета даної розробки - прискорити процес навчання для учнів основам програмування шляхом розробки графічного редактора для схем алгоритмів з функціями пошуку помилок та генерацією коду.

1 ПРОБЛЕМИ ІСНУЮЧИХ ЗАСОБІВ ДЛЯ НАВЧАННЯ

1.1 Аналіз існуючих рішень

На даний момент існує безліч програмних засобів для навчання програмуванню. Такі програми як Udemy, EdX, Codemurai, Encode, а також інші програми-збірники курсів і лекцій, створених професійними викладачами, роблять свій безумовний внесок у процес навчання програміста, тому мета розробки - доповнити ці програми, а не замінити їх.

Візуалізація грає незаперечну роль в процесі навчання, дозволяючи швидше зрозуміти матеріал [8]. Більшість типових завдань, що вирішуються програмістом в період навчання основам, можна представити у вигляді алгоритму, а отже - зобразити у вигляді схеми алгоритму. Програму, представлену в такому вигляді, набагато легше зрозуміти новачкам і розібратися в темі. Деякі програми-збірки мають у собі пісочницю, у якій можна запустити написаний користувачем код, але жодна з них не дозволяє візуалізувати процес роботи програми, що може прискорити процес розуміння основ.

Необхідно розрізняти візуальні засоби розробки і графічні мови програмування. До перших відносяться програмні продукти, що використовуються в процесі розробки - наприклад, графічні редактори для створення діаграми класів і засоби для проектування графічного інтерфейсу. Графічні мови програмування дозволяють розробляти програмне забезпечення без написання коду, або з написанням дуже малої кількості цього коду.

Навчання за допомогою візуальних мов програмування має недолік, що є частиною природи візуального програмування - відсутність коду, а точніше, відсутність необхідності писати код погано сприяє процесу набуття навичок програмування, тому потенційне вирішення проблеми має спонукати користувача писати код.

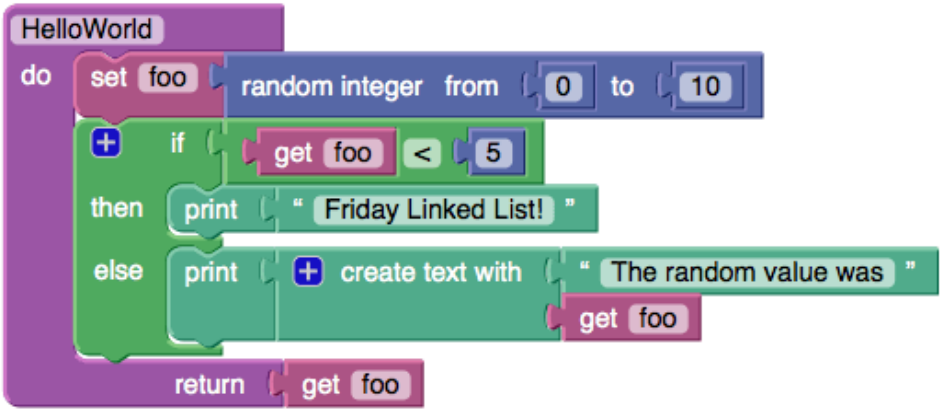


Рисунок 1.2 - Графічна мова програмування Scratch

За допомогою графічних редакторів користувач може зобразити за допомогою схеми алгоритмів будь-яку просту програму, але жоден графічний редактор не перевіряє, що написано в блоках.

Вирішення проблеми повинне мати функцію візуалізацію роботи програми, повинно змушувати користувача писати код, і при цьому цей код перевіряти. Більше того, користувач повинен мати можливість запустити створений ним код, щоб переконатися, що він працює, як планувалося.

Графічний редактор, найбільш підходящий за описом - AFCE - це безкоштовна освітня програма, що дозволяє створювати схеми алгоритмів та генерувати на її основі код.

AFCE дозволяє:

1) Візуалізувати прості програми у вигляді схеми-алгоритму

2) Генерувати на основі розташування блоків та тексту в них код для реальних мов програмування (C, C++, Ruby, Paskal, PHP та ін.)

Проте функціонал AFCE має один великий недолік - програма не перевіряє введений користувачем текст на помилки, тому згенерований код може не запуститися.

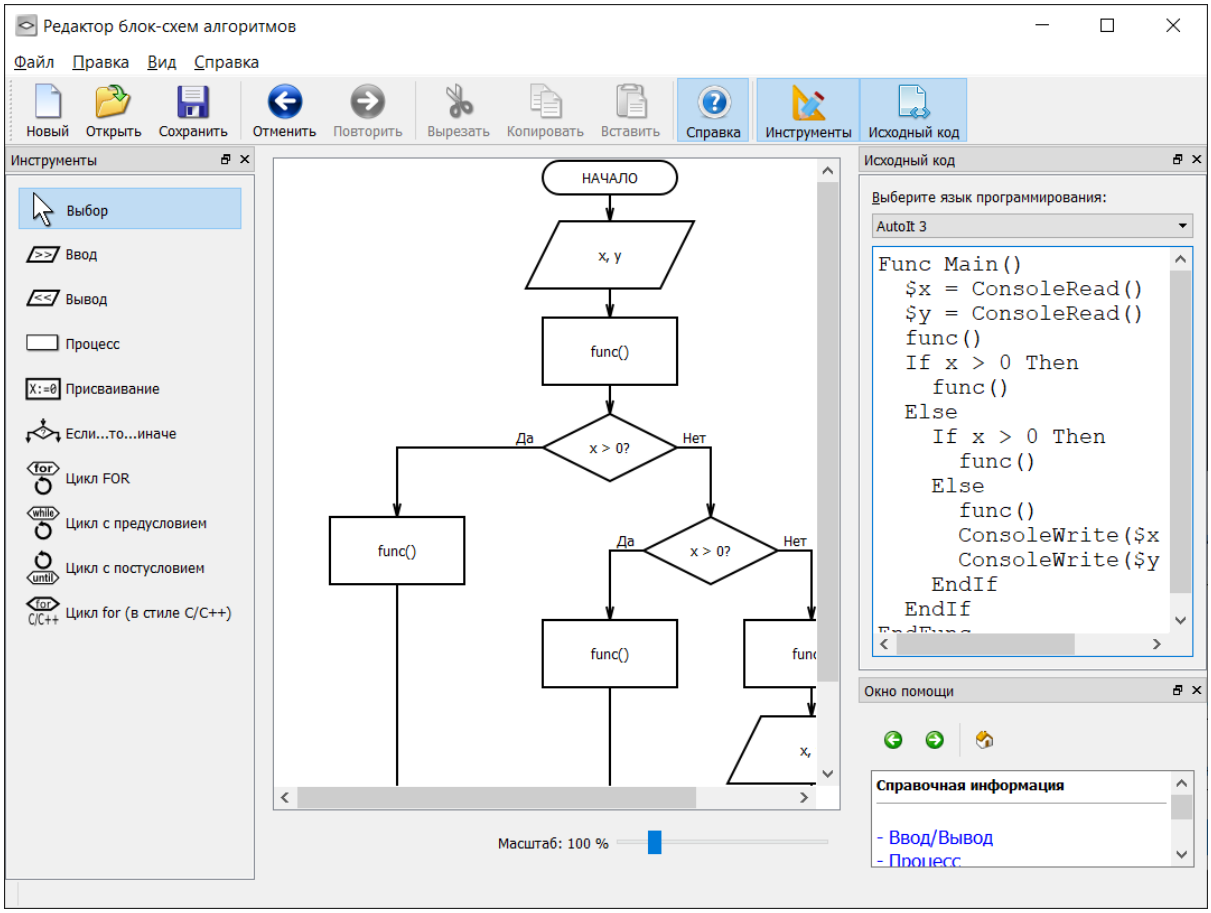


Рисунок 1.3 - Приклад роботи AFCE

Винесемо всі існуючі рішення до таблиці 1.1 та проведемо їх аналіз.

Таблиця 1.1 - Порівняння рішень

Атрибут	Програми-збірники курсів	Візуальні МП	Графічні редактори	AFCE	**Scheme Prof**
Візуалізація програми	-	+	+	+	+
Необхідність писати код	+	-	+	+	+
Перевірка коду/тексту в середині блоків	+	+	-	-	+

Програмне забезпечення, яке буде розроблено - **Scheme Prof** - має задовольняти всім переліченим потребам. Програмне забезпечення **Scheme Prof** має бути графічним редактором, який навчає основам програмування з внутрішніми засобами для візуалізації написаної програми і генерації коду для існуючого мови програмування. Це програмне забезпечення має використовуватися разом із програмами-збірниками лекцій для прискорення процесу навчання основам програмування.

1.2 Процес візуалізації та принцип роботи рішення

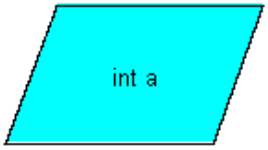
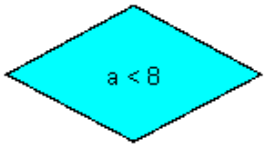
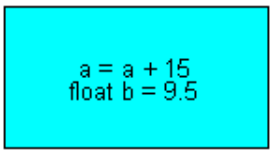
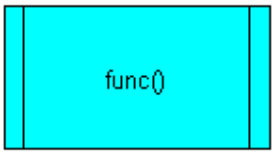
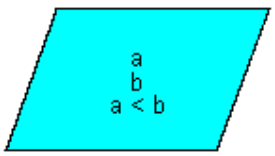
Найкращим способом для візуалізації процесу роботи програми є схеми алгоритмів, оскільки має всі необхідні засоби для візуалізації програмних команд, їх послідовності, а також для візуалізації таких ключових процесів як розгалуження, цикли, введення та виведення даних, ознаменування початку та кінця програми.

Правила створення схем алгоритмів регламентуються стандартом ГОСТ 19.701-90 [9]. Даний стандарт містить широкий спектр можливих функціональних блоків, які можуть бути використані в схемах алгоритмів, але для поставленої мети важливі лише деякі з них, що наведено у таблиці 1.2.

Таблиця 1.2 - Блоки, які можна використовувати в схемі

Умовне позначення	Назва блоку	Функція в ГОСТ	Код, який буде генеруватися
	Термінатори початку і кінця	Символи для позначення вхідної та кінцевої точки алгоритму	Відкривання і закривання тіла функції

Продовження таблиці 1.2 - Блоки, які можна використовувати в схемі

Умовне позначення	Назва блоку	Функція в ГОСТ	Код, який буде генеруватися
	Введення/виведення	Введення користувачем даних	Оголошення змінної і присвоєння їй значення, яке ввів користувач
	Умова	Поділ поточної гілки алгоритму на дві нових	Умовний оператор “if/else”
	Операція	Операція над даними	Операція, яка проводиться з даними
	Зовнішній процес	Виклик зовнішньої процедури, що визначена іншою схемою алгоритмів	Виклик функції, згенерованої на основі схеми алгоритмів, яка перебуває всередині
 	Цикл	Початок і кінець сегмента алгоритму, що повторюється при певній умові	Цикл “While”
	Введення/виведення	Виведення даних (на екран / зовнішній накопичувач)	Виведення даних на екран

Отже, схема алгоритмів - графічне представлення для аналізу або методу розв'язання задачі, в якому використовують графічні позначення для відображення даних і операцій з ними. Схема алгоритмів містить в собі певну кількість символів-блоків з пояснювальним текстом всередині, а також лінії, які об'єднують ці блоки в єдину картину.

Як базове програмне забезпечення, яке буде модифіковано в процесі розробки шляхом додавання нового функціоналу, було обрано графічний редактор, обмежений на даний момент функціоналом створення/редагування схем алгоритмів.

Програма буде дозволяти користувачеві створювати власну схему алгоритмів та розташувати в її блоках псевдокод, а після цього шукатиме помилки як у схемі в цілому, так і в коді блоків зокрема. Якщо у створеній користувачем схемі не було виявлено помилок, програма генерує код для існуючої мови програмування на основі псевдокоду та порядку розташування блоків.

Грунтуючись на тому, що у програмі вже є базовий функціонал для створення та редагування схем, розробка буде сконцентрована саме на перевірці схеми на помилки та генерацію коду. Для виконання цього завдання необхідно розробити алгоритм, який буде аналізувати розташування блоків, зв'язки між ними та написаний псевдокод в блоках, та завдяки якому буде проведена повна перевірка схеми на скоєні помилки та генерація коду для існуючої мови програмування у разі відсутності помилок.

2 МЕТОДИ ПОШУКУ ПОМИЛОК У ПРОГРАМІ

2.1 Структурні помилки та передача управління

Алгоритм пошуку помилок представлений у загальному вигляді на рисунку 2.1. Кожен із блоків на рисунку має в собі один або кілька інших алгоритмів, деякі з яких будуть розглянуті в ході цього розділу.

У схемі алгоритмів блоки є командами, які необхідно виконати, а стрілки між блоками показують послідовність передачі управління від однієї команді до іншої. Незважаючи на очевидні подібності з представленням програми в вигляді програмного коду, програма, представлена у вигляді блок-схеми, має низку своїх особливостей, які необхідно розуміти та контролювати. По-перше, навіть якщо програми в такому форматі і дозволяють швидко освоїтися в темі, вони мають очевидний недолік - передача управління між командами не може відбуватися довільно і повинна мати свої обмеження.

Мета алгоритму пошуку помилок (рис. 2.1) - перебрати кожен блок, щоб додати команди, що містяться в ній, до загального блоку тексту, а також перебрати кожну стрілку, щоб переконатися, що передача управління адресованій команді відбувається коректно.

Контекст - це набір функцій, змінних та властивостей, доступних для звернення в конкретному блоці коду. Зазвичай у програмуванні області контексту відкриваються і закриваються фігурними дужками "{" та "}". Змінні, оголошені у внутрішньому блоці коду, наприклад, усередині гілок умовних операторів або всередині циклів, недоступні у зовнішньому блоці. Управління послідовно передається від однієї команди до іншої, і при зустрічі умовного або циклічного операторів передається у внутрішній блок цього оператора. Такі мови як Fortran і Assembler дозволяють довільно використовувати оператор goto, даючи можливість передавати управління в довільну точку програми, тоді як у більш високорівневих мовах його

використання дуже обмежене - управління не можна передавати всередину циклу, гілки умовного оператора або функції. У Java взагалі оператор goto не має функціоналу, хоч його слово зарезервовано. До того ж практика використання цього оператора є поганим тоном.

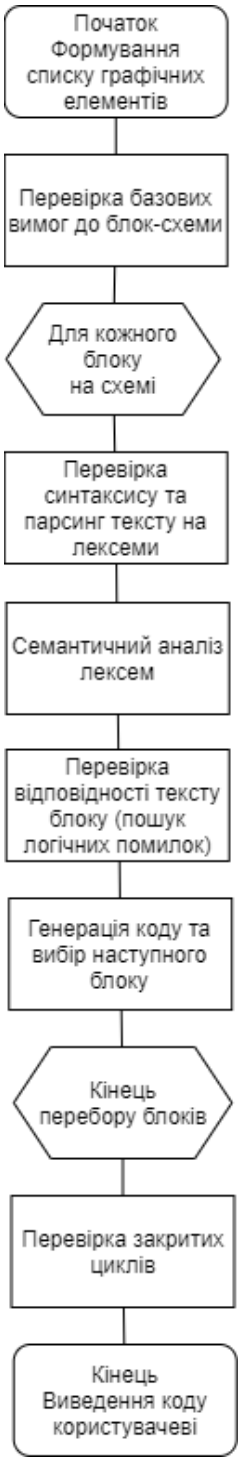


Рисунок 2.1 – Коротке представлення алгоритму пошуку помилок

У програмі у вигляді блок-схеми передача управління візуалізується у вигляді стрілки між блоками, тому необхідно виявляти зроблені користувачем помилки, пов'язані з передачею управління.

Наприклад, не можна передавати управління команді всередині гілки умовного оператора або всередині циклу, тому що такий код не може існувати навіть використовуючи конструкцію goto. Тому такі структури, як на рисунку 2.2, мають бути розпізнані як помилки, пов'язані з передачею управління.

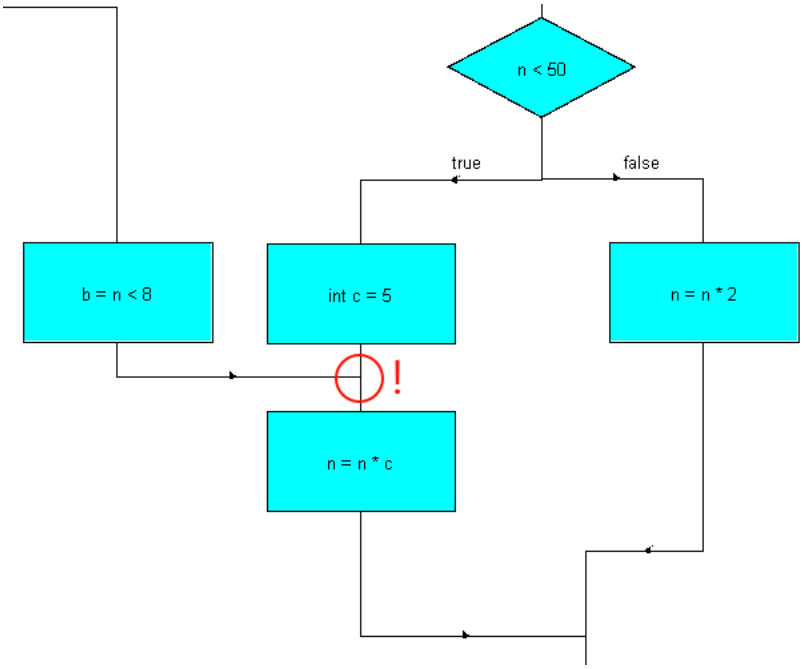


Рисунок 2.2 – Неможлива передача управління всередину гілки умовного оператора

Схожий випадок відбувається з передачею управління вже виконаній команді як на рисунку 2.3. Хоча в даному випадку це дозволяється у багатьох мовах програмування, така практика є поганим тоном і має бути розпізнана як помилка.

Ще один випадок некоректної передачі управління - самозамикаючі структури, як на рисунку 2.4. Ці структури є окремим випадком помилкової

передачі управління виконаній команді, і повинні бути розпізнані як помилка.

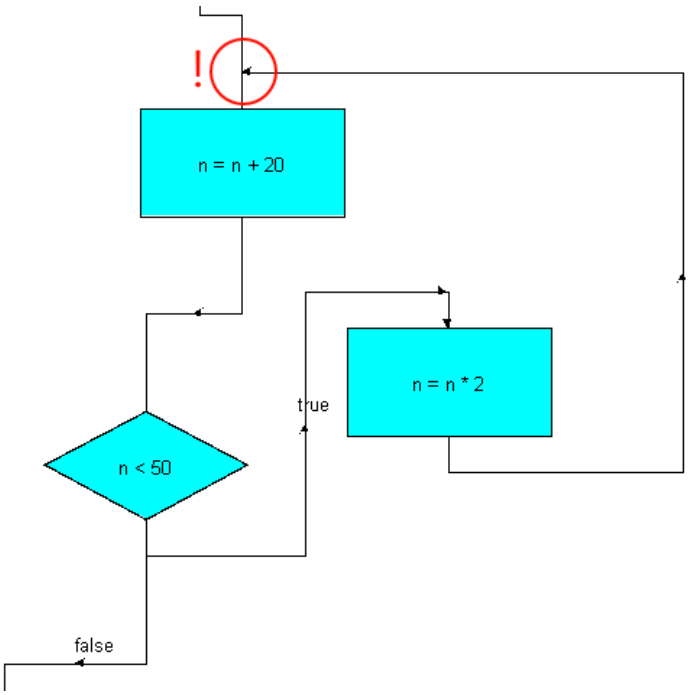


Рисунок 2.3 – Небажана передача управління вже виконаній команді

Крім помилок передачі управління, необхідно виділити інші структурні помилки, пов'язані з розміщенням блоків і зв'язків між ними:

- на схемі відсутній блок початку або кінця;
- на схемі більше одного блоку початку або блоку кінця;
- у схемі присутні розриви, внаслідок чого на схемі є блок або група блоків, до яких неможливо дістатися з блоку початку, або з яких не можна дістатися блоку кінця;
- у блоку умов не 2 виходи, а більше чи менше;
- у блоків не-умов більше одного виходу.

