Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя
(повне найменування вищого навчального закладу)
Факультет комп'ютерно-інформаційних систем і програмної інженерії
(назва факультету)
Кафедра комп'ютерних систем та мереж
(повна назва кафедри)

ПОЯСНЮВАЛЬНА ЗАПИСКА

до дипломної роботи

Магістра
(освітній ступінь)

на тему: Методи та засоби для функціонального програмування мікроконтролерів

Виконав: студент (ка) IV курсу, групи СІМ-62
спеціальності 123
Комп'ютерна інженерія
(шифр і назва спеціальності)
(підпис) Сурдомир В. П.
(прізвище та ініціали)

Керівник (підпис) Лукіяк Х. М.
(прізвище та ініціали)

Нормоконтроль (підпис) Тимя Є. В.
(прізвище та ініціали)

Рецензент (підпис) Сифенський Ю. І.
(прізвище та ініціали)

м. Тернопіль – 2019

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя
(повне найменування вищого навчального закладу)

Факультет комп'ютерно-інформаційних систем і програмної інженерії
Кафедра комп'ютерних систем та мереж
Освітній рівень магістр
Напрям підготовки
(шифр і назва)
Спеціальність 123 Комп'ютерна інженерія
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри КС
Осухівська Г. М.
«30» 09 2019 р.

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЕКТ (РОБОТУ) СТУДЕНТУ

Цуцмир Валентин Петрович
(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) Методи та засоби для функційного програмування мікроконтролерів

Керівник проекту (роботи) Луців Андрій Миколайович к.т.н. доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом по університету від «27» 09 2019 року №4/7-954

2. Термін подання студентом проекту (роботи) 27 грудня 2019 року

3. Вихідні дані до проекту (роботи) Мови та технології для програмування мікроконтролерів, парадигми програмування. Математичне забезпечення функційної парадигми

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Аналіз предметної області програмування мікроконтролерів
2. Математичне забезпечення функційної парадигми
3. Розробка програми на функційних мовах
4. Обґрунтування економічної ефективності
5. Охорона праці та безпеки в надзвичайних ситуаціях
6. Екологія

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
Актуальність та мета роботи, об'єкт дослідження, предмет дослідження; Основний зміст роботи; таблиця з парадигмами програмування; порівняння функційної та об'єктно-орієнтованої парадигми; підтримка функційної парадигми та особливості програмування мікроконтролерів різними мовами; приклад програми на мові Haskell; приклад програми на мові C++; висновки;

6. Консультанти розділів проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата: завдання видав	Підпис, дата: завдання прийняв
Екологія	Лесенко О.М., доц.	[illegible]	22.11.19
Безпека в НС	Струцюк В.С., ст. викл. каф. ОС	[illegible]	[illegible]
Обґрунт. екон. ефект.	Кириз Н.Б.	[illegible]	[illegible]
Охорона праці	Осухівська Г.М.	22.11.19	22.11.19

7. Дата видачі завдання 30.09.2019

КАЛЕНДАРНИЙ ПЛАН

№ з п	Назва етапів дипломного проекту (роботи)	Термін виконання етапів проекту (роботи)	Примітка
1	Отримання завдання	30.09.19	Виконано
2	Аналіз завдання	1.10.19	Виконано
3	Огляд літературних джерел	4.10.19	Виконано
4	Аналіз предметної області [illegible]	8.10.19	Виконано
5	Аналіз і дослідження існуючих парадигм програмування	12.10.19	Виконано
6	Аналіз математичного забезпечення функціональної парадигми програмування	14.10.19	Виконано
7	Розробка математичного забезпечення для аналізаторів на функціональній мові та порівняння з класичною програмою на основі [illegible]	18.10.19	Виконано
8	Обґрунтування економічної ефективності	1.11.19	Виконано
9	Охорона праці та безпека в надзвичайних ситуаціях	4.11.19	Виконано
10	Екологія	6.11.19	Виконано
11	Оформлення пояснювальної записки	13.11.19	Виконано
12	Оформлення графічної частини	15.11.19	Виконано
13	Рецензування роботи	18.11.19	Виконано
14	Захист роботи	27.11.19	

Студент ____________ (підпис) Сидорчук В.П. (прізвище та ініціали)

Керівник проекту (роботи) ____________ (підпис) Луцків А.М. (прізвище та ініціали)

АНОТАЦІЯ

Дипломна робота // Методи та засоби для функційного програмування мікроконтролерів // Судомира Валентина Петровича // Тернопільський національний технічний університет імені Івана Пулюя. Факультет комп'ютерно - інформаційних систем та програмної інженерії. Кафедра комп'ютерних систем та мереж // група СІм-62 // Тернопіль, 2019 р. // с. - 112, рис. - 41, бібліогр. - 27.

Ключові слова: ПАРАДИГМА ПРОГРАМУВАННЯ, МІКРОКОНТРОЛЕР, ФУНКЦІЙНА, ОБ'ЄКТНО-ОРІЄНТОВАНА, ІМПЕРАТИВНА.

Темою даної дипломної роботи є «Методи та засоби для функційного програмування мікроконтролерів».

Мета роботи полягає у досліджені методів та засобів для функційного програмування мікроконтролерів.

У дипломній роботі проаналізовано сучасні парадигми програмування, які використовуються при програмуванні мікроконтролерів, описано кожну з них та порівняно; обґрунтовано вибір оптимальної парадигми програмування для мікроконтролерів.

На основі проведених досліджень було розроблено рекомендації що до вибору парадигми та мови для програмування мікроконтролерів.

ANNOTATION

Graduate thesis // Methods and tools for microcontrollers` functional programming // Sudomyr Valentyn Petrovych // Ternopil Ivan Puluj National Technical University. Faculty of Computer Information Systems and Software Engineering. Computer Systems and Networks Department // group СІм-62 // Ternopil, 2019 y. // pages - 112, imges - 41, bibliography - 27.