Можна зменшити кількість можливих помилок, спроектувавши інтерфейс так, щоб користувач не міг робити деякі з них:

- відключати кнопки додавання блоку початку та кінця, якщо такі на схемі вже є;

- відключати кнопки створення стрілок між якимись блоками, якщо блок більше не може мати вихідних зв'язків.

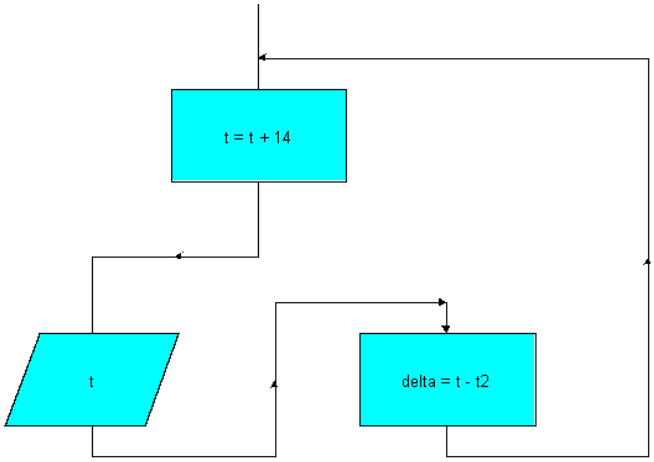


Рисунок 2.4 – Заборонена самозамикаюча структура

Остання особливість візуалізації програми у форматі блок-схеми – це цикли. У вигляді програмного коду циклічність має вигляд одного оператора while або for, візуальне представлення циклічності повинно мати 2 блоки - блок початку і блок кінця (блок початку має умову, за якої передача управління повертається до початку раз за разом).

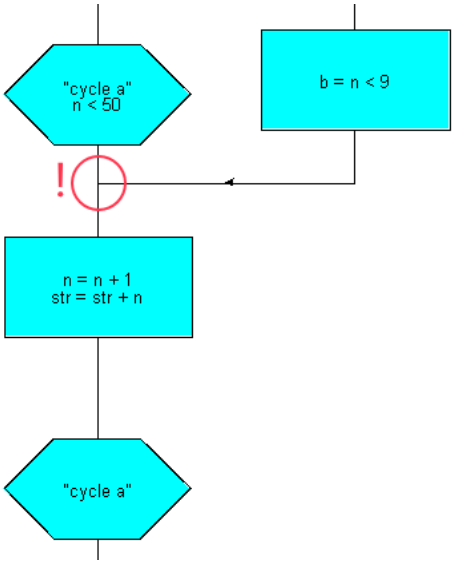


Рисунок 2.5 – Заборонена передача управління всередину тіла циклу

Як було сказано раніше, управління не може бути довільно передано в середину циклу (рис. 2.5), так само як і залишати його, не пройшовши через блок кінця циклу. Крім очевидної необхідності наявності у циклу як блоку початку, так і блоку кінця, вкладені циклічності повинні закриватися в послідовності, зворотній тій, в якій відкривалися.

Враховуючи свою природу, структурні помилки повинні бути перевірені раніше інших, тому що їх наявність робить генерацію коду неможливою і подальші перевірки марними.

2.2 Пошук структурних помилок

Перед тим, як зіставляти послідовність для перевірки блоків і шукати помилки в передачі між ними управління, необхідно переконатися, що виконані всі базові вимоги до блок-схеми.

Як базове програмне забезпечення було взято графічний редактор для схем алгоритмів. Реалізація блоків була виконана з використанням поліморфізму (рис. 2.6), тобто кожен тип блоків у таблиці 1.2 має свій власний клас з деякими переписаними методами (наприклад, форма малювання на екрані, або визначення влучення курсором по об'єкту під час кліку). Крім того, всі класи блоків мають загального класу-батька AbstractDiagramNode. Аналогічно зв'язки були спроектовані за таким же принципом - стрілка з написом і стрілка без напису успадковувалися від класу AbstractDiagramLink. У свою чергу, клас-батько для блоків AbstractDiagramNode, клас-батько для стрілок AbstractDiagramLink і клас для написів над стрілками DiagramLabel були успадковані від суперкласу DiagramObject. Отже, всі графічні елементи на схемі є нащадками класу DiagramObject.

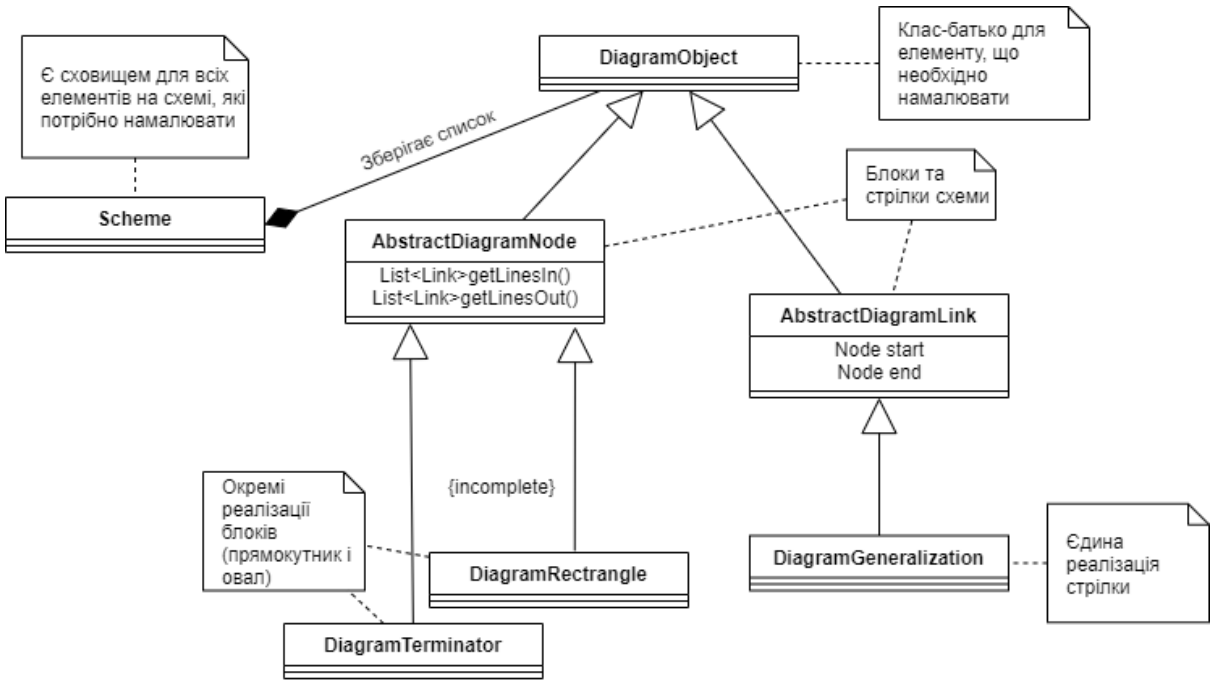


Рисунок 2.6 - діаграма програмних класів, властивості та методи яких будуть використані в алгоритмі пошуку помилок та перебору блоків

Всі класи блоків мають метод, що повертає список стрілок, що входять до блоку, і аналогічний метод для списку тих, що виходять. Кожна стрілка має два поля, в яких зазначено блоки, які ця стрілка пов'язує. Таким чином, якщо всі елементи на схемі пов'язані, за допомогою перебору можна дістатися будь-якого блоку до іншого.

У момент запуску перевірки створюється список DiagramObject елементів із усіх присутніх на схемі елементів (і блоки, і стрілки між ними).

Етап перевірки базових вимог до блок-схеми. На цьому етапі перевіряються такі помилки:

- на схемі відсутній блок початку або блок кінця;
- на схемі більше одного блоку початку або блоку кінця;
- у схемі присутні розриви, внаслідок чого на схемі є блок або група блоків, до яких неможливо дістатися з блоку початку, або з яких не можна дістатися блоку кінця;
- у блоку умов не 2 виходи, а більше чи менше;

- у блоків не-умов більше одного виходу.

Оскільки кожному типу блоку відповідає свій клас, для визначення типу блоку досить поглянути на клас, екземпляром якого є якийсь блок у списку. Користуючись тим, що кожен блок має дані про пов'язані з ним стрілки, а кожна стрілка має дані про пов'язані з нею блоки, синтезуємо два нових алгоритми:

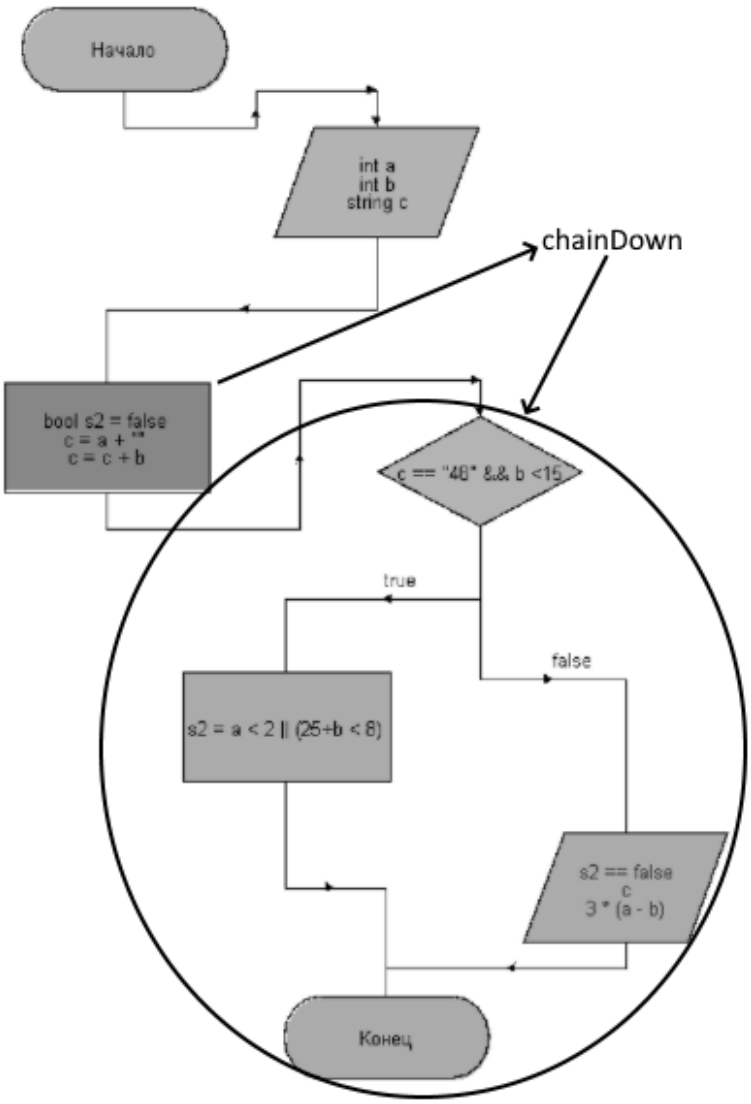


Рисунок 2.7 – Результатом алгоритму ChainDown для третього зверху блоку є множина блоків в обведеній області

1) Перебираючи множину стрілок, що виходить від якогось конкретного блоку, можна за допомогою рекурсії скласти список блоків, до яких можна дістатися з вихідного, переходячи від блоку до блоку аж до блоку-кінця (далі цей алгоритм називатиметься *ChainDown*, див. рисунок 2.7).

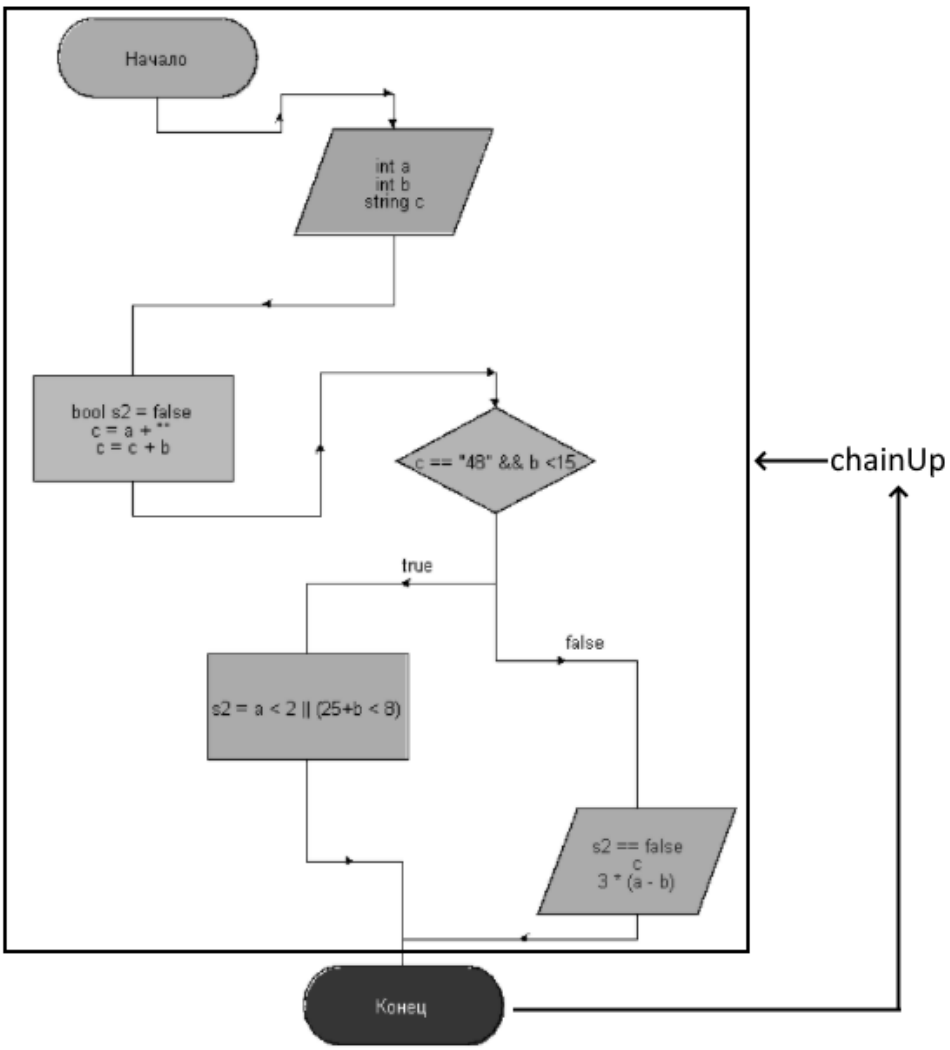


Рисунок 2.8 – Результатом алгоритму для блоку-кінця схеми є обведена множина блоків

2) Аналогічно, перебираючи множину вхідних стрілок, можна скласти список всіх блоків, від яких можна потрапити до цього блоку,

переходячи від одного до іншого аж до блоку-початку (далі цей алгоритм називатиметься *ChainUp*, див. рисунок 2.8).

Ці два алгоритми будуть використовуватися в подальшому для пошуку розривів та конструкцій, що самозамикаються.

Алгоритм перевірки на базові відповідності виглядає так:

1) Перевіряємо відсутність чи надмірну кількість екземплярів класів DiagramTerminatorStart та DiagramTerminatorEnd, які відповідають блокам початку та кінця відповідно;
2) Для того, щоб виявити непов'язані зі схемою блоки, ми порівнюємо список зв'язаних блоків з блоком початку за допомогою ChainDown, та список усіх блоків на схемі. Якщо якийсь із блоків не потрапив до списку, отриманого за допомогою ChainDown, отже, до нього неможливо дістатися з блоку початку, отже, він відірваний від загальної схеми.
3) Для того, щоб виявити самозамикаючі конструкції, необхідно застосувати ChainUp для блоку кінця. Відсутність якогось блоку в цьому списку означає, що цей блок знаходиться в області, що самозамикається, тому що управління не може з цієї області вийти.
4) Щоб перевірити необхідну кількість виходів для кожного блоку, необхідно під час перебору списку блоків поглянути на клас блоку: блок кінця не може мати вихідних зв'язків, блок умови завжди має дві, інші блоки завжди по одній.

Будь-яке виявлене порушення під час роботи цього алгоритму свідчить про наявність структурних помилок у програмі, написаній користувачем.

2.3 Порядок обходу схеми

Мета програми - перетворити блок-схему на працюючий програмний код для якоїсь придатної для новачків мови програмування, яка підтримує послідовні операції, розгалуження та циклічності. Як мову, для якої буде виконана генерація, була обрана C++, але надалі плануються розширити спектр доступних для генерації коду мов.

Таким чином, алгоритм генерації коду не повинен бути прив'язаний до специфіки якоїсь конкретної мови, а спиратися на базові концепції структурного програмування – послідовність, розгалуження та циклічності.

Умовний оператор та оператор циклу створюють внутрішні області контексту та передають управління першій команді цієї області. Іншими способами передати управління внутрішньої області циклу або області розгалуження не можна. Оскільки стрілки позначають процес передачі управління, на цьому етапі необхідно виявити, чи не вторгаються стрілки у внутрішні області умовних і циклічних операторів. Для цього необхідно перебрати кожну стрілку на схемі та переконатися в тому, що керування буде передано коректно.

У програмному коді області контексту відкриваються і закриваються фігурними дужками "{" і "}" умовними та циклічними операторами (рис 2.9), а у вигляді блок-схем наступним чином (рис 2.10):

1) Блок умови, який є еквівалентом умовного оператора, створює два потоки виконання з власним контекстом і передає керування одному з них залежно від виконання умови, що відповідає відкриванню фігурних дужок у програмному коді та переходу до мітки else (якщо необхідно). Ці області з контекстом закриваються, коли обидва потоки виконання сходяться в одному блоці, що відповідає закриттю фігурної дужки.

```
// Створення зовнішнього контексту
#include <iostream>
int main()
{ // Створення головного контексту
    std::cout << "Enter a number: ";
    int a;
    std::cin >> a;
    if (a > 15)
    {  // Створення нового контексту
        std::cout << "You entered " << a << "\n";
        std::cout << a << " is greater than 15\n";
    }  // Закриття нового контексту
    else
    { // Створення нового контексту
        std::cout << "You entered " << a << "\n";
        std::cout << a << " is not greater than 15\n";
    } // Закриття нового контексту
    return 0;
} // Закриття головного контексту
// Закриття зовнішнього контексту
```

Рисунок 2.9 – Як відбувається відкриття та закриття областей контексту в програмному коді

2) Блок початку циклу, який є еквівалентом циклічного оператора, створює лише один потік виконання зі своїм контекстом, що відповідає фігурній дужці, відкривачій тіло циклу. Внутрішня область контексту циклічного оператора закривається при зустрічі блоку кінця циклу, що відповідає фігурній дужці, що закриває тіло циклу.

Таким чином, створення нової області контексту відбувається, коли:

1) відбувається перехід до першого блоку в одній з гілок умовного оператора;
2) відбувається перехід до першого блоку тіла циклу,

а закриття області контексту, коли:

1) відбувається перехід до блоку, що має кілька вхідних стрілок;
2) відбувається перехід до блоку до блоку кінця циклу.

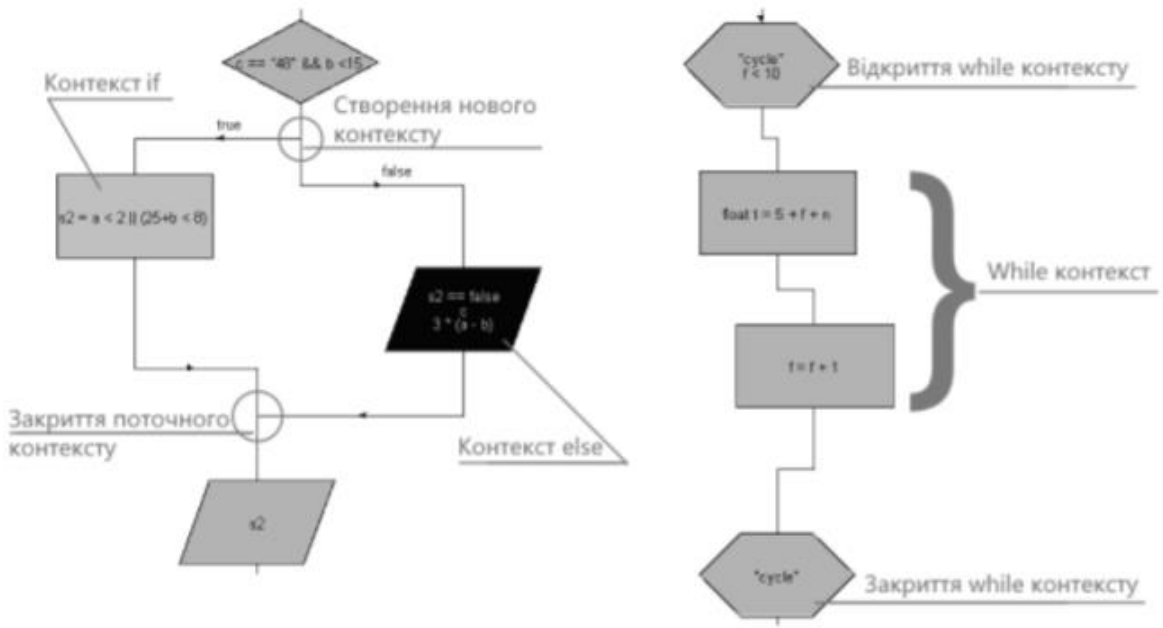


Рисунок 2.10 – Як відбувається відкриття та закриття областей контексту в блок-схемі

Таким чином, структурна помилка, пов'язана з некоректною передачею керування всередину якоїсь контекстної області, буде розглянута як спроба закрити поточну контекстну область. Про процес виявлення такої помилки див. докладніше у розділі програмної реалізації. Окрім цього, згенерований код повинен відповідати наступним вимогам:

1) кожна команда починається з нового рядка;
2) розмір відступу безпосередньо залежить від глибини контексту;
3) команди мають бути вказані в тій послідовності, в якій їх зв'язав користувач за допомогою стрілок;
4) створені області контексту повинні бути закриті фігурними дужками у послідовності, зворотній тій, в якій відкривалися;
5) при зустрічі блоку умов першою має бути оброблена гілка з істинною умовою, тому що в програмному коді гілка з істинною умовою завжди пишеться першою;

6) згенеровані внутрішні функції мають бути оголошені перед тілом основної функції, оскільки імена цих функцій можуть бути нерозпізнані компілятором.

Попередній етап з перевіркою базових вимог до схеми гарантує, що всі блоки, крім блоків-умов і блоку-кінця, мають одну і лише одну вихідну стрілку, таким чином виключаючи невизначеність у тому, якій команді буде передано управління після цієї. Дотримуючись цього правила, ми можемо легко перебрати всі блоки у схемі, в якій немає блоків умов, тому що за однією командою гарантовано слідує одна і тільки одна наступна.

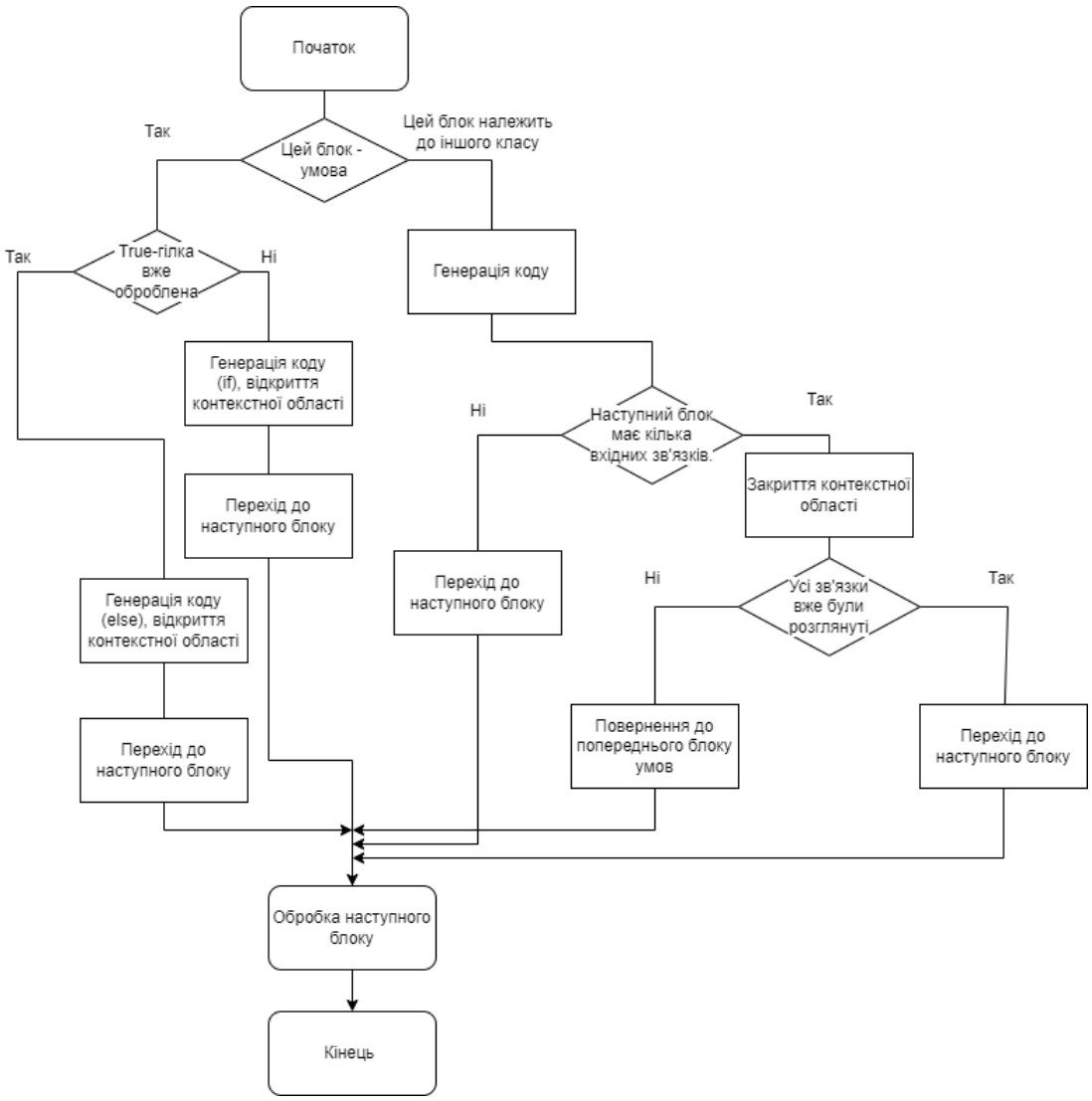


Рисунок 2.11 – Спрощений алгоритм перебору блоків

Щодо блоків-умов, ґрунтуючись на тому, що гілка з істинною умовою в програмному коді пишеться раніше, то гілка з блоками з істинною умовою має пріоритет над гілкою з хибним. Крім цього, після закриття області контексту гілки з істинною умовою необхідно повернутися до блоку умови, а потім згенерувати код для гілки з помилковою умовою. Оскільки закриття області контексту є сходження кількох стрілок в одному блоці, то при зустрічі блоку з кількома вхідними стрілками вважається, що на ньому закінчується внутрішній блок коду. Далі, якщо розглядається true-гілка, то ми закриваємо поточну область контексту та повертаємось до блоку умови, який цю гілку відкрив. Якщо ми розглядаємо false-гілку, то ми закриваємо поточну область контексту і переходимо до блоку з кількома стрілками.

Цей алгоритм зображений на рисунку 2.11.

Розглянемо роботу цього алгоритму на прикладі (рис. 2.12 і 2.13), де числами зліва від блоків та стрілок вказано порядок, з яким перебираються блоки та стрілки:

1) Блоки 1 і 3 генерують код і перевірка переходить до наступного блоку;
2) Блок 5 - блок-умова, тому дивимося, чи ми проходили true-гілку. Інформація про пройдені гілки буде зберігатися як поля у класу DiagramRhoumbus, який безпосередньо є класом для блоків-умов. Так як true-гілка ще не пройдена, генеруємо if-код, створюючи нову контекстну область за допомогою “{“, і переходимо до блоків всередині true-гілки;
3) Блок 7 генерує код та перевірка переходить до наступного блоку;
4) Блок 9 генерує код. Оскільки наступний після нього блок має кілька вхідних стрілок, перевіряємо, чи ми пройшли всі стрілки, що ведуть до нього. Оскільки ми підійшли до цього блоку вперше, то повертаємось до блоку умов;

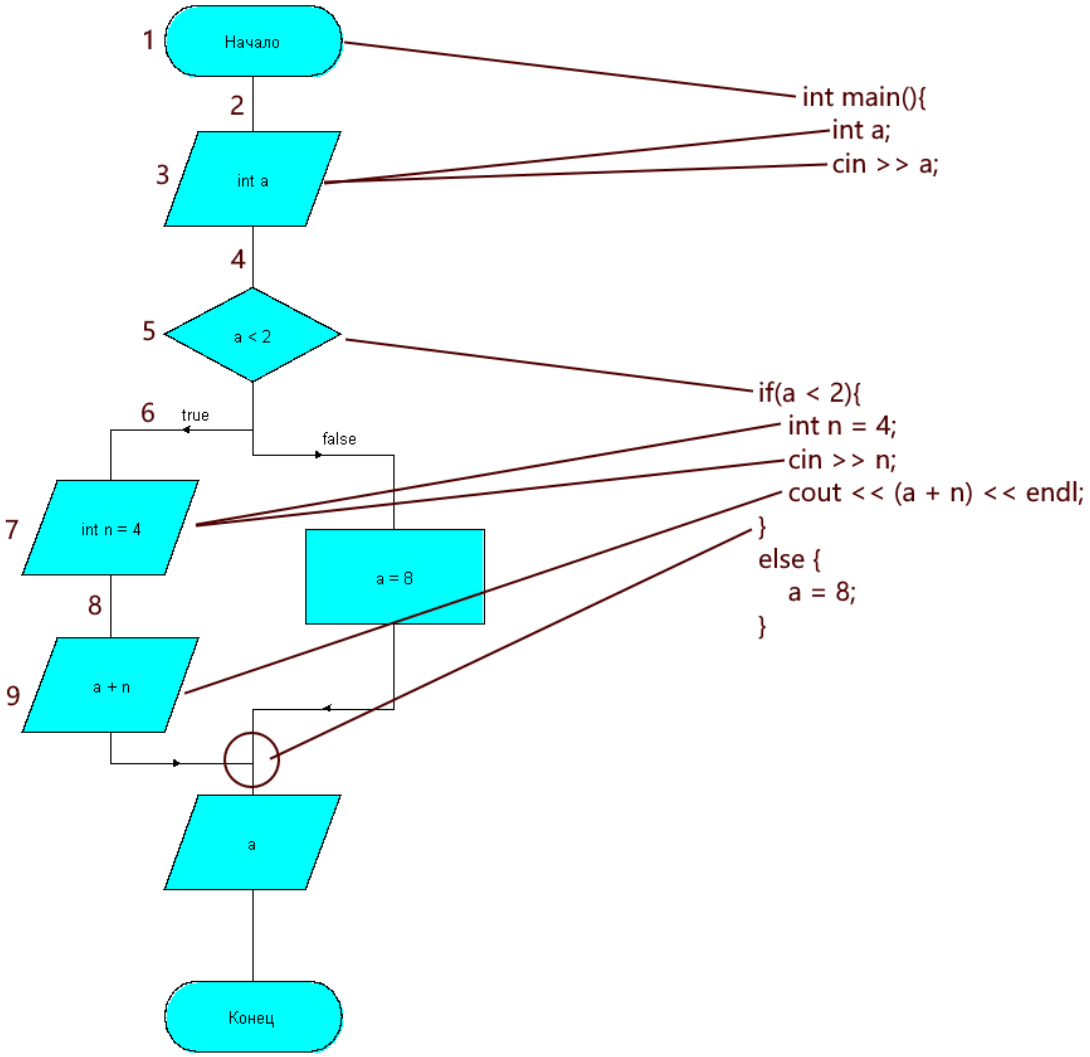


Рисунок 2.12 – Перебір блоків, частина 1

5) Оскільки ми зустріли блок із кількома вхідними стрілками, ми закриваємо контекстну область за допомогою “}”;
6) Повертаємося до блоку умови (10), так як true-гілка пройдена, генеруємо else-код, відкриваємо нову контекстну область за допомогою "{", переходимо до блоку всередині false-гілки;
7) Блок 12 генерує код. Оскільки наступний після нього блок має кілька вхідних стрілок, перевіряємо, чи ми пройшли всі стрілки, що ведуть до нього. Оскільки це так, то закриваємо контекстну область за допомогою “}”, та переходимо до наступного блоку;

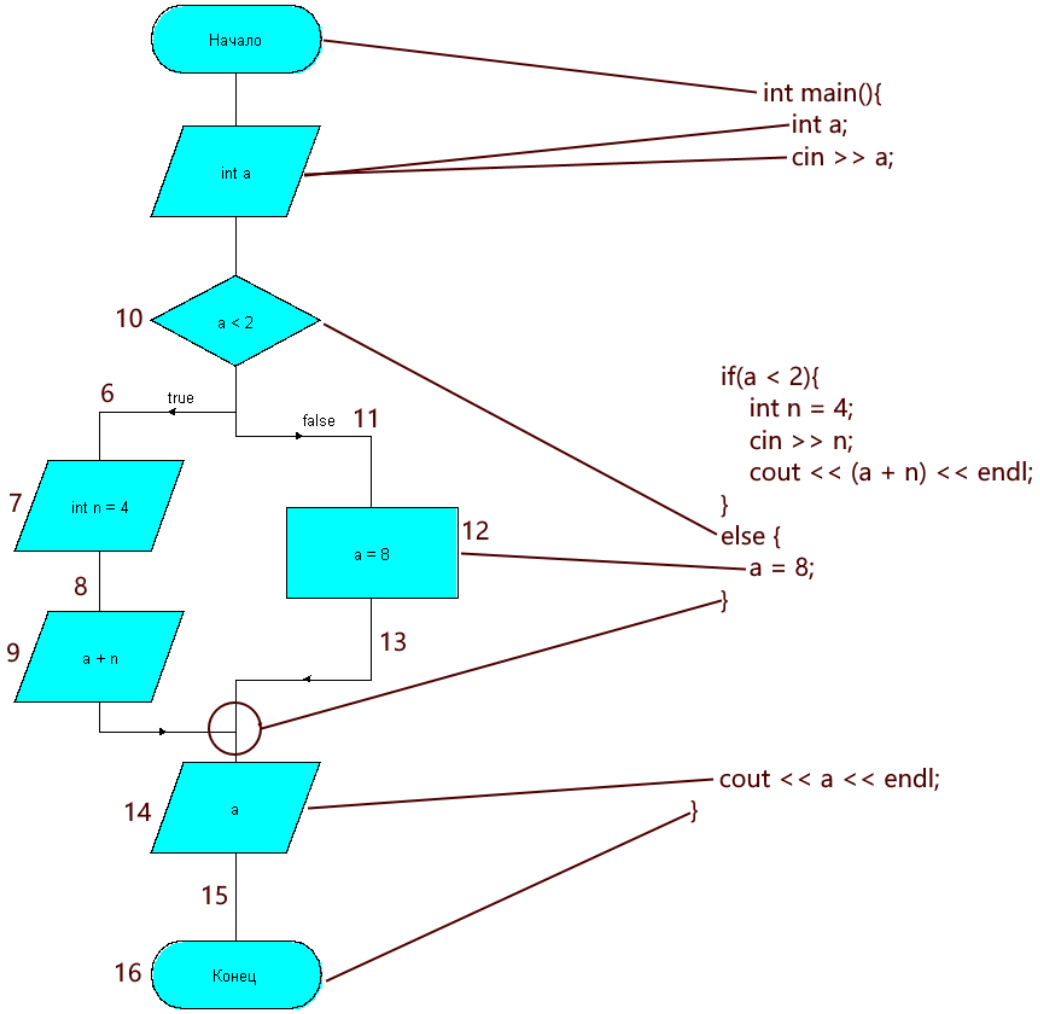


Рисунок 2.13 – Перебір блоків, частина 2

8) Блок 14 генерує код та перевірка переходить до наступного блоку;
9) Блок 16 генерує код та закінчує перевірку блоків;

Втім, алгоритм не розглядає випадок вкладених один в одного умов, тому його необхідно доповнити (рис 2.14).

Якщо ми зустрічаємо блок, у якої вхідних стрілок більше двох, це означає, що тут закривається відразу кілька областей контексту (як аналог з коду, кілька "}" поспіль). Таким чином, перед тим, переходити до цього блоку, необхідно обробити всі шляхи до нього, тобто пройти через всі true і false гілки умов, що зустрілися перед цим блоком. Алгоритм на рис. 2.14

був доповнений обведеними блоками зліва, щоб обробляти випадки закриття кількох контекстних областей одночасно.