Keywords: PROGRAMMING PARADIGM, MICROCONTROLLER FUNCTIONAL, OBJECT-ORIENTED, IMPERATIVE.

The topic of this thesis is " Methods and tools for microcontrollers` functional programming ".

The purpose of the work is to investigate methods and tools for functional programming of microcontrollers.

The thesis analyzes modern programming paradigms used in programming microcontrollers, describes each of them and compare them; justified the choice of the optimal programming paradigm for microcontrollers.

Based on the conducted research, recommendations were developed regarding the choice of paradigm and language for programming of microcontrollers.

ЗМІСТ

ВСТУП

Актуальність теми роботи. Сучасні мови програмування мікроконтролерів при створенні програмного забезпечення в основному відносяться, до імперативних та об’єктно-орієнтованих парадигми. Популярність використання мов що належать до цих парадигм, пояснюється тим, що простотою використання, а також надійністю, високою продуктивністю та сумісністю з великою кількістю уже написаних бібліотек для використання зовнішніх пристроїв накшталт датчиків. Проте останнім часом все популярнішою стає функційна парадигма програмування. Сучасні найбільш популярні об’єктно орієнтовані мови програмування для повноцінних комп’ютерів, такі як C# та Java, уже підтримують багато корисних особливостей функційного програмування. Для програмування мікроконтролерів теж було б корисно все частіше використовувати функційну парадигму і в особливості мову програмування Haskell, код програм на якій значно менший та зрозуміліший читачеві при написанні аналогічних програм ніж імперативними мовами.

Метою роботи є дослідження методів та засобів для функційного програмування мікроконтролерів.

Для досягнення поставленої мети потрібно розв’язати такі задачі:

– проаналізувати імперативну, об’єктно орієнтовану та функційну парадигми програмування, визначити переваги та недоліки кожної з них та порівняти;

– проаналізувати задачі які стоять при програмуванні мікроконтролерів;

– проаналізувати функційну парадигму та провести компаративний її аналіз з іншими парадигмами програмування, які застосовуються при програмуванні мікроконтролерів;

– обґрунтувати доцільність використання однієї з парадигм;

– провести верифікацію запропонованих підходів, шляхом прототипу;

– розробити рекомендації, яких необхідно дотримуватись при програмуванні мікроконтролерів з використанням запропонованої парадигми.

Об’єкт дослідження: процес вибору парадигми та мови при написанні програмного забезпечення для мікроконтролерів.

Предмет дослідження: технології для програмування мікроконтролерів використовуючи функційну парадигму програмування.

Методи дослідження. Для вирішення поставлених задач використано наступні методи теорії програмування та математичної статистики, об’єктно орієнтований, імперативний та функційний підхід до розробки програм.

Наукова новизна одержаних результатів:

– уперше запропоновано метод програмування мікроконтролерів використовуючи функційну парадигму програмування мовами Haskell або C++, проведено дослідження що до ефективності використання функційної парадигми при створенні програмного забезпечення для мікроконтролерів, та порівняння її до об’єктно-орієнтованої та імперативної парадигм.

– Проведено експеримент, в ході якого порівнювалась швидкість написання програмного продукту використовуючи різні парадигми програмування.

Практичне значення одержаних результатів. Розроблені рекомендації та практики дають змогу вибирати кращу парадигму для написання програмного продукту для мікроконтролера в залежності від функцій які він має виконувати.

Публікації. Результати дослідження апробовано на II міжнародній студентській науково-технічній конференції «Природничі та гуманітарні науки. Актуальні питання» (25-26 квітня 2019 р.) та VIII міжнародній науково-технічній конференції молодих учених та студентів «Актуальні задачі сучасних технологій» (27-28 листопада 2019 р.) Тернопільського національного технічного університету імені Івана Пулюя у вигляді тез конференцій.

Структура роботи. Робота складається з пояснювальної записки та графічної частини. Пояснювальна записка складається із вступу, шести розділів,

висновків, список використаних джерел та додатків. Обсяг роботи: пояснювальна записка – 113 аркушів формату А4, графічна частина – 10 аркушів формату А1.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ПРОГРАМУВАННЯ МІКРОКОНТРОЛЕРІВ

1.1. Мікропроцесори, мікрокомп'ютери, мікроконтролери

У той час як всі мікропроцесори, мікрокомп'ютери та мікроконтролери володіють певними схожими характеристиками, а їх терміни часто використовуються попеременно, існують між ними і певні відмінності, що дають змогу класифікувати їх в окремих категоріях.

1.1.1. Мікропроцесори. Найпростішим з цих трьох категорій є мікропроцесор. Також відомі як центральні процесори, ці пристрої, як правило, знаходяться в центрі набагато більшої системи, наприклад настільного комп'ютера, і в основному використовуються для обробки даних. Вони, зазвичай, складаються з арифметично-логічного пристрою, декодера команд, ряду регістрів і ліній цифрового входу / виходу [2], які зображені на рис. 1.1.