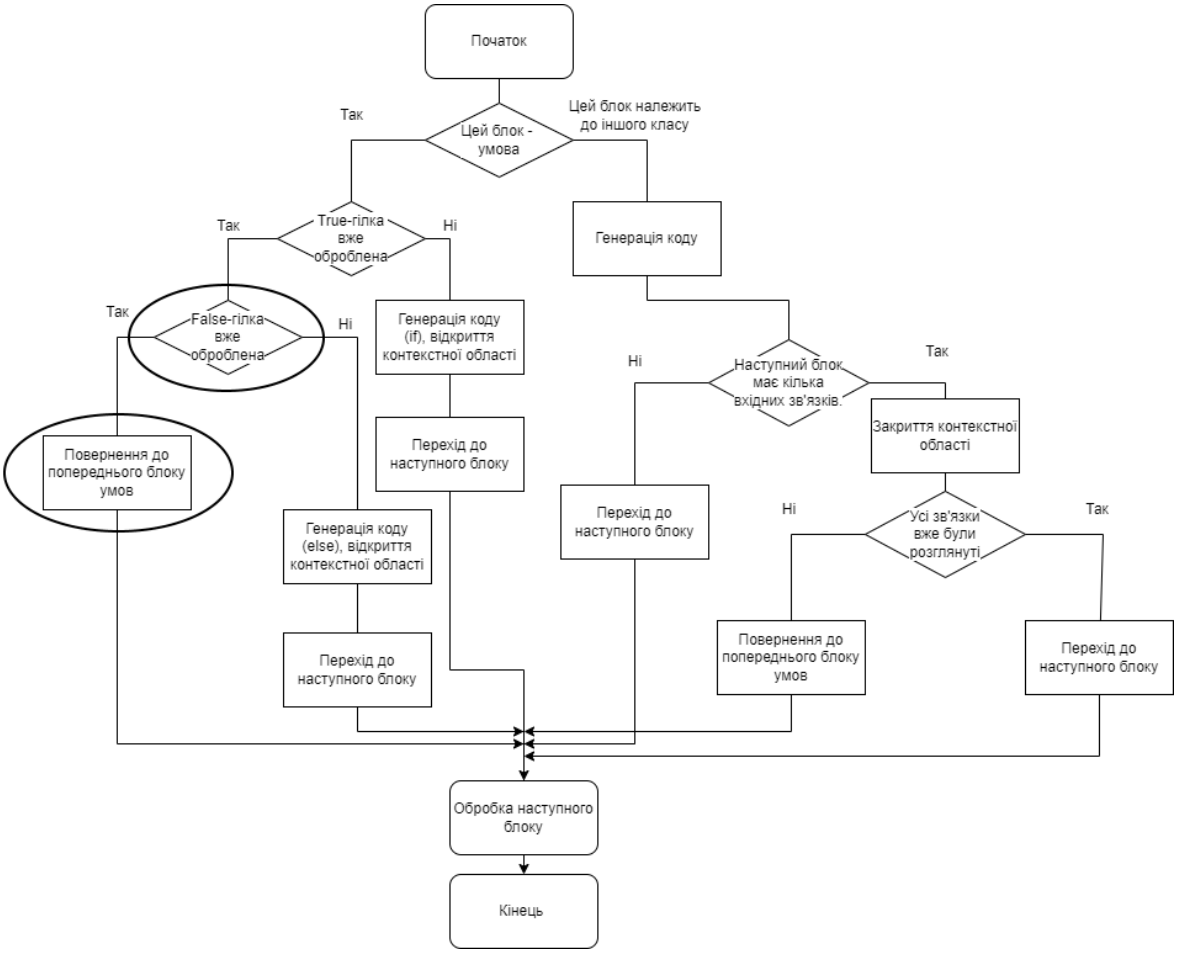


Рисунок 2.14 – Повний алгоритм перебору блоків

Тепер, користуючись цим алгоритмом, можна гарантовано перебрати всі блоки у схемі від блоку початку до блоку кінця, записавши при цьому кожну згенеровану команду в загальний блок коду рівно один раз, враховуючи при цьому розгалуження та цикли. Синтаксичні, семантичні та логічні помилки зустрічаються тільки в коді всередині блоку і не торкаються інших блоків, тому перевірятимуться для кожного блоку індивідуально в ході повного перебору за допомогою створеного алгоритму. Їхня перевірка відбуватиметься в рамках блоку "Генерація коду" (рис 2.14).

3 ТЕХНІЧНЕ ЗАВДАННЯ

3.1 Визначення функціональних вимог

Функціональні вимоги до програмного забезпечення визначають набір можливостей, які програмне забезпечення має надати користувачеві для вирішення його завдань.

- Програмне забезпечення має дозволяти користувачеві візуалізувати написану ним програму у вигляді схеми алгоритмів;
- Програмне забезпечення має виявляти помилки у програмі користувача та перераховувати їх користувачу;
- Програмне забезпечення має генерувати код для існуючої мови програмування на основі положення блоків та коду в них.

Також функціональні вимоги можуть бути записані як варіанти використання (use case). Варіант використання, або прецедент використання (англ. Use case) - це опис поведінки системи, коли вона взаємодіє з будь-ким або будь-чим із зовнішнього середовища. В даному випадку - опис можливої взаємодії між користувачем-програмістом та графічним редактором, який розробляється.

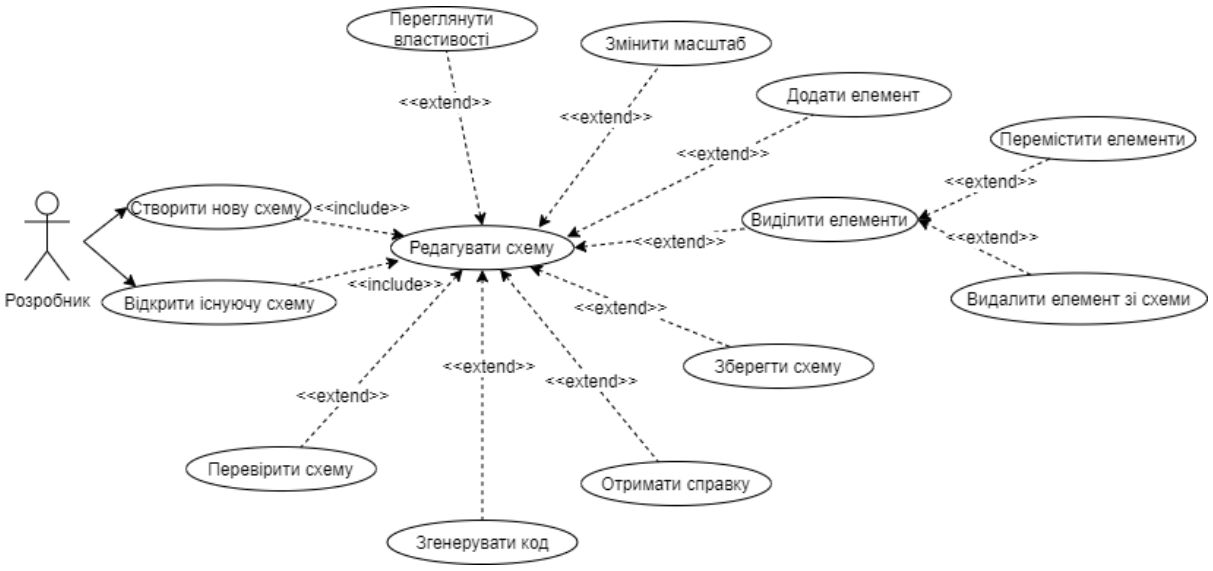


Рисунок 3.1 – *Use case* діаграма *Scheme Prof*

Базовий функціонал у вигляді створення, редагування та збереження схем уже реалізовано у графічному редакторі.

На даний момент редактор схем алгоритмів не перевіряє написаний текст всередині блоків. Користувач записує в них код, знаючи, що код усередині блоків виконується в тій послідовності, в якій пов'язані блоки один з одним за допомогою стрілок.

До базової перевірки схеми входять перевірки таких умов:

- повинна бути рівно одна точка входу та рівно одна точка виходу;
- можна від кожного елемента схеми потрапити до будь-якого іншого, переміщаючись за стрілками від блоку до іншого;
- з блоку умов повинно бути рівно два виходи.

До того ж, написаний код усередині блоків має відповідати типу блоку, а саме:

- у блоці умови має бути bool змінна або вираз, що повертає bool тип;
- у блоці введення даних даних має оголошуватися змінна;
- у блоці операції повинна проводитись якась операція над даними з використанням констант або змінних;
- у блоці виведення даних має бути змінна або вираз;
- блок, що знаменує початок циклу, повинен мати ім'я циклу та умову завершення цього циклу;
- блок, який знаменує кінець циклу, повинен мати тільки ім'я циклу
- текст всередині блок-підпрограми є ім'ям функції.

Помилки, скоєні при створенні схеми алгоритмів та наповненні її кодом, можна класифікувати за кількома категоріями:

- структурні помилки, пов'язані з розташуванням блоків і послідовністю їх виконання;
- синтаксичні помилки, пов'язані з нерозпізнаними символами, лексемами, або відсутністю необхідних;

- семантичні помилки, пов'язані з некоректним використанням констант та змінних всередині операцій;
- логічні помилки, пов'язані з невідповідністю коду та типу блоків (наприклад, операція в блоці умови повертає значення типу, відмінного від bool).

Програмне забезпечення має виявляти всі перелічені помилки та повідомляти про них користувачеві.

Крім перерахованих функціональних вимог, існуючий на даний момент редактор схем алгоритмів необхідно допрацювати наступним чином:

- розширити множину блоків, доступних користувачеві-програмісту, додавши блок-цикл і блок-підпрограму;
- усередині блоку підпрограми має знаходитися власна схема алгоритмів;
- перевірка схеми на правильність має захоплювати схеми у блоках-підпрограмах;
- схеми алгоритмів у блоках-підпрограмах мають бути оформлені у вигляді окремої функції із власним тілом функції при генерації коду;
- блоки, в яких було знайдено помилки, мають бути графічно виділені;
- користувач повинен за бажанням отримати довідку щодо синтаксису псевдокоду та його коректного використання у блоках.

В якості мови програмування, для якого буде генеруватися код, була обрана мова C++, виходячи з особистого досвіду, і так як це одна з перших мов, з якою стикається програміст, не рахуючи мови C. Мова C не підтримує bool і string типи даних, тому було вирішено вибрати C++.

Функція генерації коду викликається після проведення перевірки на всі перелічені типи помилок. Таким чином, максимально знижується для користувача можливість отримання коду, в якому є помилки.

3.2 Нефункціональні вимоги

Програма повинна мати зовнішній вигляд, схожий на інші графічні редактори. Кольорова гама графічних елементів має бути строгою та стриманою, а розташування елементів має бути логічним та передбачуваним для людей, які вже роботали з іншими графічними редакторами. Розробник може бути не ознайомлений із правилами роботи з програмою перед початком роботи.

Сценарії атрибутів якості:

1) Надійність:
    a) Якщо користувач отримує згенерований код, код скомпілюється і запуститься у 94% випадків як мінімум;
    b) Якщо користувач припуститься помилки, пов'язаної з нестачею блоків, він про це дізнається як мінімум у 98% відсотках випадків;
    c) Якщо користувач припуститься синтаксичну помилку в якомусь блоці, блок буде підсвічений як некоректний у 94% випадків як мінімум;
    d) Якщо користувач спробує використати неіснуючу змінну, він про це дізнається у 92% випадків як мінімум
2) Продуктивність:
    a) Програма підтримувати як мінімум 5 рівнів вкладених функцій усередині блоків-підпрограм
    b) Процес пошуку помилок повинен займати не більше 3 секунд
    c) Процес генерації коду має тривати не більше 10 секунд
3) Зручність використання:
    a) Користувачеві вдається візуалізувати програму, яку він хотів, протягом півгодини.

b) Користувач може знайти інформацію в довідці про конкретний блок протягом не більше 2 хвилин

4) Супроводження:

a) Якщо розробник вирішить розширити безліч блоків, доступних користувачеві, користувач отримає доступ до нього протягом не більше тижня

b) Якщо розробник вирішить додати нову мову програмування, для якої буде генеруватися код, користувач отримає доступ до неї протягом двох тижнів.

c) Якщо розробник вирішить додати до списку помилок, що перевіряються, нову, користувач отримає доступ до нового функціоналу протягом не більше трьох днів.

В якості програмного забезпечення, яке буде модифіковано в процесі розробки, було обрано графічний редактор, обмежений на даний момент функціоналом створення/редагування схем алгоритмів, структура та способи взаємодії з яким описана в у наступному розділі.

4 АРХІТЕКТУРА СИСТЕМИ

4.1 Структура графічного редактора

На даний момент у графічному редакторі (написаному на Java з використанням Java Abstract Window Toolkit) реалізовано базовий функціонал у вигляді створення, редагування, збереження та базової перевірки схем. Повна діаграма класів зображена на рисунку 4.1.

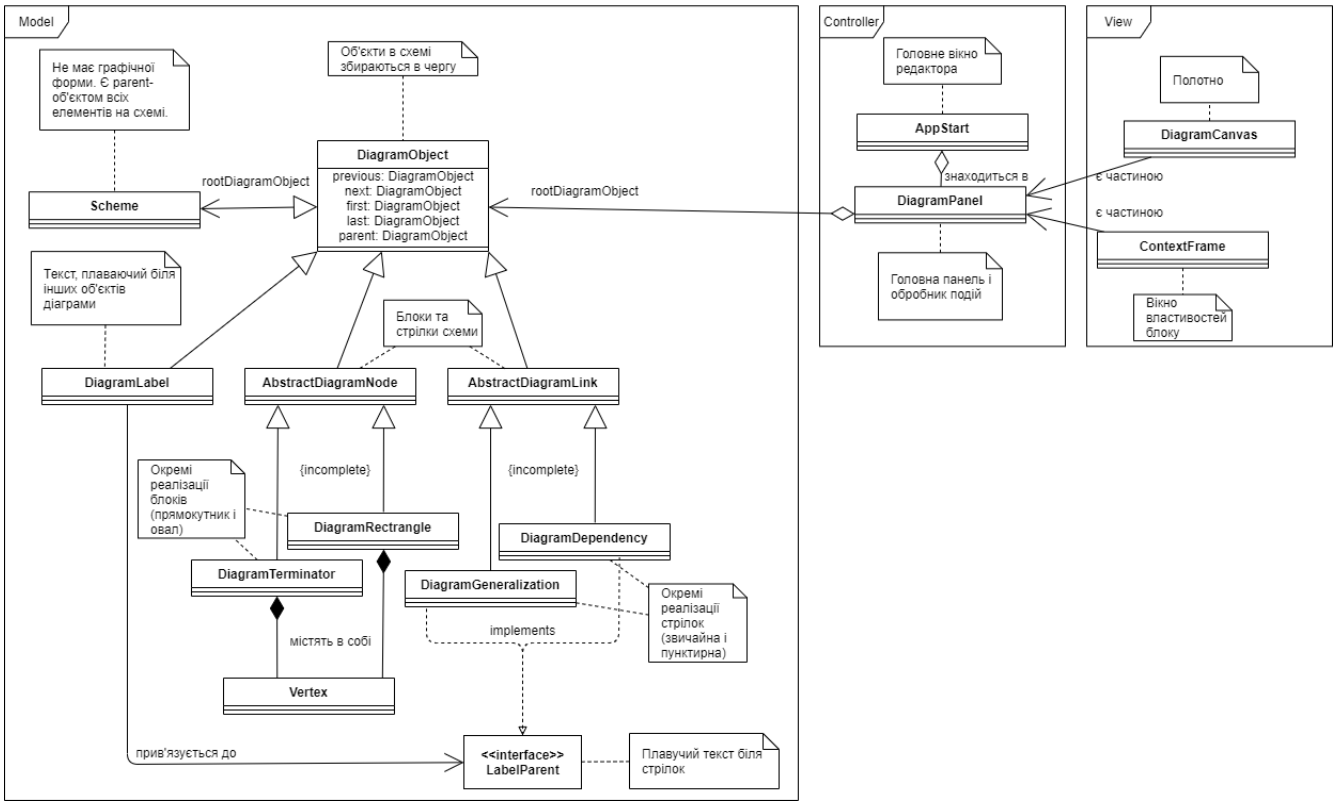


Рисунок 4.1 – Діаграма програмних класів існуючого на даний момент редактора

Для вирішення поставленого завдання потрібні лише деякі з них; також додамо клас SchemeCompiler, який займатиметься перевірками, винесемо ці класи та уточнимо зв'язки між ними для поточного завдання на рисунок 4.2.

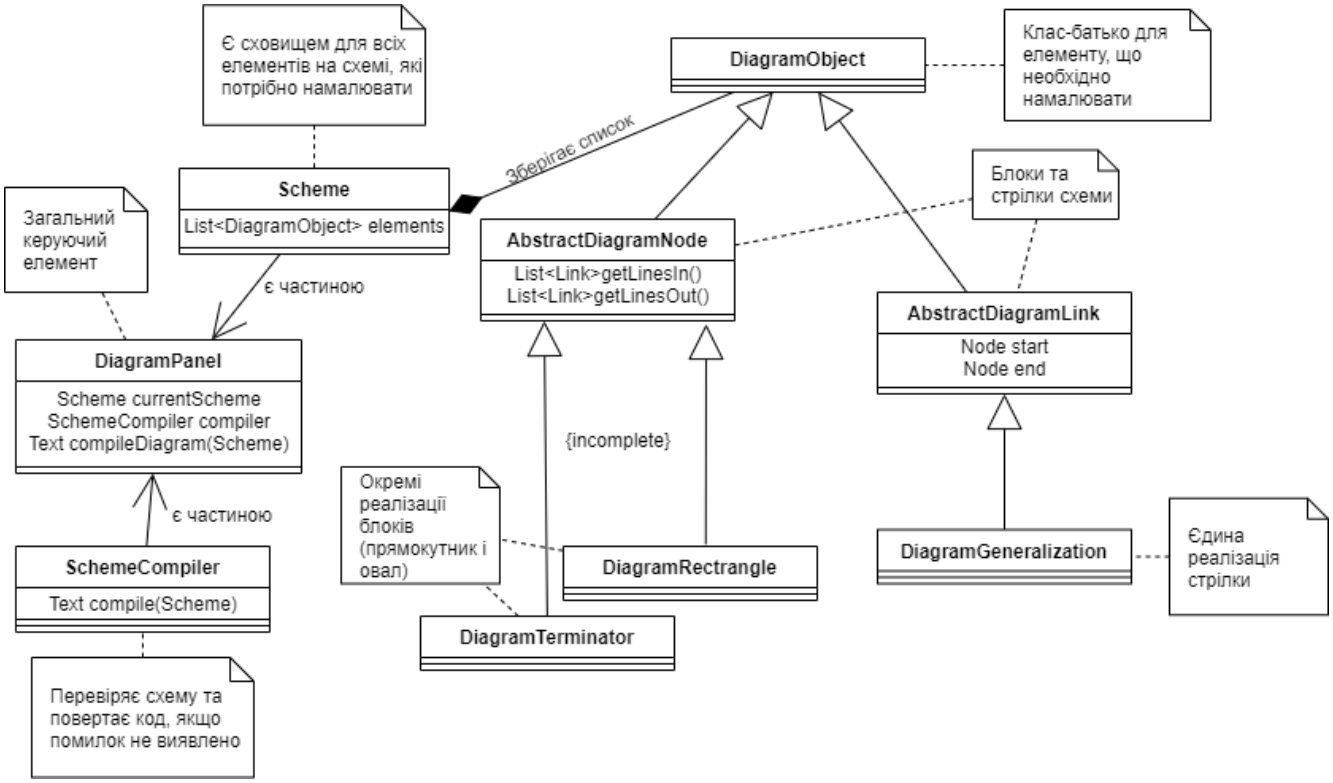


Рисунок 4.2 – Спрощена діаграма програмних класів

Клас DiagramPanel - цей клас реалізує більшу частину інтерфейсу (полотно, вибір блоків, керування схемою) і є обробником подій кліків на полотно. Клас у тому числі реалізує кнопку, при натисканні на яку запускається перевірка та генерація коду. Має такі методи та поля:

- Scheme scheme - містить посилання на схему, елементи якої треба намалювати;
- redraw() - оновлює картинку на полотні, малюючи всі DiagramObject, які знаходяться в екземплярі scheme;
- String compileDiagram(Scheme scheme) - Перевіряє схему на базові структурні помилки, створює екземпляр SchemeCompiler, який проводить синтаксичний, семантичний, логічний аналіз, генерує та повертає програмний код.

4.2 Класи для роботи з даними редактору

Клас DiagramObject – базовий абстрактний клас для будь-якого елемента, який потрібно намалювати на полотні (блок, стрілка, текст над стрілками), містить:

- метод Draw(), який малює елемент на полотні. Метод перевизначається кожним класом, що успадковується від цього класу (форма прямокутника для блоку операцій, форма ромба для блоку умов, лінія зі стрілкою для передачі команди);
- Scheme getParent() - повертає посилання на схему, в якій знаходиться цей елемент.

Клас Scheme - клас, який є сховищем для DiagramObject, яке потрібно намалювати на полотні. Має такі методи та поля:

- HashMap <Int id, DiagramObject obj> diagramObjects - словникова структура, де кожен об'єкт на схемі має свій унікальний id номер;
- DiagramTerminatorStart getStartTerm() - повертає блок початку;
- DiagramTerminatorEnd getEndTerm() - повертає блок кінця; Методи getStartTerm() та getEndTerm() викликаються завжди після перевірки того факту, що в схемі є один і лише один блок початку/кінця, так що методи гарантовано повертають посилання на єдино вірний екземпляр;
- Scheme parentScheme - Якщо ця схема є вкладеною в іншу (за допомогою блоку-процесу), то в цьому полі буде посилання на схему, всередині якої знаходиться ця.

Клас AstractDiagramNode - батьківський клас всіх блоків, має такі методи та поля:

- String caption - текст, який увів у блок користувач;
- List <AbstractDiagramLink> getLinesIn() - список стрілок, що входять до блоку;

- List <AbstractDiagramLink> getLinesOut() - список стрілок, які виходять із блоку;
- String code - згенерований рядок з кодом, який потім буде додано до загального блоку коду;
- generateCode(SchemeCompiler.CodeGenerator codeGenerator) - цей метод додає до загального блоку коду новий, використовуючи рядок code. Метод буде перевизначено у кожному дочірньому класі. Наприклад, якщо code має значення "a < 5", то блок-умова додасть "if(a < 5)", а блок-виведення "cout << (a < 5) << endl;";
- ArrayList <AbstractDiagramNode> getChainedBlocksDown() - реалізує алгоритм ChainDown, перебираючи всі блоки, пов'язані з цим блоком, йдучи від нього;
- ArrayList <AbstractDiagramNode> getChainedBlocksUp() - реалізує алгоритм ChainUp, перебираючи всі блоки, пов'язані з цим блоком, йдучи до нього;
- ArrayList<AbstractDiagramNode> scopeOpeners - цей список порожній, якщо блок має лише одну вхідну стрілку. В іншому випадку, так як у цього блоку кілька вхідних стрілок, це означає, що йому може бути передано управління з декількох місць програми. Враховуючи послідовність програмного коду та відсутність безумовних переходів на кшталт goto, таке можливе лише при закритті кількох областей контексту одночасно (кілька "}" поспіль), коли керування передається із внутрішньої області контексту в одну із зовнішніх. Щоразу, як алгоритм перебору доходить до цього блоку, до цього списку додається останній блок, який відкрив контекстну область. Область контексту вважається коректно закритою, коли її можливі потоки виконання сходяться у одному блоці, як у рисунку 4.3 зліва. На рисунку 4.3 справа сходяться два потоки від різних блоків, що відкрили контекстну область. Щоразу, коли алгоритм, що

перебирає блоки, доходить до цього блоку, сюди буде додано контекстно-утворюючий елемент (блок умови або блок циклу), контекст якого був цим блоком закритий. Якщо такий елемент вже вказано, замість додавання нової копії видаляється стара. Таким чином, якщо всі контекстні області були коректно закриті, список буде порожній. Якщо цьому блоку було некоректно передано керування з іншої області, то у списку будуть вказані контекстно-утворюючі елементи, області контексту яких не було закрито коректно в цьому блоці. Таким чином, зберігаючи в блоці інформацію про те, як алгоритм до цього блоку дістався, можна виявити останній тип структурних помилок, пов'язаних з некоректною передачею управління чужу контекстну область.

- void compilingReset() - очищає всі тимчасові дані, пов'язані з генерацією коду та перевіркою (рядок code, лист scopeOpeners, і флаг passed у пов'язаних з блоком стрілок).

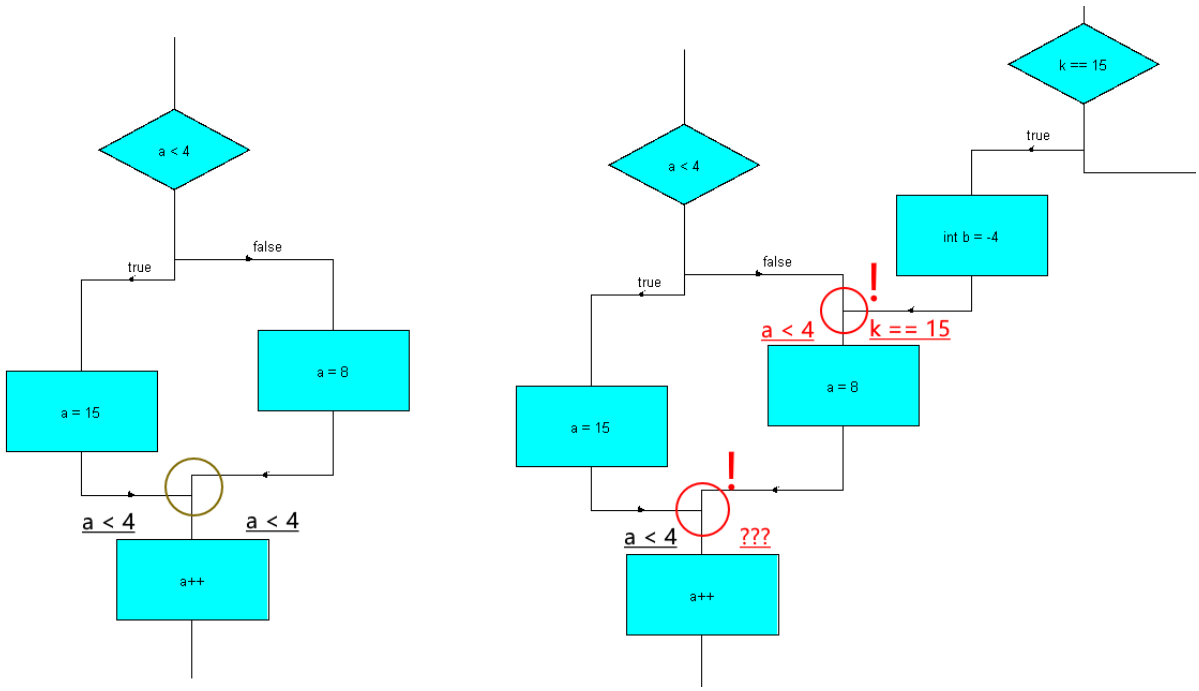


Рисунок 4.3 – Коректне і некоректне закриття контекстних областей

Клас AstractDiagramLink - батьківський клас для всіх класів-стрілок (для розглянутої задачі достатньо одного типу стрілок, тому реалізований всього один клас, відповідальний за стрілки). Має такі поля:

- AbstractDiagramNode start - блок, з якого виходить стрілка.
- AbstractDiagramNode end - блок, до якого входить стрілка.
- String caption - якщо вихідний блок - блок умови, приймає значення " true " чи " false " . Якщо вихідний блок будь-який інший, рядок порожній.
- bool passed - boolean флаг, який зберігає дані, була ця стрілка переглянута перебираючим алгоритмом, чи ні.

Класи DiagramRectangle, DiagramParallelogram - є дочірніми класами AbstractDiagramNode і мають ті ж методи та поля, що й батьківський клас, перевизначаючи деякі методи на кшталт малювання під специфіку описуваної фігури (прямокутник і паралелограм).

Клас DiagramTermitanorStart, DiagramTermitanorEnd - є дочірніми класами AbstractDiagramNode і мають ті ж методи та поля, що й батьківський клас, перевизначаючи деякі методи на кшталт малювання під специфіку описуваної фігури (овал).

Клас DiagramRhombus - Дочірній клас AbstractDiagramNode має такі методи, специфічні для блоку-умови:

- getTrueBranch() - повертає вихідну від цього блоку стрілку, яка веде до початку true-гілки;
- getFalseBranch() - повертає вихідну від цього блоку стрілку, яка веде до початку false-гілки.

Ці методи викликаються після перевірки, що від блоку виходять дві і лише дві стрілки, гарантовано повертаючи однозначний результат.

Клас DiagramDiamond - дочірній клас AbstractDiagramNode, має такі поля та методи, специфічні для блоку-циклу:

- String cycleName - ім'я циклу

- boolean isOpening - цей блок відкриває або закриває цикл
- ArrayList<AbstractDiagramNode> getBlocksInLoop() - повертає список блоків між відкриваючим і закриваючим блоком, використовуючи модифіковані алгоритми chainDown і chainUp, додаючи до списку всі блоки, від яких можна дістатися від блоку початку/кінця, не зустрівши блок кінця/початку. Якщо цей список потрапляє блок початку схеми, то отже було всередину циклу було передано управління, не пройшовши через блок початку циклу. Якщо в цей список потрапляє блок кінця схеми, то управління було некоректно передано з середини циклу в довільну точку програми, не пройшовши через блок кінця циклу. Хоча і обидві ці помилки можна виявити за допомогою поля scopeOpeners (так як вони є окремим випадком передачею управління в чужу контекстну область), цей метод дозволяє конкретизувати, що помилка пов'язана з довільним входом/виходом з циклу.

Клас DiagramPreprocess - дочірній клас AbstractDiagramNode, має поле, специфічне для наперед визначеного процесу - посилання на схему, яка знаходиться всередині нього.

Клас DiagramGeneralization - дочірній клас AbstractDiagramLink, має ті ж поля та методи, що і батьківський клас, перевизначаючи деякі з них (наприклад, малювання стрілки так, щоб вона згиналася під прямим кутом).

Для перевірки схеми на помилки та генерації коду створимо **клас SchemeCompiler**, екземпляр якого буде створюватися і використовуватися в головному управляючому класі DiagramPanel.

5 ПРОГРАМНА РЕАЛІЗАЦІЯ

5.1 Клас SchemeCompiler

Програмна реалізація полягає в доповненні графічного редактора, тому всі розроблені класи написані Java мовою, як сам і редактор. Клас **SchemeCompiler** перебиратиме блоки, шукатиме в коді блоків помилки, і генеруватиме код, його структура зображена рисунку 5.1.

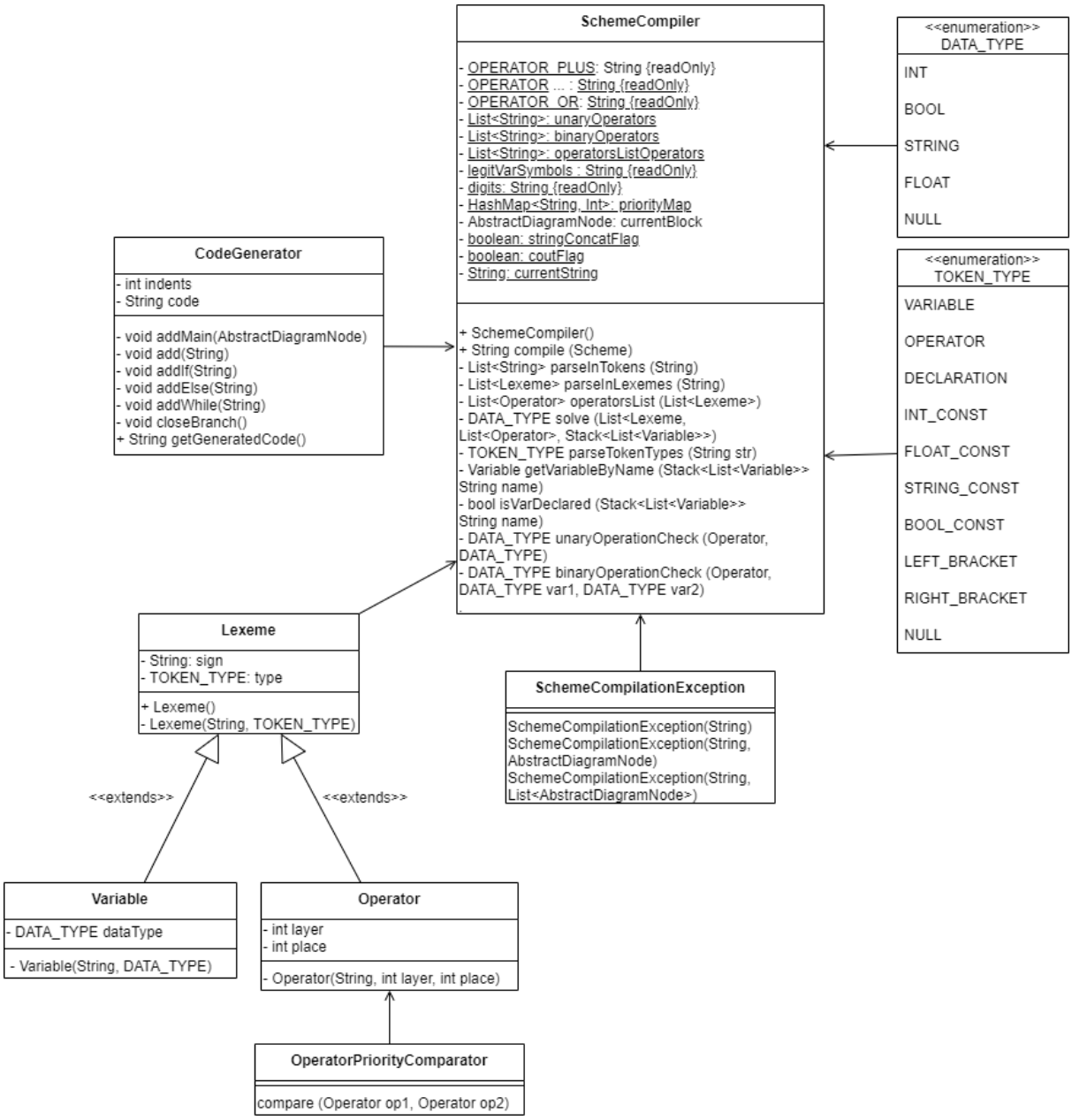


Рисунок 5.1 - Структура класу SchemeComplier

Усі поля та методи цього класу приватні, крім конструктора класу та методу String compile (Scheme), який приймає як параметр існуючу схему, а повертає скомпільований код (якщо всі перевірки пройдені успішно). Якщо під час перевірок виявляється помилка, кидається SchemeCompilationException - дочірній клас від Runtime Expection, що має у собі 3 конструктора:

1) SchemeCompilationException(String message) - помилка відноситься до всієї схеми в цілому (відсутня блок початку/кінця).
2) SchemeCompilationException(String message, AbstractDiagramNode block) - помилка відноситься до якогось конкретного блоку (у блоці знайдено синтаксичну помилку).
3) SchemeCompilationException(String message, List<AbstractDiagram Node> blocks) - помилка відноситься до групи блоків (до якихось блоків не можна дістатися з блоку початку схеми).

Усі блоки, в яких є помилки, забарвлюються цим класом у червоний колір.

Користувачеві доступні чотири типи даних: цілісний int, число з плаваючою точкою float, логічний тип boolean, і рядок string. Дані, доступні для операцій, винесені в структуру enum. Тип NULL зарезервований для випадків, коли компілятор не може визначити тип змінної або результат операції.

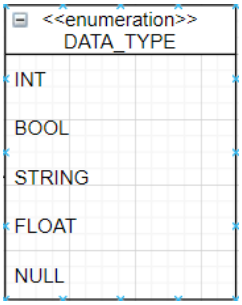


Рисунок 5.2 - Типи даних, що доступні для операцій

Список можливих для використання операторів був створений на основі списку існуючих операторів C++, їхні пріоритети були розставлені відповідно до правил мови (табл. 5.1).

Таблиця 5.1 - Оператори

Пріор.	Познач.	Опис	Допустимі дані
1	++	Інкремент, збільшує число на 1	int, float
2	--	Декремент, зменшує число на 1	int, float
3	!	Унарне "Ні"	bool
4	*	Множення	int, float
5	/	поділення	int, float
6	+	складання	int, float, string, bool (якщо зі string)
7	-	віднімання	int, float
8	<	менше	int, float
9	<=	менше або дорівнює	int, float
10	>	більше	int, float
11	>=	більше або дорівнює	int, float
12	==	дорівнює	int, float, string, bool
13	!=	не дорівнює	int, float, string, bool
14	&&	логічне "і"	bool
15	||	логічне "або"	bool
16	=	присвоювання	якщо одного типу: int, float, string, bool

Кожен оператор записаний у форматі рядка, який зберігає його символ:

```
final static String OPERATOR_LESS_OR_EQUALS = "<=";
final static String OPERATOR_NOT = "!";
```

Оператори розподілені за двома категоріями: унарні та бінарні; кожна категорія має свій список List<String> - unaryOperators та binaryOperators. Список List<String> operatorsList використовується для того, щоб визначити, відноситься набір символів до операторів, чи ні.

Рядок legitVarSymbols зберігає набір символів, легальних для ідентифікаторів (від латинської a до Z, а також символ "_"). Цифри винесені в окремий рядок digits. Ідентифікатори можуть мати цифри, але не можуть з них починатися.

Словник HashMap<String, Integer> priorityMap містить інформацію про пріоритети, де кожному оператору String відповідає унікальний пріоритет Integer. Чим більше число, тим вищий у оператора пріоритет.

AbstractDiagramNode currentBlock - зберігає посилання на блок, який зараз розглядається алгоритмом.

Флаг stringConcatFlag та флаг coutFlag використовуються під час генерації коду для блоків-операцій та блоків введення-виведення.

Рядок currentString - рядок, який на даний момент аналізується алгоритмом.

Конструктор SchemeCompiler викликається у класі DiagramObject, коли користувач вирішує перевірити схему на помилки. Метод ініціалізує списки операторів (unaryOperators, binaryOperators, operatorsList) і повертає посилання на створений конструктором екземпляр цього класу.

Токен - набір символів, які мають сенс для компілятора. Існує три види токенів: оператори, змінні та константи. Змінні мають вигляд набору символів, легальних використання у ідентифікаторах (від a до Z, включаючи "_" і цифри, але не починаючи ідентифікатор з них). Константи мають вигляд даних у сирому вигляді, тобто "25" для числа, "true" для boolean, і будь-який набір символів у лапках для рядків. Крім звичайних

операторів, також є спеціальні – права та ліва дужка. NULL зарезервований для випадків, коли компілятор не може визначити тип якогось токена. Можливі типи токенів винесені в структуру enum (рис 5.2).

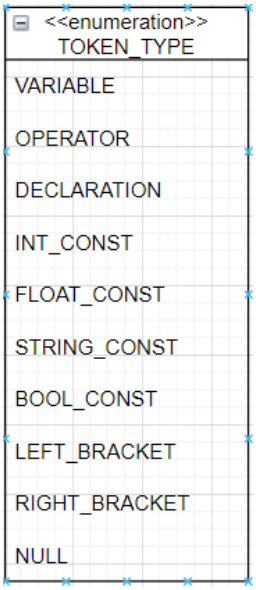


Рисунок 5.2 - Типи даних, що доступні для операцій

Інші методи і поля будуть розглянуті у наступних підрозділах цього розділу.

5.2 Синтаксичний аналіз

Метод ArrayList<String> parseInTokens (String str) розбирає переданий йому рядок на токени. На даному етапі токенам ще не присвоюється лексичний зміст (наприклад, що “int” - це тип даних), у цьому методі тільки складається список з токенів, які розділені між собою пробілами або операторами, при цьому виявляючи токени, які є синтаксично неправильними і не можуть бути розпізнані компілятором. Наприклад, такі конструкції, як "25k", "25.5.5", "+*" не мають сенсу, тому що їх не можна віднести до жодного з перерахованих типів токенів. Такі ключові слова як "int" або "true" на даному етапі будуть розпізнані як імена змінних, а

установка якому типу лексем відноситься конкретний токен і привласнення йому лексичного сенсу буде проведено пізніше.

Розглянемо розбір на токени рядку “string s = "qq" + 15.1” (рис 5.3). Символи перебираються один за одним, доповнюючи поточний токен і, коли треба, змінюючи його тип. Змінна типу TOKEN_TYPE type зберігає тип токена на даний момент.