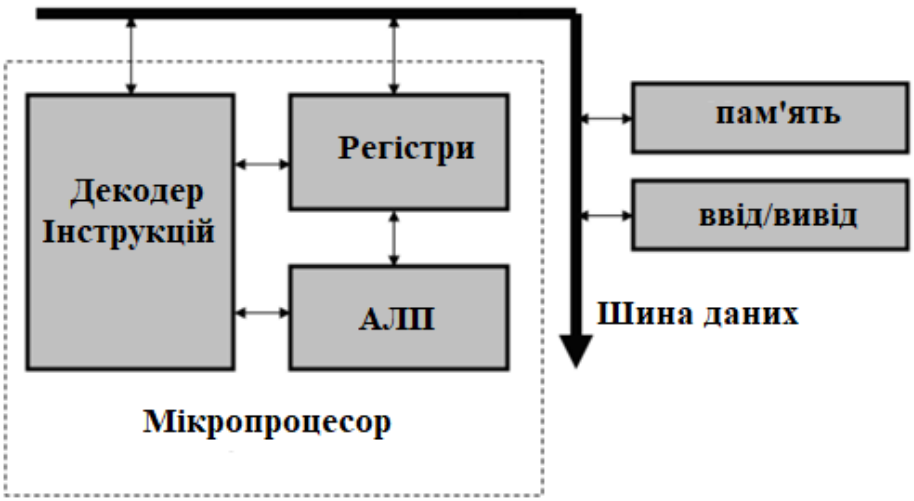

Рис. 1.1. Базові компоненти мікропроцесора

Деякі процесори також включають в себе місця пам'яті, такі як кеш або стек, які здебільшого використовуються для більш швидкого тимчасового зберігання і зчитування даних, ніж при доступі до системної пам'яті. Крім того, процесор повинен підключатися до певної форми шини даних для доступу до пам'яті і вхідних / вихідних периферійних пристроїв, зовнішніх для самого процесора.

Залежно від архітектури пам'яті мікропроцесор може мати лише кілька регістрів, таких як програмний лічильник для відстеження адреси наступної інструкції та регістр команд для завантаження і зберігання наступної команди, або ж можуть бути десятки регістрів. Ці додаткові регістри відомі як регістри загального призначення, що зберігають дані під час їх використання.

1.1.2. Мікрокомп'ютери. Мікрокомп'ютер містить всі компоненти комп'ютера на невеликій печатній платі, але не на одному кристалі. Цей термін в цілому поширюється на ноутбуки і настільні комп'ютери, однак не використовуються для цих пристроїв [2]. Компонентні пристрої мікрокомп'ютера складаються з процесора, пам'яті та інших пристроїв зберігання даних, а також пристроїв вводу-виводу, які зображені на рис. 1.2.

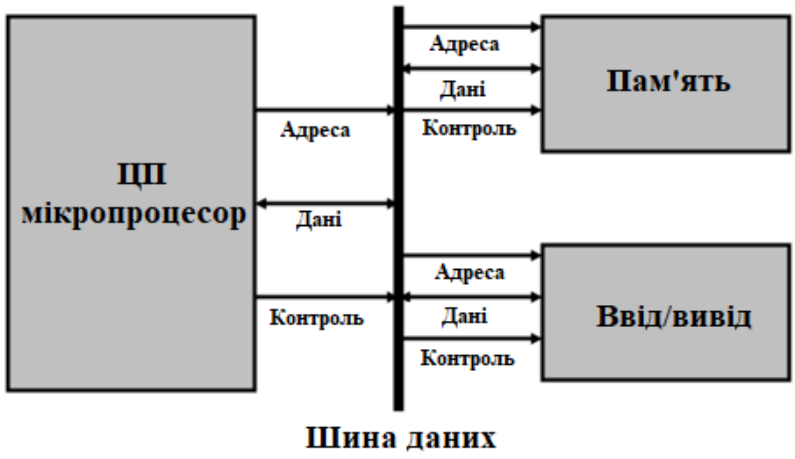

Рис. 1.2. Базові компоненти мікрокомп’ютера

Кілька прикладів пристроїв введення-виведення включають клавіатуру, дисплей, мережу тощо; але ними можуть і бути будь-які пристрої, які мікрокомп'ютер використовує для збору або поширення інформації.

1.1.3. Мікроконтролери. Мікроконтролер є, у певному сенсі, перехрестям між мікропроцесором і мікрокомп'ютером. Як і мікропроцесори, термін мікроконтролер відноситься до одного пристрою. Однак мікроконтролер містить весь мікрокомп'ютер на одній мікросхемі. Тому мікроконтролер буде мати процесор, вбудовану пам'ять, а також безліч пристроїв вводу-виводу. При використанні мікроконтролера замість мікрокомп'ютера спрощується загальний дизайн, для досягнення цього жертвується гнучкістю. Мікрокомп'ютер може бути налаштований так, щоб мати певну кількість пам'яті або прикріплених пристроїв. Мікроконтролери, як правило, обмежені розмірами пам'яті та периферійними пристроями виробником.

Існує велика кількість варіантів мікроконтролерів та їх можливостей, однак це може бути обмеженням в деяких обставинах. Оскільки мікроконтролери призначені скоріше як автономні пристрої для збору та контролю даних, а не для взаємодії з людиною, для чого часто використовуються мікрокомп'ютери, їхні стандартні пристрої вводу-виводу відрізняються. Аналого-цифрові перетворювачі (АЦП), таймери і зовнішні переривання є стандартними периферійними пристроями, що знаходяться на мікроконтролерах, тоді як клавіатури, монітори та інші пристрої, що використовуються щодня для управління персональним комп'ютером.

1.2. Програмування мікроконтролерів

На відміну від способів програмування комп’ютерів та мікрокомп’ютерів, програмування мікроконтролерів дуже обмежено і вимагає раціонального використання ресурсів у середовищі де вони обмежені.

Мікроконтролери часто використовують 8-бітні процесори з робочими частотами до 40 МГц та оперативною пам’яттю від 0,5 до 8 Кб. Навіть більші мікроконтролери, такі як сімейство ESP з 32-бітними процесорами, робочі частоти якого сягають до 240 МГц і об’ємом пам’яті 520 КБ, непридатні для сучасного Linux ядра та користувацького простору, не кажучи вже про стек технологій для декларативного програмування. Для програмування пристроїв такого типу традиційно було лише два варіанти. Доступний метод, який часто використовується при навчанні початкових та проміжних курсів, - це графічна мова на основі програмного забезпечення на зразок scratch, яка використовує підхід перекладу шаблонів коду, який, виглядає як блоки, які виконують певні операції, відповідає фактичному вихідному коду C [1]. Другий підхід, який застосовується на академічному чи вищому рівнях, - це програмувати безпосередньо, використовуючи мови C та C++.