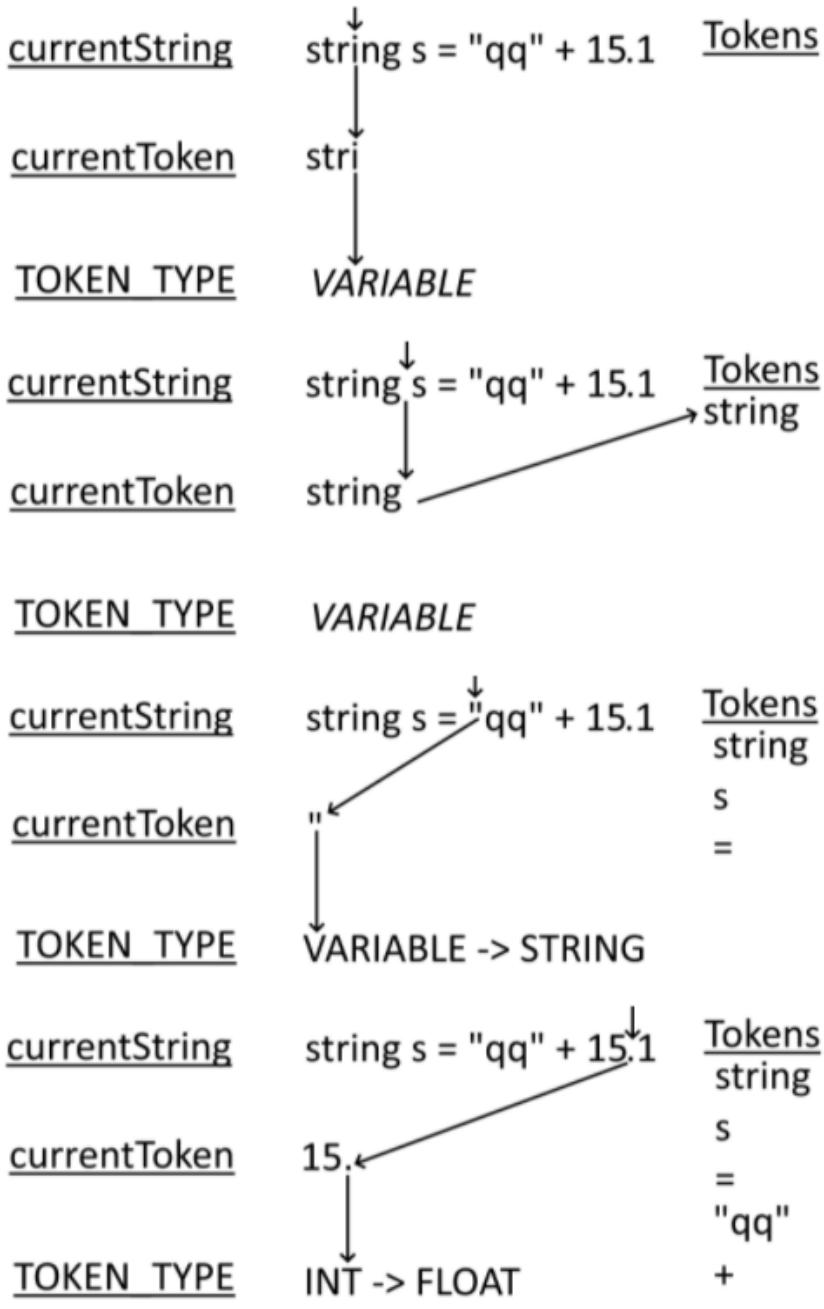


Рисунок 5.3 - процес парсингу на токени

Існує кілька правил:

1) Два токена зі змінними/константами не можуть йти поспіль, не маючи між собою оператора;
2) Якщо токен порожній, то лапки перетворюють тип токена на рядок-константу (STRING_CONST). Рядок-константа може містити будь-який символ, тому будь-який відмінний від лапок символ просто додається до токену. Рядок-константа закривається лапками.

   Спроба відкрити лапки, коли поточний токен не порожній (наприклад, *74*), призведе до синтаксичної помилки, оскільки токен *74"* не є жодним типом з токенів;
3) Оператори можуть складатися з одного (+, !, <) або двох (++, <=, ||) символів. Будь-який зустрінутий символ, що не належить до множини латинських літер або цифр, буде розпізнаний як операторський символ. Якщо токен порожній або вже містить операторський символ, то він приписується до токену і встановлює значення типу значення токена-оператора (OPERATOR).

   Якщо токен не порожній, і при цьому не є операторським, то поточний токен завершується, додається до списку токенів, на його місці утворюється новий, в який записаний операторський символ. Щоразу, коли щось завершує токен з оператором, відбувається перевірка токена: якщо такий оператор існує (+, ||, чи <=), він додається до списку токенів. Якщо такого оператора немає (|&, +-, #, */), то виводиться повідомлення про синтаксичну помилку;
4) Цифри можуть входити до складу числових констант або ідентифікаторів змінних, однак ідентифікатори змінних не можуть з цифр починатися.

   Якщо токен порожній, цифра записується, встановлюючи тип INT_CONST. Якщо токен не порожній, цифра може стати лише продовженням якогось числа (FLOAT_CONST, INT_CONST),

ідентифікатора (VARIABLE), або рядку-константи (STRING). В іншому випадку, буде розпізнана синтаксична помилка;

5) Латинський символ може входити до складу ідентифікаторів та рядків-констант. Спроба дописати його до числа-константи (INT_CONST, FLOAT_CONST) буде розпізнана як синтаксична помилка, тому що ідентифікатор не може починатися з цифри. Латинський символ може бути записаний у порожній токен, тоді тип токену стане VARIABLE;
6) Точка може бути дописана лише до INT_CONST або STING_CONST. Спробу дописати її будь-якому іншому типу або у пустий токен буде розпізнано як синтаксичну помилку.
7) Дужки завершують токен та додаються до списку токенів, їх аналіз буде проведено пізніше.

Дотримуючись цих правил, рядок *string s = "qq" + 15.1* перетворюється на список з шести токенів: *string, s, =, “qq”, +, 15.1*.

Другий етап синтаксичного аналізу - визначення типу кожного токена, при цьому на основі кожного токена створюється екземпляр класу Lexeme.

Клас Lexeme – клас, який програмно реалізує лексеми, тобто токени, у яких виявилося можливим визначити тип – змінна, константа, оператор або тип даних.

Клас має два поля:

- String sign - строка, в якій зберігається набір символів, що характеризує це лексему ("int", "true", "15", "++")
- TOKEN_TYPE type - один з типу токенів (VARIABLE, OPERATOR, DECLARATION, INT_CONST, FLOAT_CONST, BOOL_CONST, STRING_CONST, RIGHT_BRACKET, LEFT_BRACKET).

Також має стандартний конструктор із порожнім тілом та конструктор з параметрами (String tSign, TOKEN_TYPE tType), який встановлює передані дані у поля створеного екземпляра.

Клас Variable успадковує клас Lexeme, і на додаток до спадкових полів має поле DATA_TYPE dataType, який містить тип даних для змінної: INT, BOOL, STRING, FLOAT.

Клас має конструктор, що приймає як параметр рядок як ім'я змінної та тип даних, який ця змінна зберігає. Поле TOKEN_TYPE для екземплярів цього класу завжди дорівнює VARIABLE.

Класс Operator успадковує клас Lexeme, і на додаток до спадкових полів має поле int layer – шар пріоритетів. Оператор множення має пріоритет вище оператора додавання, але оператор додавання повинен виконуватися раніше, якщо він знаходиться в дужках. Поле int layer показує, у якій кількості дужок знаходиться оператор. Відповідно, якщо оператор знаходиться за дужками, то layer дорівнюватиме 0.

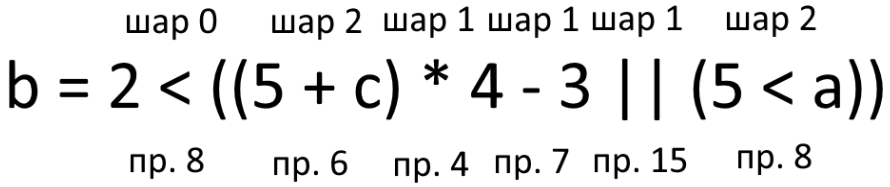


Рисунок 5.4 - Розставлення операторам шару пріоритету (зверху) та порядку пріоритету з документації C++ (знизу)

Клас має конструктор, що приймає як параметр рядок як позначення оператора та число з його шаром пріоритету. Поле TOKEN_TYPE для екземплярів цього класу завжди дорівнює OPERATOR.

Необхідно впорядкувати оператори так, щоб спочатку вони порівнювалися за рівнем дужок, а потім за номером пріоритету, який встановлено мовою C++. Для цього було створено клас

OperatorPriorityComparator, який успадковує базовий Java клас Comparator, реалізуючи його метод compare:

```
@Override
public int compare(Operator o1, Operator o2) {
    int n = o2.layer - o1.layer;
    if(n == 0) return priority Map.get(o1.sign) - priorityMap.get(o2.sign);
        else return n;
    }
```

Повертаємося до **класу DiagramCompiler**:

Метод TOKEN_TYPE parseTokenType(String str) використовується для визначення типу токена та створення на його основі лексеми. Метод намагається по черзі розпарсити рядок на різні типи даних:

1) Якщо парсинг цілого числа за допомогою стандартного Java-метода Integer.parseInt(str) вдався, то повертається INT_CONST;
2) Якщо парсинг дробового числа за допомогою стандартного Java-метода Float.parseFloat(str) вдався, то повертається FLOAT_CONST;
3) Якщо рядок дорівнює "true" або "false", то повертається BOOL_CONST;
4) Якщо рядок дорівнює одному з позначень типів даних (“int”, “bool”, “string”, “float”), то повертається DECLARATION;
5) Якщо рядок оточений лапками з обох боків, то повертається STRING_CONST;
6) Якщо рядок дорівнює "(", то повертається LEFT_BRACKET;
7) Якщо рядок дорівнює ")", то повертається RIGHT_BRACKET;
8) Якщо рядок еквівалентний якомусь із операторів в operatorsList, то повертається OPERATOR;
9) Якщо жодного з типів не було розпізнано, повертається тип VARIABLE.

Метод lexemesParse (String str) виконує синтаксичний аналіз та працює наступним чином:

1) розбиває рядок на токени за допомогою parseInTokens(str);
2) перебирає кожен із токенів, створюючи для кожного з них екземпляр класу Lexeme. Якщо токен - оператор, створює екземпляр Operator, тимчасово встановлюючи його шар в 0. Якщо токен - змінна, створює екземпляр Variable, тимчасово встановлюючи тип даних, що зберігаються в NULL до семантичного аналізу.

Метод ArrayList<Operator> operatorsList(ArrayList<Lexeme> lexes) витягує з існуючого списку лексем оператори, розставляємо їм шари, номер якого дорівнює сумарній кількості дужок, якими цей оператор оточений. Потім він сортує оператори за допомогою OperatorPriorityComparator та повертає відсортований список.

5.3 Семантичний аналіз

Мета семантичного аналізу полягає в наступному:

1) Розпізнати оголошення змінної з наданням якогось типу даних для її подальшого використання.
2) Розпізнавати помилки, коли змінну перед використанням не було оголошено; коли була оголошена в контексті, звідки не може бути доступна для цього рядка коду; або коли тип даних, що зберігається, в змінній не підходить для використання з якимось оператором.
3) Розпізнавати помилки, коли оператори використовуються з типом даних, які для цього оператора непризначені (спроба поділити рядок, або скласти два boolean).

Для першого та частково другого пункту використовуються раніше введені концепції відкривання та закривання контекстних областей, а також структура даних Stack<List<Variable>> varStack.

Щоразу, коли зустрічається блок умови чи блок циклу, відкривається нова контекстна область, а стек додається новий List<Variable>. Всі оголошені в цій області змінні будуть додані в останній елемент стека. Щоразу, коли зустрічається блок із кількома вхідними стрілками, закривається остання відкрита контекстна область, і з стека видаляється останній елемент. Таким чином, оголошені змінні всередині області не буде видно зовні.

Для пошуку змінної в стеку використовуються два методи:

1) getVariableByName(Stack<List<Variable>> varStack, String variable) переглядає стек і повертає посилання на змінну. Цей метод використовується в конвертації списку лексем в convertedData (див. нижче), щоб визначити який тип зберігає змінна;

2) isVarDeclared(Stack<List<Variable>> varStack, String variable) переглядає стек і повертає true, якщо змінна з таким іменем існує. Цей метод використовується під час попередніх перевірок, щоб виявляти випадки повторного оголошення змінної з тим самим ім'ям або спроби використовувати змінну, якої немає в стеку.

Щодо перевірок використання операторів із коректними типами даних, ситуація спрощується тим, що для цієї перевірки немає необхідності обчислювати значення якогось виразу. Наприклад, для вираження “7 + 5” немає необхідності складати 7 і 5 і необхідно всього лише переконатися, що тип токена 7 (INT) і тип токена 5 (INT) можуть бути використані з оператором “+”. Складність ситуації полягає в тому, що для вираження “int a = 7 + 5” необхідно спочатку перевірити легальність операції "7 + 5", а потім перевірити легальність операції “a = результат (7 + 5)”.

Метод DATA_TYPE solve(ArrayList<Lexeme> lexemes, ArrayList<Operator> operators, Stack<List<Variable>> vars) на основі списку лексем, пріоритетності операторів, та списку доступних у цій контекстній області змінних обчислює значення даного йому виразу.

Наприклад, підготовка даних рядку *string s = "qq" + 15.1* до цього методу виглядає так з виглядає наступним чином:

1) Розбиваємо рядок на токени: *string, s, =, “qq”, +, 15.1;*
2) Створюємо для кожного токена лексему (таблиця 3.2);
3) Вилучаємо оператори та сортуємо їх: “+”, потім “=”;
4) Змінна Stack<List<Variable>> vars є локальною змінною для функції перевірки та змінюється з кожним перевіреним блоком. Якщо цей блок перший, то в стеку немає жодної змінної.

Таблиця 5.2 - Список екземплярів класу Lexeme для *string s = "qq" + 15.1*

Поле String sign	Поле TOKEN_TYPE	Поле layer (Operator)/ DATA_TYPE (Variable)
string	DECLARATION	
s	VARIABLE	STRING
=	OPERATOR	0
“qq”	STRING_CONST	
+	OPERATOR	0
15.1	FLOAT_CONST	

Цей метод складається з двох частин: підготовка до обчислення виразу, і безпосереднє обчислення виразу. Оскільки нам необхідно зберігати тип даних результату для кожної операції, оскільки цей тип даних буде пізніше використовуватися в інших операціях, було прийнято рішення створити структуру даних ArrayList<Object> convertedData, який зберігатиме тип даних для операцій (INT, FLOAT, STRING, BOOL) та необхідні оператори.

Наприклад, вираз *string s = "qq" + 15.1*, коли s зберігає в собі тип STRING, буде в convertedData виглядати так: *STRING = STRING + FLOAT*.

Алгоритм з перетворення набору лексем в convertedData аналізує кожну лексему у списку переданих та залежно від TOKEN_TYPE цієї лексеми робить наступне:

VARIABLE:

Вираз у блоці може містити одну лише змінну у двох випадках - якщо це зміна BOOL типу, яка знаходиться в блоці умови, або якщо ця змінна будь-якого типу, яка знаходиться в блоці виведення (рис. 5.5).

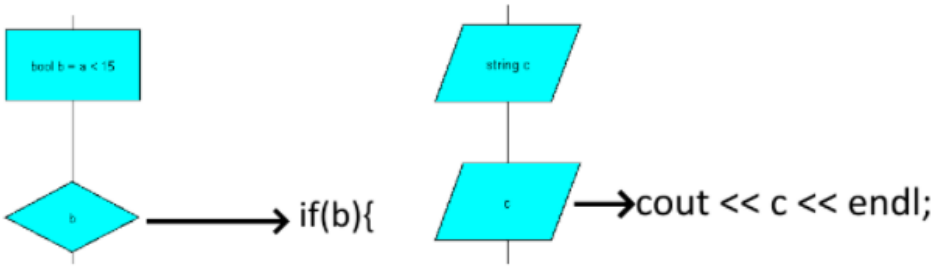


Рисунок 5.5 - Випадки використання в блоці виразу з однією змінною

Якщо в рядку всього дві лексеми: тип даних та ідентифікатор, то це розпізнається як оголошення змінної з введенням користувача. Оскільки в цьому виразі немає операторів, код буде згенерований без обчислень, і алгоритм перейде до наступної команди (рис. 5.6).

У такому разі генерується код, специфічний для блоку, алгоритм переходить до наступного блоку. Докладніше про процес генерації див. у наступному підрозділі.

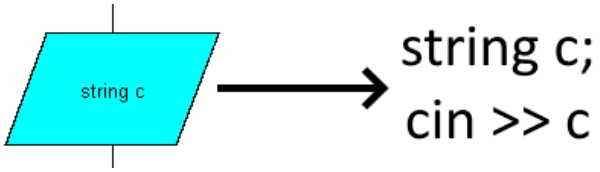


Рисунок 5.6 - Оголошення змінної з введенням користувача

У всіх інших випадках змінна додається в конвертовані дані у вигляді типу, який вона зберігає.

У всіх трьох випадках використовується метод boolean isVarDeclared(Stack<variable>>varStack, String variable):

1) Якщо змінна в блоці всього одна, то перевіряється, чи була вона оголошена (семантична помилка, якщо ні);

2) Якщо відбувається оголошення змінної, то перевіряється, чи існує вже змінна з таким ім'ям (семантична помилка);

3) Якщо змінна використовується у виразі з оператором, то спочатку перевіряється, чи була ця змінна оголошена (семантична помилка, якщо ні), а потім застосовується метод getVariableByName(Stack<List<Variable>> varStack, String name), який поміщає в convertedData тип даних, який вона зберігає.

STRING_CONST, INT_CONST, FLOAT_CONST, та BOOL_CONST:

Константи конвертуються у відповідний їм тип даних і додається до convertedData. Крім того, оскільки змінні та константи не можуть знаходитися поряд без оператора між ними, то поява двох констант/змінних поспіль розцінюється як семантична помилка.

OPERATOR:

Оператор просто додається в convertedData.

DECLARATION:

Якщо наступна лексема після цієї - незайняте ім'я для змінної, то оголошується нова змінна з ім'ям наступної лексеми, яка зберігатиме зазначений тип даних, що вказак в цій. У решті випадків таке використання лексеми, що відповідає за тип даних, вважається семантичною помилкою. Лексеми з типом DECLARATION у convertedData не додаються.

Дотримуючись цих правил, вираз *string s = "qq" + 15.1*, коли *s* зберігає в собі тип STRING, буде в convertedData виглядати так: STRING = STRING + FLOAT.

Тепер, коли у нас відомий список операторів та типи даних, з якими ці оператори застосовуються, ми можемо перейти до другої частини методу solve – обчислення.

Метод DATA_TYPE binaryOperationCheck (Operator operator, DATA_TYPE var1, DATA_TYPE var2) намагається застосувати оператор operator до типів даних var1 та var2. Якщо операція пройшла успішно, повертається тип даних результату. Наприклад:

- Результатом додавання двох INT буде INT;
- Результатом додавання INT і FLOAT буде FLOAT;
- Результатом додавання INT і STRING буде STRING;
- Результатом додавання STRING і STRING буде STRING;
- Результатом додавання STRING і BOOL буде BOOL;
- Спроба наприклад скласти BOOL і BOOL, або відняти INT з STRING, призведе до появи семантичної помилки.

Аналогічно, метод DATA_TYPE unaryOperationCheck (Operator operator, DATA_TYPE var1) намагається застосувати унарний оператор типу даних var1, і повертає тип даних, якщо операція виконалася успішно. Наприклад, застосування опретора "логічне не" до BOOL повертає тип BOOL, а спроба застосувати до будь-якого іншого типу є семантичною помилкою.

Для кожного оператора у відсортованому списку за шарами та пріоритетом проводиться наступне:

1) Якщо оператор бінарний, то береться наступна та попередня лексеми. Якщо якоїсь із лексем не існує (тобто якщо оператор є першою або останньою лексемою у списку), то це вважається семантичною помилкою, оскільки бінарний оператор може застосовуватися лише

до двох операндів. Якщо якась із цих двох лексем є не типом даних, а оператором, то це також вважається семантичною помилкою, оскільки оператор не може застосовуватися до іншого оператора (наприклад, два оператори поспіль: "2 + * 6"). Для унарних операторів проводяться аналогічні перевірки.

2) За допомогою методів DATA_TYPE unaryOperationCheck (Operator operator, DATA_TYPE var1) та DATA_TYPE binaryOperationCheck (Operator operator, DATA_TYPE var1, DATA_TYPE var2) до операторів намагаються бути застосовані до отриманих типів даних. Якщо операція успішна, то типи даних та оператор замінюються на результат цих методів (рис. 5.7).

Рисунок 5.7 - Процес обчислення виразу в convertedData

Процес заміни повторюється, поки в convertedData не залишиться один оператор.

5.4 Логічний аналіз та генерація коду

Якщо останній оператор не є оператором присвоєння, перевіряється, чи ця команда знаходиться в блоці операції. Якщо ця команда перебуває у блоці операції, і останній оператор не є присвоюванням, це означає, що

результат виразу ніде не зберігається і користувачеві не виводиться, що є логічною помилкою.

Результат виразу може ніде не зберігається, якщо результатом виразу є BOOL тип даних, і цей вираз знаходиться в блоці умови. У такому разі генерується if-код за участю цього виразу ("*a<15+b*" -> *if(a<15+b)*").

Також результат виразу може ніде не зберігається, якщо вираз знаходиться в блоці виведення, у такому випадку генерується код виведення за участю цього виразу ("*7 - 4*" -> "*cout << (7 - 4) << endl;*").

Для генерації коду використовується **клас CodeGenerator**, який має такі поля та методи:

String code - основний блок коду, який наповнюється кодом під час перебору блоків;

int indents - кількість відступів у цій точці коду безпосередньо залежить від глибини контексту;

void addMain(AbstractDiagramNode currentBlock) - додає блок коду або *int main(){,* або *назва_функції(){*, якщо аналізований блок - блок препроцесса.

void add (String str) - додає в код рядок з урахуванням відступу. Використовується здебільшого, коли треба просто додати рядок (*int a = 15;*)

void addIf(String str) - додає в код рядок *if(str){*, збільшує відступ indents на один;

void addElse() - додає в код рядок *else{*, збільшує відступ indents на один;

void addWhile(String str) - додає в код рядок *while(str){*, збільшує відступ indents на один;

void closeBranch() - додає в код "}", знижує відступ indents на один;

String getGeneratedCode() - повертає згенерований код у рядку code.

Щоразу, коли метод solve завершується успіхом, вираз, який йому передано, передається у певному вигляді екземпляр CodeGenerator:

Блок операції – вираз передається у вигляді, в якому його записав користувач;

Блок умови - якщо true-гілка ще не пройдена, то у вигляді if-коду, якщо пройдена - у вигляді else-коду. Якщо в блоці умови знаходиться вираз, результат обчислення якого відрізняється від BOOL, це розглядається як логічна помилка. Якщо блок умови містить більше ніж один рядок, то це також є логічною помилкою, тому що в if може бути лише один вираз;

Блок введення-виводу - якщо вираз усередині є оголошенням змінної, то генерується код з оголошенням змінної та привласненням їй значення, яке введе користувач (*int n -> int n; cin >> n;*).

В іншому випадку вираз усередині буде використано в коді виведення в консоль (*"7 - 4" -> "cout << (7 - 4) << endl;"*);

Блок циклу - якщо це блок, що відкриває цикл, то у вигляді while-коду; якщо блок закриває цикл, то у вигляді закриваючої фігурної дужки "}". Відкриваючий блок циклу повинен мати два рядки: у першому рядку має знаходитися ім'я циклу у форматі рядка-костнанти (тобто оточеної лапками), а в другому рядку - умова, за якої цей цикл виконується. Вираз за умови циклу повинен мати рішення типу BOOL. Блок кінця може мати лише один рядок із назвою циклу у форматі рядка-константи. Всі інші конструкції сприймаються як логічна помилка;

Блок препроцесу повинен містити ім'я функції без аргументу (назва_функції()). Якщо ім'я функції містить нелегальні символи, якщо ім'я функції не закінчується на "()", або якщо всередині цього блоку якийсь вираз, це розглядається як логічна помилка. Якщо всі правила дотримані, то створюється новий SchemeCompiler, який намагається знайти помилки та згенерувати код для схеми, що знаходиться всередині цього блоку-препроцесу. Згенерований код зі схеми всередині цього блоку буде потім доданий до результату генерації первинної схеми таким чином, щоб внутрішні функції були оголошені раніше, ніж їх використання.

Після генерації коду відбувається вибір наступного блоку за правилами, зазначеними у розділі 2.3 - порядок обходу схеми.

Алгоритм завершується, коли доходить до блоку кінця схеми.

Після цього проводиться остання перевірка - всі відкриті цикли повинні бути закриті, причому в послідовності, зворотній тій, в якій відкривалися. Ця перевірка відбувається з використанням алгоритмів ChainUp та ChainDown. Якщо якийсь із циклів не закритий, або порушена послідовність закриття циклів, це сприймається як логічна помилка.

Таким чином, якщо алгоритм перебрав всі блоки і не виявив структурних, синтаксичних, семантичних або логічних помилок, користувач отримує згенерований код на основі блок-схеми, яку він створив.

5.5 Тестування програмного модулю

Розробка присвячена програмній реалізації алгоритму пошуку помилок, який був втілений в метод класу DiagramPanel compileScheme(Scheme scheme). Цей метод створює екземпляр класу SchemeCompiler та передає йому схему, яку необхідно перевірити на помилки. Мета тестування – перевірити алгоритм пошуку помилок на здатність виявляти помилки у схемах. Помилки, які може зробити користувач, можна віднести до чотирьох класів:

- структурні;
- синтаксичні;
- семантичні;
- логічні.

Деякі структурні помилки перевіряються перед тим, як передати схему екземпляру SchemeCompiler:

- На схемі відсутній блок початку;

- На схемі відсутній блок кінця;
- На схемі є блоки, до яких неможливо потрапити з початку;
- На схемі є блоки, з яких неможливо потрапити до кінця;
- На схемі присутні блоки умов, у яких менше двох виходів із блоку.

Тестування алгоритму було проведено методом білого ящика. Було створено понад 50 кейсів (схем з помилками), внутрішні помилки яких можна віднести до одного з чотирьох перерахованих класів. Ці кейси використовуються у тестуванні щоразу, коли в програму вводиться новий функціонал. Кейс передається алгоритму, а на виході чекається повідомлення про помилку, що знаходиться всередині схеми.

Як бібліотека з функціоналом для тестування була обрана junit (v. 4.0), а основним механізмом для перевірки алгоритму на працездатність - очікування виключення SchemeCompilationException з певним повідомленням всередині.

Таблиця 5.3 - Кейси перед запуском алгоритму

Опис помилки	Очікувана частина з повідомлення	Файл, який містить схему з цією помилкою
На схемі відсутній блок початку	На схемі відсутній блок початку	test_start.json
На схемі відсутній блок кінця	На схемі відсутній блок кінця	test_end.json
На схемі є блоки, які відірвані від схеми	У наступні блоки неможливо потрапити з блоку початку	test_outOfReach_start.json

Продовження таблиці 5.3 - Кейси перед запуском алгоритму

Опис помилки	Очікувана частина з повідомлення	Файл, який містить схему з цією помилкою
На схемі є самозамикаючі структури	З наступних блоків неможливість потрапити до блоку кінця	test_outOfReach_end.json
У якогось блоку умов менше двох стрілок-виходів	Блоки умов повинні мати рівно два виходи	test_conditionBlock.json

Всі подальші перевірки відбуваються у екземплярі класу SchemeCompiler у методі String compile (Scheme scheme). Розглянемо кейси для структурних помилок, найважливіші з яких – помилки, пов'язані з передачею управління.

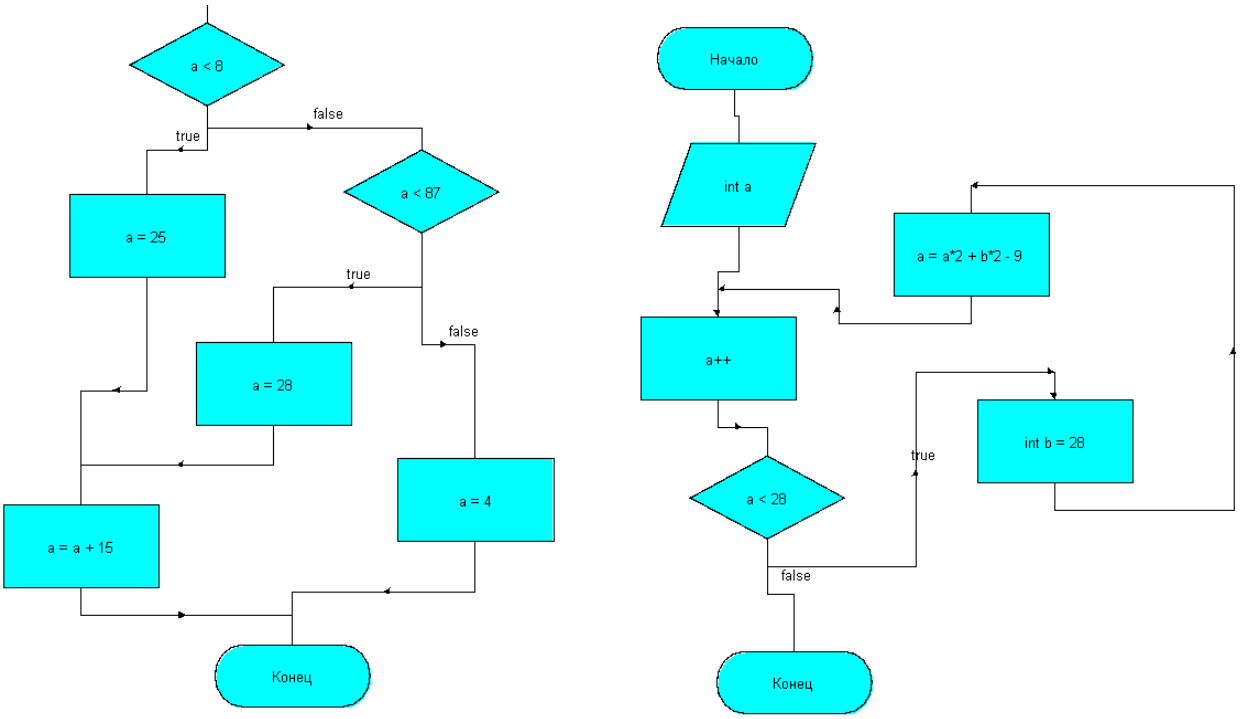


Рисунок 5.8 - Файли схем з помилками: test_controlTransfer1.json (зліва), test_controlTransfer2.json (справа)

Для цього типу помилок було створено п'ять кейсів, два з яких зображені на рисунку 5.8.

Таблиця 5.4 - Структурні помилки

Опис помилки	Очікувана частина з повідомлення	Файл, який містить схему з цією помилкою
Передача управління в іншу контекстну область (вар. з умовою)	Управління не може бути передано команді в іншій контекстній області	test_controlTransfer1.json
Заборонена передача управління до вже виконаної команди (вар. 1)	Повернення до раніше пройденої точки програми хоч і можливе, але є поганим тоном.	test_controlTransfer2.json
Заборонена передача управління до вже виконаної команди (вар. 2)	Повернення до раніше пройденої точки програми хоч і можливе, але є поганим тоном.	test_controlTransfer3.json
Передача управління в іншу контекстну область (вар. із входом в цикл)	Не можна входити в середину циклу, не пройшовши через блок, що його відкриває.	test_controlTransfer4.json
Передача управління в іншу контекстну область (вар. з виходом з циклу)	Не можна виходити з циклу, не пройшовши через блок, що його закриває.	test_controlTransfer5.json

Останні три структурні помилки пов'язані з циклами:

Таблиця 5.5 - Структурні помилки, пов'язані з циклами

Опис помилки	Очікувана частина з повідомлення	Файл, який містить схему з цією помилкою
На схемі є блок, який відкриває тіло циклу, але немає блоку, який його закриває.	Цикли з наступними іменами мають бути закриті.	test_notClosedLoops.json

Продовження таблиці 5.5 - Структурні помилки, пов'язані з циклами

Опис помилки	Очікувана частина з повідомлення	Файл, який містить схему з цією помилкою
Контекстні області закриваються у неправильній послідовності	Цикли повинні закриватися в послідовності, зворотній тій, як відкривалися.	test_loopClosingOrder.json
На схемі є блок, який закриває тіло циклу, але немає блоку, який його відкриває.	Невідоме ім'я циклу, що закривається. Якщо ви хочете оголосити новий цикл, у другому рядку має бути BOOL умова.	test_loopClosing.json

Наступний крок - синтаксичні помилки, пов'язані з розбивкою рядка на лексеми, некоректним використанням дужок та оголошенням змінних.

Таблиця 5.6 - Синтаксичні помилки

Рядок з помилкою	Очікувана частина з повідомлення	Файл, який містить помилку
25q	Невідома лексема 25q	test_syntax1.json
"25""ss80"	Невідома лексема "25""	test_syntax2.json
truel	Вираз truel не є операцією	test_syntax3.json
25.5.5	Невідома лексема	test_syntax4.json
int a = 2(56 * 4)	Вираз 2 56 не є операцією	test_syntax5.json
int a = 2*(56 * (4)	Кількості дужок, що відкривають і закривають, відрізняються.	test_syntax6.json
int a = 2*+56 * 4	Невідома лексема *+	test_syntax7.json
int (a = 2 * 56 - 4)	За оголошеним типом int повинна слідувати змінна	test_syntax8.json
int bool = 8	За оголошеним типом int повинна слідувати змінна	test_syntax9.json

Наступний крок - семантичні помилки, пов'язані з використанням змінних, операторів та типами даних.

Таблиця 5.7 - Семантичні помилки

Блок з помилкою	Очікувана частина з повідомлення	Файл, який містить схему з цією помилкою
int c = a - 8	Невідома змінна *a*	test_semantic1.json
bool b = true; int c = b - 8	Оператор - незастосовний до BOOL та INT	test_semantic2.json
int c = 5; int c = 15	Змінна *c* таким ім'ям вже існує в цій області	test_semantic3.json
Використання змінної, оголошеної в іншому контексті	Невідома змінна	test_syntax4.json
int c у блоці операції	Усі оголошені змінні або повинні мати значення за умовчанням, або перебувати в блоці введення користувача	test_syntax5.json
int c = 8 у блоці умови	Оголошення змінних допустиме лише в блоці операцій або введення	test_syntax6.json
int c = 8; c = "8"	Змінній *c* типу INT не може бути присвоєно значення STRING	test_syntax9.json
true + 50	Оператор + не застосовується до BOOL та INT	test_syntax10.json
“ssq” - “25”	Оператор - не застосовується до STRING та STRING	test_syntax11.json

Продовження таблиці 5.7 - Семантичні помилки

Блок з помилкою	Очікувана частина з повідомлення	Файл, який містить схему з цією помилкою
50 * true	Оператор * не застосовується до INT та BOOL	test_syntax12.json
“225” / 2.5	Оператор / не застосовується до STRING та FLOAT	test_syntax13.json
!25	Оператор ! не застосовується до INT	test_syntax14.json
“ssq” || “2556”	Оператор || не застосовується до STRING та STRING	test_syntax15.json
false && 0	Оператор && не застосовується до BOOL та INT	test_syntax16.json
“25” < 25.5	Оператор < не застосовується до STRING та FLOAT	test_syntax17.json
“15”++	Оператор ++ не застосовується до STRING	test_syntax18.json
false --	Оператор - не застосовується до BOOL	test_syntax19.json

Передостанній крок – перевірка на логічні помилки.

Таблиця 5.8 - Логічні помилки

Помилка	Очікувана частина з повідомлення	Файл, який містить схему з цією помилкою
У блоці умови знаходиться вираз, результат обчислення якого не є типом BOOL	У блоці умови має бути BOOL вираз	test_logic1.json

Продовження таблиці 5.8 - Логічні помилки

Помилка	Очікувана частина з повідомлення	Файл, який містить схему з цією помилкою
Порожній блок із циклом	Блоки циклів повинні містити назву свого циклу у верхньому рядку.	test_logic2.json
У блоці умови більше одного рядка	У блоці умови може бути лише один рядок	test_logic3.json
Вираз перебуває у блоці операції, та його результат не записується в змінну (*5+15*)	Операція 5 + 15 не має сенсу, тому що її результат ніде не зберігається і не використовується	test_logic4.json
Порожній блок із процесом, який був визначений раніше	Усі функції мають бути пройменовані	test_logic5.json
У блоці процесу більше одного рядка	В одному блоці процесу може бути лише одна функція	test_logic6.json
Ім'я функції закінчується не пустим списком параметрів	Наприкінці імені функції повинні стояти дужки без параметрів "()"	test_logic7.json
Нелегальні для ідентифікаторів символи	У імені функції допустимі лише латинські літери та цифри	test_logic8.json
Ім'я функції починається з цифри	Ім'я функції не може починатися з цифри	test_logic9.json
У блоці циклу якийсь вираз замість STRING-константи на місці імені циклу	У першому рядку блоку циклу має знаходитись ім'я циклу у форматі STRING	test_logic10.json
Усередині циклу оголошується ще один із таким же ім'ям	Не можна всередині циклу оголосити цикл з таким самим ім'ям	test_logic11.json
У умові циклу вказано операцію, результат якої відмінний від типу BOOL	У другому рядку циклу, що відкривається, повинна знаходитися BOOL умова	test_logic12.json

Останній тип перевірки проводиться частково вручну - алгоритм генерує код із однієї з кейс-схем, а потім цей код запускається в якомусь C++ компіляторі. Також код переглядається на відповідності до схеми користувача.

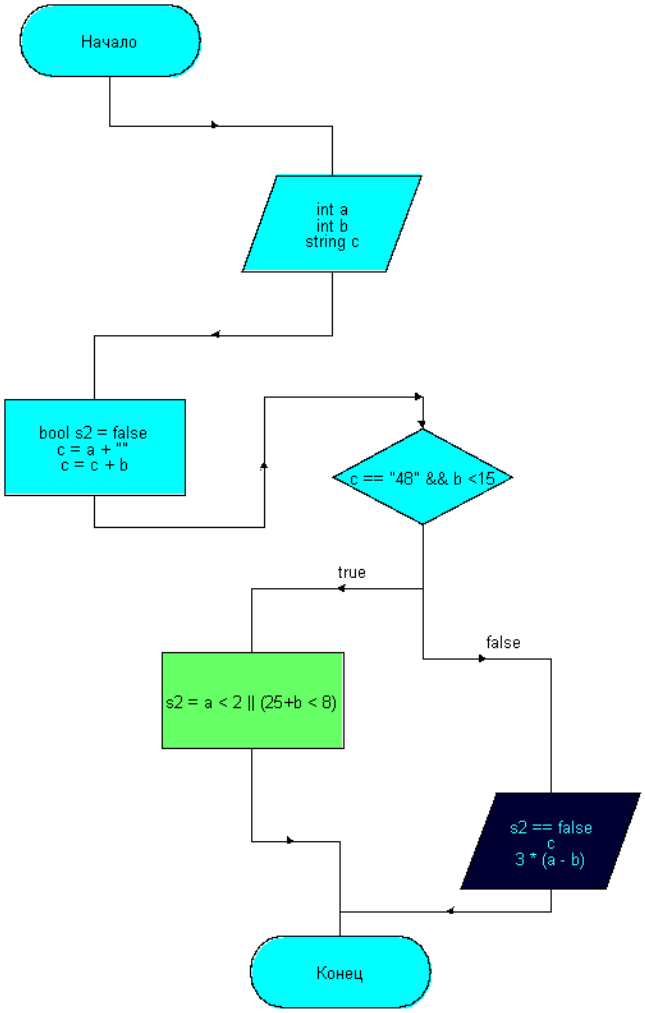


Рисунок 5.9 - Файл правильної схеми regular1.json

Згенерований C++ код для кейсу №1 (рис. 5.9):

```
#include <iostream>
using namespace std;

int main(){
        int a;
        cin >> a;
        int b;
        cin >> b;
```

```
        string c;
        cin >> c;
        bool s2 = false;
        c = to_string(a) + "";
        c = c + to_string(b);
        if(c == "48" && b <15){
                s2 = a < 2 || (25+b < 8);
        }
        else {
                cout << (s2 == false) << endl;
                cout << c << endl;
                cout << (3 * (a - b)) << endl;
        }
}
```

Кейс №2 був створений з акцентом на if-else розгалуження та закриття областей контексту (рис. 5.10).