Обидва підходи обмежують користувача щодо доступних сучасних парадигм програмування. Імперативне програмування, схоже, не має реальних альтернатив, навіть незважаючи на те, що такі системи, які можуть бути обладнані датчиками, кнопками, індикаторами, дисплеями тощо, в принципі добре підходять для програмування за допомогою інших парадигм. Особливо в таких інтерактивних додатках, як екологічне зондування та робототехніка, бажані декларативні підходи, орієнтовані на події, або на основі правил

1.3. Парадигми програмування

Парадигма програмування - це філософія, стиль і загальний підхід до написання коду. Більшість визначень даного терміну є настільки широкими, що вони є досить марними, цей термін має більший сенс при обговоренні конкретних парадигм [3]. Парадигма в першу чергу визначається базовою програмною одиницею і самим принципом досягнення модульності програми. В якості цієї одиниці виступають визначення (декларативне, функційне

програмування), дії (імперативне програмування), правила (продукційне програмування), діаграми переходів (автоматне програмування) і ін. сутності [4]. У сучасній індустрії парадигма програмування дуже часто визначається набором інструментів програміста, а саме, мовою програмування і використовуваними бібліотеками, тому парадигм за останні роки виникло дуже багато, їх розподіл зображено на рис. 1.3.

Рис. 1.3. Розподіл парадигм програмування

Парадигма програмування визначає те, в яких термінах програміст описує логіку програми. Наприклад, в імперативному програмуванні програма описується як послідовність дій, а в функційному програмуванні представляється у вигляді вираження і безлічі визначень функцій. У популярному об'єктно-орієнтованому програмуванні програму прийнято розглядати як набір взаємодіючих об'єктів. ООП, в основному, є по суті імперативним

програмуванням, доповненим принципом інкапсуляції даних і методів в об'єкт і спадкуванням тобто принципом повторного використання розробленого функціоналу [5]. Важливо відзначити, що парадигма програмування не визначається однозначно мовою програмування - багато сучасних мов програмування допускають використання різних парадигм. Так на мові Сі, яка не є об'єктно-орієнтованою, можна писати об'єктно-орієнтованим чином, а на Ruby або Java в основу яких в значній мірі покладена об'єктно-орієнтована парадигма, можна писати згідно стилю функційного програмування.

1.4. Імперативна парадигма програмування

Сьогодні імперативне програмування ще широко застосовується для створення програмного забезпечення для цифрових комп'ютерів. В основі цієї парадигми лежить погляд на те, що програми - це послідовність інструкцій чи команд, які змінюють деякий цифровий стан. Використовуючи функції, що залежать від ситуації, можна також змінити стан цифрового комп'ютера, щоб ініціювати зміни в оточенні комп'ютера. Приклади таких дій включають:

- друк символу на екрані (або моніторі);
- спрацьовування руки робота;
- включення лампочки;
- коригування швидкості руху літака.

Регулювання відповідності між станом комп’ютера та оточенням зазвичай здійснюється за допомогою комбінації апаратного та програмного забезпечення яке часто називають драйверами пристроїв, де в основному й застосовується дана парадигма [4].

1.4.1. Змінні та їх призначення.

Цифрові комп'ютери містять фіксовану кількість комірок пам'яті. Загалом, стан кожної з цих комірок пам'яті - це рівно один із двох станів, які часто називають "high" та "low", або ж просто "1" та "0".

Одиничні комірки пам’яті можуть використовуватися лише для представлення простих значень, таких як булеві ("істинне" та "хибне"). На більш високому рівні невеликі сукупності комірок пам'яті називаються словами [10]. На ще більш високому рівні об’єднання цих слів використовуються для представлення різних видів даних, таких як цілі числа, числа з плаваючою комою, символи та рядки. Стан комп’ютера - це сукупність станів усіх комірок пам'яті.

Кожен накопичувач має унікальну адресу. У мовах програмування низького рівня ці адреси можна явно використовувати в програмах. Однак існує маса недоліків явного використання таких місць, особливо, якщо на комп'ютері може бути величезна кількість одиниць пам’яті [10]. Наприклад, ПК із пам'яттю 32 МБ має приблизно 32 * 10 ^ 6 різних адрес. З цієї та інших причин імперативні мови програмування зазвичай пропонують програмістам поняття змінних, що надає їм можливість назвати місця пам'яті більш абстрактно.

1.4.2. Декларація та присвоєння. Якщо мова програмування не забезпечує засоби для неявного декларування змінних (як у випадку з C), змінні повинні бути оголошені перед їх використанням. При декларація змінних зазвичай вказують тип інформації, яка буде зберігатися у кожній змінній. Тип визначає, скільки місця в пам'яті потрібно резервувати для змінної та як будуть інтерпретуватися дані в цій області пам'яті. Наприклад, в мові програмування C змінні можна оголосити так як зображено на рис. 1.4.

```
int Age;
char name[] = "Ivan";
```

Рис. 1.4. Оголошення змінних у C

Тут вік оголошено цілим числом, тоді як ім'я - це рядок. У більшості мов заявленим змінним можна надати початкове значення, як показано у наведеному вище прикладі, при цьому змінна name ініціалізується з рядком "Ivan".

Іноді зручно називати значення, які можуть використовуватися, але не змінюватися під час виконання програми. Заявивши, що воно буде мати постійне значення, дозволяє компілятору створити більш ефективний код. Такі змінні називаються константами, варто зауважити, що константи мають сенс лише при ініціалізації з значенням як показано на рис. 1.5.

```
Int const DrivingAge = 18;
Float const pi = 3.142f;
```

Рис. 1.5. Оголошення констант у мові С

Взагалі вважається хорошою практикою програмування давати змінні змістовні імена, які можуть бути зрозумілі широкій аудиторії читачів. Зокрема, може бути важко прочитати код іншої людини, якщо ім'я DA було використано у наведеній вище декларації замість DrivingAge. Наявність зрозумілих імен змінних допомагає зробити код більш читабельним.