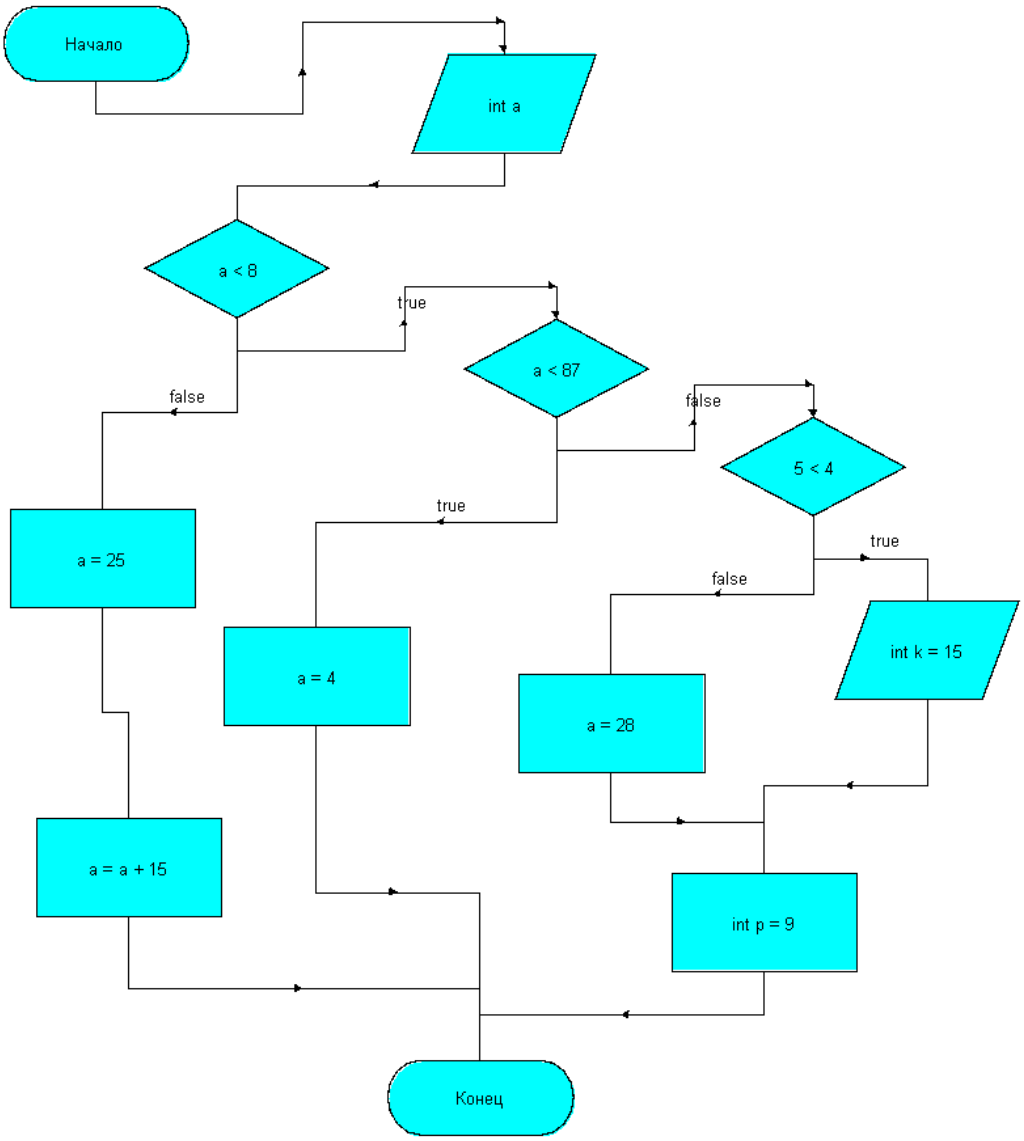


Рисунок 5.10 - Файл правильної схеми regular2.json

Згенерований C++ код для кейсу №2:

```
#include <iostream>
using namespace std;

int main(){
	int a;
	cin >> a;
	if(a < 8){
		if(a < 87){
			a = 4;
		}
		else {
			if(5 < 4){
				int k = 15;
				cin >> k;
			}
			else {
				a = 28;
			}
			int p = 9;
		}
	}
	else {
		a = 25;
		a = a + 15;
	}
}
```

Кейс №3 використовує блок процесу із вкладеною всередину схемою:

```
#include <iostream>
using namespace std;
void ssq(){
	int ss = 25;
	cin >> ss;
	int c = 25;
	string c2 = "12";
	c2 = to_string(c) + c2;
	if(ss < 15){
	}
	else {
		c2 = c2 + to_string(ss);
		cout << c2 << endl;
	}
}


int main(){
	bool b;
	cin >> b;
	b = true;
	int a = 20;
	string c = "str";
	while(a < 25){
```

```
                a = a + 1;
                if(b){
                        c = c + to_string(a);
                }
                else {
                        ssq();
                }
        }
}
```

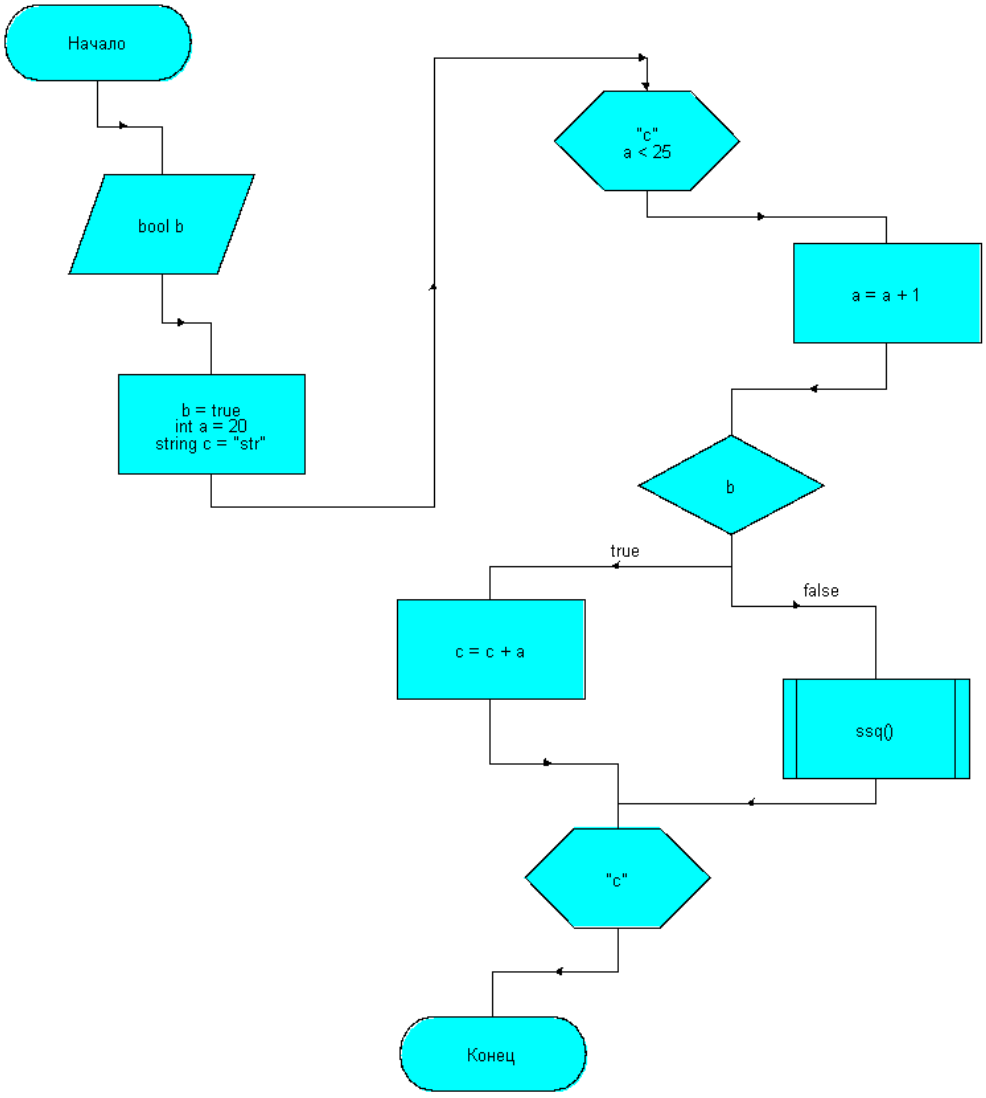


Рисунок 5.11 - Файл правильної схеми regular3.json

Вкладена схема у функцію ssq() має структуру, зображену на рисунку 5.12.

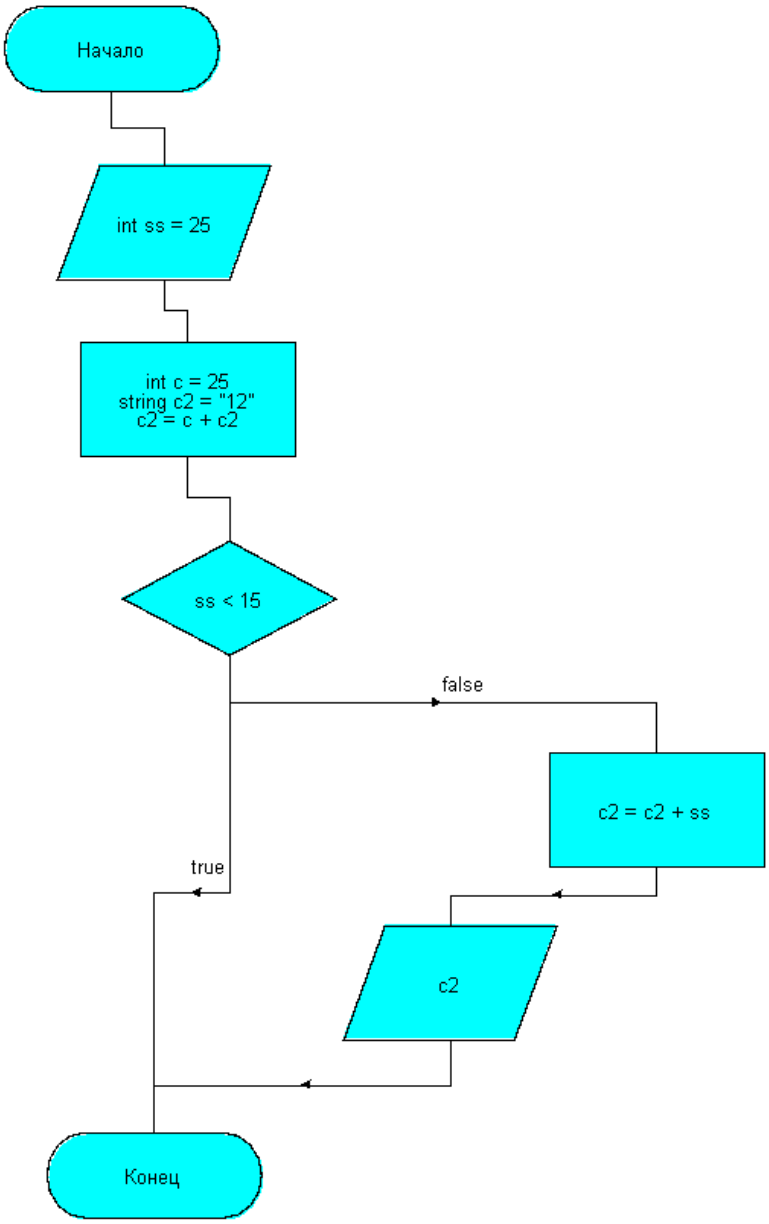


Рисунок 5.12 - Вкладена схема у regular3.json

Як і планувалося, код був згенерований так, щоб оголошення функції відбувалося раніше виклику функції:

Відсоток покриття тестами головного класу SchemeCompiler зображений на рисунку 5.13.

70% classes, 40% lines covered in package 'code'

Element	Class, %	Method, %	Line, %
DiagramGeneralization	100% (1/1)	20% (2/10)	7% (15/198)
DiagramHashMap	100% (1/1)	50% (2/4)	53% (17/32)
DiagramLabel	100% (1/1)	10% (1/10)	18% (3/16)
DiagramObject	100% (1/1)	12% (6/48)	15% (22/143)
DiagramPanel	72% (8/11)	21% (16/75)	29% (252/855)
DiagramParallelogram	100% (1/1)	15% (2/13)	20% (8/39)
DiagramPreprocess	100% (1/1)	14% (2/14)	16% (9/55)
DiagramRectangle	100% (1/1)	15% (2/13)	23% (8/34)
DiagramRhombus	100% (1/1)	21% (3/14)	21% (9/41)
DiagramTerminator	100% (1/1)	13% (2/15)	11% (5/43)
DiagramTerminatorEnd	100% (1/1)	100% (1/1)	100% (3/3)
DiagramTerminatorStart	100% (1/1)	100% (2/2)	100% (5/5)
LinesTableModel	0% (0/1)	0% (0/10)	0% (0/30)
Scheme	100% (1/1)	33% (3/9)	14% (17/118)
SchemeCompiler	100% (10/10)	100% (34/34)	90% (606/668)
Vertex	100% (1/1)	33% (1/3)	66% (4/6)

Рисунок 5.13 - Відсотки покриття тестами

Таким чином, у ході даного підрозділу було проведено тестування розробленого програмного забезпечення, у процесі якого тестами було покрито 100% методів основного класу SchemeCompiler.

6 РОЗРАХУНОК ЕФЕКТИВНОСТІ ПРОГРАМНОЇ СИСТЕМИ

Для визначення ефективності програмного продукту було запрошено 8 добровольців, які лише кілька днів тому почали вивчати програмування за допомогою курсів, збірника лекцій тощо. Було обрано 7 завдань різного рівня складності на основні поняття у структурному програмуванні (послідовні операції, розгалуження та цикли), вирішення яких може бути представлене у вигляді схеми алгоритму.

Добровольці не були обмежені інструментами, які можна використовувати для вивчення програмування, але з 4 них додатково користувалися розробленим програмним забезпеченням для візуалізації програмного коду та пошуку помилок у програмі Scheme Prof.

Добровольці в міру того, як навчалися, вирішували ці завдання і повідомляли рішення. Мета кожного з них – вирішити всі 7 завдань [10], максимальний час – 3 місяці. Результати винесено в таблицю 6.1.

Таблиця 6.1 - Показники ефективності

	З Scheme checker				Без Scheme checker			
	№1	№2	№3	№4	№5	№6	№7	№8
№1	6	12	14	18	16	21	22	25
№2	2	5	4	6	7	5	6	5
№3	2	4	4	7	4	5	7	3
№4	8	14	15	21	16	22	25	25
№5	2	4	2	4	4	5	3	5
№6	8	18	20	24	18	22	22	25
№7	1	2	2	2	3	2	3	3
Днів	29	60	61	82	68	82	88	91
Годин/день	4,5	2,5	2,5	1,5	2,5	2	2	1,5
Годин	130,5	150	152,5	123	170	164	176	136,5
Серед. арифм.				139				161,625

Враховуючи різну зайнятість людей, простий підрахунок витрачених днів не може бути показовим, необхідно додати ще одне параметр - кількість годин, яке у середньому витрачає людина щодня на вивчення програмування. Очікується, що людина, яка витрачає більше часу на день, впорається з вирішенням завдань швидше, тому для вимірювання ефективності буде використано добуток кількості днів на середню витрачену кількість годин на навчання.

У таблиці на перетині номера завдання та номера добровольця зазначено кількість витрачених на завдання днів. Помноживши цю кількість на зайнятість людини (годин на день, витрачених на вивчення програмування), та склавши цей показник для всіх завдань, можна обчислити кількість годин, які людина витратила на вивчення основ програмування. Знайшовши середнє арифметичне кожної групи (з використанням Scheme Prof і без нього) можна побачити, що люди, які використовують Scheme Prof, вивчили основи програмування на 14% швидше, що показує ефективність розробленого програмного забезпечення.

ВИСНОВКИ

Кваліфікаційна робота присвячена проектуванню, розробці та тестуванню програмного забезпечення для навчання алгоритмізації.

Спочатку було проведено аналіз існуючої предметної області, знайдено проблеми у ній, і було проведено пошук існуючих рішень.

У другому розділі було створено алгоритм, що дозволяє шукати помилки у створеній користувачем програмі, алгоритм охоплює пошук структурних, синтаксичних, семантичних та логічних помилок.

Потім було складено список функціональних і нефункціональних вимог до системи, що розробляється. Як основу для розробки було обрано графічний редактор. Далі було проведено програмну реалізацію розробленого алгоритму, а також реалізовано функцію генерації коду на основі псевдокоду, який користувач ввів у кожен блок, а також на основі послідовності, в якій ці блоки пов'язані.

В передостанньому розділі було проведено тестування, яке охопило весь розроблений у ході програмної реалізації клас, перевіряючи працездатність алгоритму пошуку помилок майже на 50 кейсах. Згідно з проведеними випробуваннями ефективності завдяки створеному програмному забезпеченню вдалося зменшити витрачений на навчання час майже на 14%.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Globallogic. Сайт компанії. URL: https://www.globallogic.com/ua/about/news/ukraine-global-talent-project/ (дата звернення 11.08.2021)
2. Bestprogrammer, сайт для публікацій та новин. URL: https://bestprogrammer.ru/izuchenie/skolko-nuzhno-vremeni-chtoby-nauchitsya-programmirovat (дата звернення 25.07.2021)
3. Skillbox, сайт компанії. URL: https://skillbox.ru/media/code/skolko_nuzhno_uchitsya_na_programmista_s_nulya/ (дата звернення 15.09.2021)
4. Dou.ua, сайт компанії. URL: https://dou.ua/forums/topic/24085/ (дата звернення 25.08.2021)
5. DOI: https://doi.org/10.15276/aait.01.2020.7. Kolesnikova, K. V., Lukianov, D. V. & Olekh T. M. “The Role of a Higher Education Diploma in the Professional Career of the Specialist in the Future”. Applied Aspects of Information Technology. Publ. Nauka i Tekhnika. Odessa: Ukraine. 2020; Vol. 3 No. 1: 456–466.
6. Tiobe, сайт компанії. URL: https://www.tiobe.com/tiobe-index/ (дата звернення 22.07.2021)
7. HOPL, сайт компанії. URL: https://hopl.info/ (дата звернення 23.07.2021)
8. DOI: https://doi.org/10.15276/hait.04.2019.8, Larshin, V. P. & Lishchenko, N. V. “Educational Technology Information Support”. Herald of Advanced Information Technology. Publ. Science i Technical. 2019; Vol.2 No.4:p.317–327. Odesa. Ukraine.
9. ГОСТ 19.701-90 (ISO 5807-85). Схеми алгоритмів, програм, даних і систем. Позначення умовні і правила виконання.

10. Itmathrepetitor, сайт компанії. URL: http://www.itmathrepetitor.ru/prog/zadachi-na-vychisleniya/ (дата звернення 25.09.2021).