У більшості мов програмування оголошені змінні починаються або зі стандартного початкового значення наприклад, 0 для цілих чисел, або можуть мати будь-яке довільне значення можливо, залишене попередніми програмами. Для збереження значень змінних та для зміни вмісту змінної загалом використовується оператор присвоєння.

Присвоєння є основним засобом оновлення стану комп'ютерів. Оскільки імперативне програмування стосується побічних ефектів у пам’яті, присвоєння є центральними для цих мов. Наступні три команди С на рис. 1.6 ілюструють різні присвоєння, які можна зробити змінній Age, у випадку якщо вона була декларована вище.

```
Age = 25;
Age = DrivingAge - 1;
Age = Age + 1;
```

Рис. 1.6. Присвоєння у мові С

У лівій частині присвоєння задається змінна, значення якої буде оновлено, в той час як у правій задається вираз, який буде оцінено до того, як змінній буде присвоєно нове значення. У першому твердженні значення 25 просто зберігається у змінній. У другому твердженні спочатку береться вміст DrivingAge, а потім від нього віднімається 1. Нарешті, результат зберігається у Age. У третьому рядку спочатку береться вміст Age, потім додається 1 до цього вмісту, і він остаточно записується назад у Age. Це означає, що якщо значення Age до третього твердження було 42, воно буде 43.

1.4.3. Структури контролю. Як зазначалося раніше, програма - це послідовність інструкцій. Порядок виконання за замовчуванням для послідовності команд, відомий як послідовне виконання [8]. Для ілюстрації, якщо програма повністю складається з команд призначення та введення/виведення, комп'ютер просто виконає ці команди по порядку. Однак часто трапляється так, що більше контролю накладається на порядок виконання різних послідовностей команд. Імперативні мови програмування забезпечують різноманітність того, що називають структурами управління, щоб програмісти могли писати програми, де порядок виконання різних інструкцій може залежати від стану машини.

1.4.4. Умови. Програмам часто доводиться виконувати щось в залежності від стану. Ось як стан машини вплине на дії, які виконуватиме програма. Основна конструкція для цього - це твердження if, else if, else. У C це твердження має загальну форму як на рис. 1.7.

```
if (умова) {вітка істинності}
else {вітка хибності}
```

Рис. 1.7. Умова if у мові C

Наприклад, умова С читає ціле число і друкує парне або непарне в залежності від паритету числа зображена на рис. 1.8.

```
if (number % 2 == 0) { printf("%d is even.", number); }
else { printf("%d is odd.", number); }
```

Рис. 1.8. Умова перевірки числа на парність у мові С

Ще одна умовна конструкція - інструкція перемикач. Замість того, щоб обмежувати себе двома альтернативами, інструкція перемикач дозволяє мати стільки альтернатив, скільки потрібно, приклад зображено на рис. 1.9.

```
switch (n)
{
case 0:
printf("Ви ввели нуль.");
break;
case 1:
case 4:
…
case 9:
printf("n є повним квадратом.");
break;
case 2:
printf("n = 2.");
}
```

Рис. 1.9. Умова перемикач у мові С

Коли виконується інструкція перемикач, вираз оцінюється та порівнюється зі списком шаблонів. Дія, пов'язана з першим успішно узгодженим шаблоном, виконується. Значення зазвичай також повинні бути неперервними, тобто вираз може не відповідати більш ніж одному значенню.

Варто зауважте, що інструкція перемикач взагалі не може бути використаний для заміни вкладеної послідовності операторів if, else if, else.

Інструкція перемикач може відповідати значенням лише простому шаблону, а не колекції умов.

1.4.5. Ітерації. Повторення одного і того ж коду не обов'язково має однаковий ефект, оскільки початковий стан машини може бути різним. Багато різних функцій можна виконати, повторно виконуючи фрагмент коду, поки певна умова не буде виконана.

Найбільш поширеною конструкцією ітерації є цикл while. У С вона має форму, яка зображена на рис. 1.10.

```
while (умова){
тіло циклу
} кінець циклу
```

Рис. 1.10. Конструкція циклу у мові С

Ця структура управління повторює тіло циклу до тих пір, поки умова дорівнює істинності. Умова перевіряється, коли оператори while починають виконуватись та після завершення кожного проходу через тіло циклу [9]. Якщо умова, вказана вище в умові, ніколи не буде хибною, програма не припиняється, це часто описується як "програма переходить у нескінченний цикл". Випадкові неприпинення можуть бути серйозною проблемою в функціях у критичних для безпеки програмах.

Наступна програма С обчислює найменше невід'ємне ціле число x, яке задовольняє нерівність $x \cdot x > x + 100$, зображена на рис. 1.11.

```
while (Num * Num <= Num + 100)
{ Num = Num + 1;}
```

Рис. 1.11. Обчислення найменшого невід'ємного цілого число на мові С

Важливим особливим випадком циклів у той час, коли потрібно повторити послідовність команд певну кількість разів, це відбувається досить часто, щоб

гарантувати наявність окремої конструкції, яка називається for. Наприклад, код на рис. 1.12 обчислює $5 \cdot 6$.

```
for (int i = 0; i <= 6; i = i + 1)
{ Num = Num + 5;}
```

Рис. 1.12. Обчислення $5 \cdot 6$ на мові С

1.4.6. Процедури та функції. Функції та процедури - це основні складові імперативних програм. Це невеликі частинки коду, які використовуються для виконання певного завдання, і вони використовуються з двох основних причин. Перша причина полягає в тому, що їх можна використовувати, щоб уникнути повторення коду всередині програми. Якщо в програмі є операції, які виконуються в різних її частинах програми, то є сенс видалити повторний код і створити окрему функцію або процедуру, яку можна викликати замість цього. Це не тільки зменшить розмір програми, але і полегшить підтримку коду [7]. Це додає програмному коду модульності. Цей модульний підхід також полегшує абстрагування даних, шляхом централізації функцій зберігання даних.

Друга причина уважного використання функцій та процедур - допомогти визначити логічну структуру програми, розбивши її на ряд менших модулів із очевидними цілями. Залежно від мови програмування, як використовується, також можна скласти бібліотеку функцій та процедур та імпортувати їх для використання в інших програмах.

Функції та процедури дуже схожі, в деяких мовах програмування є лише функції, а процедури розглядаються як особливий випадок функції, так само як квадрат - це особливий вид прямокутника. Взагалі кажучи, функції повертають значення коли закінчують виконувати код всередині них, тоді як процедури ні, процедура є лише функцією, яка не повертає значення.

Під поверненням значення, мається на увазі, що функція створює певні результати, які передаються назад до змінної до якої було використане

присвоєння виклику функції. Підпрограма, яка обчислює квадрат числа, була б функцією, тому що результат обчислення потрібно передавати назад.

Змінні, задекларовані в межах функцій або процедур, називаються локальними - тобто вони можуть використовуватися лише в межах цієї функції або інших функцій, викликаних цією функцією. Це називається областю змінних.

Локальні змінні знищуються, коли функція або процедура закінчує своє виконанням, і їх значення втрачаються. Наступного разу, коли ця функція або процедура буде викликана знову, змінні відтворюються. Якщо потрібно, щоб локальні змінні зберігали свої значення, коли функція чи процедура викликається знову, потрібно оголосити їх статичними.

Для деяких функцій знадобляться аргументи - значення, які передаються функції, на яких буде грунтуватися дія, яку буде виконувати функція.

1.5. Об’єктно-орієнтована парадигма програмування

У цьому вступному розділі представлено огляд деяких основних понять об'єктно-орієнтованого програмування. Ця глава передбачає лише деяке ознайомлення з основами об'єктно-орієнтованим програмуванням.

1.5.1. Об’єкти, поля та методи.

Об'єкти - основні частинки в об'єктно-орієнтованій системі. Вони можуть представляти людину, місце, банківський рахунок, таблицю даних або будь-який предмет, який чи яким програма має обробляє. Вони також можуть представляти визначені користувачем дані, такі як вектори, час та списки. Проблема програмування аналізується з точки зору об'єктів та характеру зв'язку між ними. Об'єкти програми слід вибирати таким чином, щоб вони відповідали об'єктам реального світу. Об'єкти займають місце в пам’яті та мають пов’язану з ним адресу, як запис у Паскалі, або структура в C.

Коли програма виконується, об'єкти взаємодіють, надсилаючи повідомлення один одному. Наприклад, якщо "клієнт" і "рахунок" є двома

об'єктами в програмі, то об'єкт "клієнт" може надіслати повідомлення об'єкту "рахунок", який запитує баланс рахунку. Кожен об'єкт містить дані та код для обробки даних. Об'єкти можуть взаємодіяти, не знаючи деталей даних або коду один одного. Достатньо знати тип повідомлення який приймає об’єкт та тип відповіді, що повертається об'єктами [6].

1.5.2. Інкапсуляція. Об'єктно-орієнтоване програмування визначається двома основними ознаками: інкапсуляцією та успадкуванням. Оскільки об’єкт є постачальником послуг, члени об'єкта не є сукупністю споріднених незалежних членів. Швидше за все, члени об'єкта діляться відповідальністю за надання послуги, яку об'єкт створений надавати, тобто вони взаємно залежать один від одного в наданні послуги. Ця взаємна залежність особливо стосується "активного" компонента об'єкта, тобто його методів. В об'єктно-орієнтованому програмуванні методи об'єкта часто викликають один одного, тобто об'єкт може рекурсивно викликати власні методи. Поля об’єкта записують "стан" об'єкта. Методи об’єкта можуть отримати доступ до цих полів для отримання цієї інформації.

Об'єкт інкапсулює свої дані, оскільки об'єкт поєднує свої поля з методами, які можуть отримувати доступ до цих полів та керувати ними. Оскільки поля - це члени, вбудовані всередину об'єкта, методи об'єкта можуть отримати доступ до полів об'єкта, тобто даних, які об'єкт інкапсулює, так само, як вони можуть викликати інші методи об'єкта через спеціальну змінну this або self. Інкапсуляція даних (полів) за допомогою функцій (методів), які їх обробляють, була головною мотивацією розвитку ООП.

1.5.3. Класи. Клас - це синтаксична конструкція програмування яка використовується для створення об'єктів з певною поведінкою. Клас - це шаблон, з якого створюються об'єкти, що мають однакову поведінку. Об'єкти, створені за допомогою певного класу, називаються екземплярами цього класу.

Об'єкти містять дані та код для маніпулювання цими даними. Весь набір даних і код об'єкта можна зробити визначеним користувачем типом даних за допомогою класу. Фактично об'єкти є змінними класу. Після визначення класу ми можемо створити будь-яку кількість об'єктів, що належать до цього класу. Кожен об'єкт асоціюється з даними класу, за допомогою якого вони створені. Таким чином, клас - це сукупність об'єктів подібних типів. Наприклад, манго, яблуко та помаранчеві члени класу фруктів. Класи визначаються користувачем, і ведуть себе як вбудовані типи мови програмування. Синтаксис, який використовується для створення об'єкта, не відрізняється від синтаксису, який використовується для створення структури в С.

Клас має ім'я, яке називається ім'ям класу. Ім'я класу, як правило, асоціюється з поведінковою класу, тобто із специфікацією (формальною чи неофіційною) послуги, що надається екземплярами класу. Асоціація назв класів з їх поведінкою зазвичай використовується розробниками, оскільки це дає можливість розробникам розробляти програмне забезпечення на основі поведінки об'єктів у своєму програмному забезпеченні.

Клас у сучасному об'єктно-орієнтованому програмуванні може мати спеціальний "мета-метод", який називається конструктором. Конструктор класу використовується для створення екземплярів класу. Зазвичай у конструктора є код, який може ініціалізувати поля об'єкта. Спеціальні методи класу, конструктори зазвичай мають те саме ім'я, що і назва класу, екземпляр якого вони конструюють. Тобто конструктор це функція яка виконається як тільки буде створений екземпляр даного класу. Деструктор – теж спеціальний "мета-метод" класу який виконується коли об’єкт знищується.

1.5.4. Успадкування. Успадкування - це концепція яка вказує на повторне використання коду, не повторюючи і не переписуючи код. Вона являє собою створення нових класів із існуючих класів, і ці нові класи називаються похідними класами або підкласами.

Існуючий клас називається базовим класом або батьківським класом або суперкласом. Похідний клас повторно використовує поля та методи базового класу, крім того, він може додавати та змінювати їх [10]. Отже програмісти можуть уникати переписування та тестування коду, який вже існує. Метою успадкування є підтримка уточнення класів у похідних класах або підкласах. Є багато переваг використання концепції успадкування в мові програмування. Безпосереднє моделювання таких ієрархій полегшує розуміння концептуальної структури програм, успадкування підтримує ті загальні властивості класів, які факторизовані, що описуються один раз і використовуються при необхідності. Це призводить до модульності та полегшує складання та підтримку складних програм.

Ієрархії спадкування підтримують методику, коли найбільш загальні класи містять загальні властивості різних класів. Ці класи спочатку розробляються та перевіряються, а потім розробляються спеціалізовані класи, додаючи більше деталей до існуючих класів. Успадкування реалізує відносини "Is-A". У об’єктно-орієнтованому програмуванні відносини Is-A означають, що один об’єкт є типом іншого. Наприклад, студент - це людина, автомобіль - транспортний засіб тощо.

Клас успадковує декларації змінних, а також методи свого базового класу. Додаючи нові змінні екземпляра та нові методи, а також переосмисливши методи базового класу, атрибути похідного класу можуть бути переглянуті. Похідний клас не може отримати доступ до приватних членів свого базового класу; по-іншому ця ситуація буде суперечити концепції інкапсуляції. Похідний клас має доступ до загальнодоступних та захищених членів свого базового класу. Вони є спадковими та видимими для користувачів. Існують різні типи успадкування, які залежать від мови програмування:

– Одиночне успадкування: похідний клас, який успадковує властивості та поведінку від одного базового класу, називається одинарним успадкуванням.

– Багаторівнева спадщина: Якщо клас походить з іншого похідного класу, він називається багаторівневим успадкуванням.

– Високо-ієрархічне спадкування: Якщо з базового класу походить більше одного класу, це називається ієрархічним успадкуванням.

– Гібридне успадкування: це поєднана форма одинарного та множинного успадкування.

– Множинне спадкування: Якщо клас походить з більш ніж одного класу, він називається множинним успадкуванням.

1.5.5. Поліморфізм. Словникове визначення поліморфізму це властивість мати багато форм. В комп’ютерних науках під поліморфізмом розуміється здатність мови програмування описувати об'єкти по-різному на основі класу або типу даних.

Поліморфізм дозволяє інтеракцію з об’єктом, навіть якщо невідомо, що саме є об'єктом. Завдяки поліморфізму розмір програм для програмування можна зменшити. Крім того, зрозуміти ці програми легше, ніж інші. Якщо поліморфізма не існує, програмістам доводиться перевіряти об'єкти по черзі, щоб визначити, який тип і метод викликаються відповідно до типу об'єкта. Поліморфізм активізується в таких ситуаціях, що звільняє дизайнера від цих незручностей і дозволяє гнучкість.

Щоб краще зрозуміти поняття поліморфізму, потрібно добре осмислити успадкування. Поліморфізм тісно пов'язаний з концепцією успадкування в об’єктно-орієнтованих мовах. Існує просте правило, яке називається " is-a", щоб знати, чи є успадкування правильним дизайном для даних користувача. Це правило зазначає, що кожен об'єкт підкласу є об'єктом надкласу. Для пояснення взаємозв'язку спадщини та поліморфізму наведено приклад на рис. 1.13.

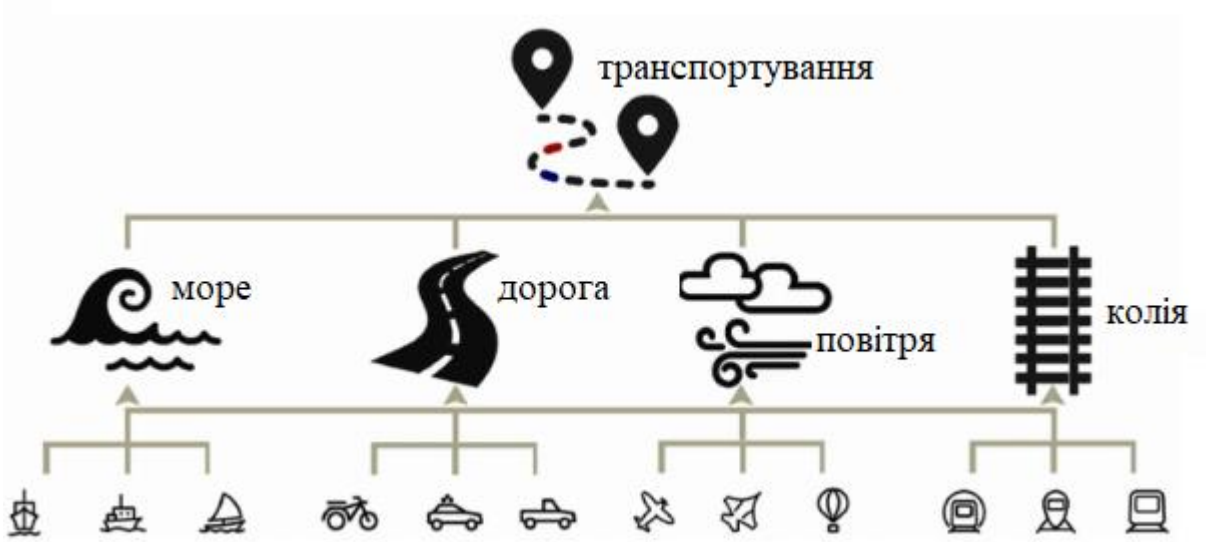

Рис. 1.13. Приклад успадкування

На малюнку представлений клас транспорту з підкласами, що включає відповідні режими (море, дорога, повітря, залізниця) та транспортні засоби. Наприклад, всі дорожні транспортні засоби - тип транспортного засобу. Таким чином, має сенс для класу дорожніх транспортних засобів бути підкласом класу транспортних засобів. Звичайно, навпаки не вірно, оскільки не кожен транспортний засіб - це дорожній транспортний засіб [7]. Всі автомобілі та велосипеди - дорожні транспортні засоби, тому їх можна згрупувати як підклас дорожнього класу. Спортивний і звичайний автомобіль, які не показані на малюнку, позначені підкласами автомобільного класу, як показано нижче на прикладі.

Найбільш загальне поняття успадкування можна пояснити таким чином: Спільні риси гоночного та звичайного автомобіля:

– це дорожні транспортні засоби;
– вони мають двигун;
– вони призначені для перевезення людей;
– вони споживають паливо;
– їм потрібно Водій;
– загальні риси автомобіля та велосипеда;

– дорожній транспортний засіб;
– вони призначені для перевезення людей;
– їм потрібен водій;
– загальні риси дорожнього та морського транспортних засобів;
– вони призначені для перевезення людей;
– їм потрібен водій.

Як видно з прикладу, таких інформацій, як "перевозити людей" та "потрібен водій", повторюється більше, ніж один раз. Кожну категорію інформації не потрібно зберігати в цих класах, оскільки спадкування надає їм це безпосередньо. Якщо загальні ознаки визначені в класі транспортного засобу, всі підкласи можуть брати цю інформацію лише з одного класу. Таким чином, якщо необхідно оновити систему, може бути достатньо лише однієї зміни, пов’язаної з концепцією в класі транспортного засобу. І це називається поліморфізмом. Поліморфізм також пов'язаний з "перевантаженням" і "заміщенням".

Перевантаження - це метод поліморфізму, який має однакову назву з різними параметрами. Перевантаження - це функція, яка дає можливість класу володіти двома або більше методами, що мають однакову назву. Якщо тип методу повернення не є однаковим, відбудеться помилка.

Заміщення - це метод поліморфізму заміни методу що задається в базовому класі, в підкласі. Заміщений метод може бути доданий, перезаписавши, метод успадкований від базового класу. Таким чином, він забезпечує використання методу успадкованого класу. У цьому випадку програмне забезпечення дозволяє гнучкість, яка може зробити іншу роботу, використовуючи один і той же метод.

1.5.6. Абстракція. Слово абстрактний означає відмежований від будь-якого конкретного екземпляра. Абстракція полягає у розробці моделей з точки зору інтерфейсу та функціональності замість деталей щодо реалізації. Таким чином, це пов'язано з інкапсуляцією та приховуванням даних. Абстракція застосовується до моделі, розглядаючи процес ідентифікації об'єкта. Він

використовується для зменшення складності розробки та реалізації, що фокусується на сенсі поведінки, щоб уникнути конкретизації. Завдяки абстракції внутрішні класи класів захищаються від помилок на рівні користувача, що порушує стан об'єктів. Абстрактний клас є батьківським класом, який дозволяє успадкуватися від нього, містить абстрактні члени. Ці члени у ньому лише декларуються, а не реалізуються. Реалізація абстрактних членів здійснюється в межах похідного класу. Інший тип члена - це віртуальний член. На відміну від абстрактного члена, віртуальні члени реалізовані в батьківському класі. Для оголошення абстрактного класу використовується ключове слово "abstract". Функції та властивості абстрактних членів також оголошуються за допомогою цього ключового слова.

1.5.7. Асоціація, агрегація і композиція. Коли ми думаємо про об'єктно-орієнтовану природу програмування, ми завжди думаємо про об'єкти, та взаємозв'язок між ними. Об'єкти пов'язані між собою та взаємодіють між собою за допомогою методів. Іншими словами, об’єкт одного класу може використовувати послуги/методи, що надаються об'єктом іншого класу. Цей вид відносин називається асоціацією.

Агрегація та композиція є підмножинами асоціації, тобто вони є конкретними випадками асоціації як показано на рис. 1.14.

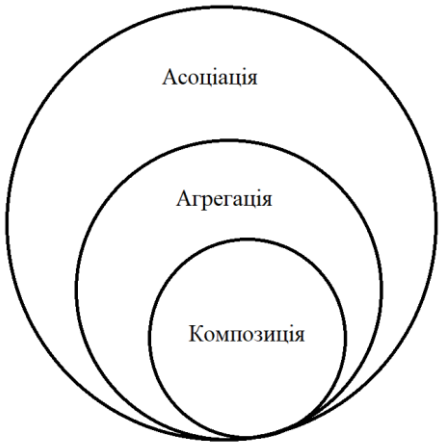

Рис. 1.14. Асоціація та її випадки

Як в агрегації, так і в композиції об'єкт одного класу "володіє" об'єктом іншого класу. Але є тонка різниця, у композиції об'єкт класу, який належить об'єкту класу, що належить його володіючому класу, не може існувати самостійно. Він завжди буде існувати як частина об'єкта який ним володіє, але в агрегації, залежний об'єкт є самостійним і може існувати, навіть якщо об’єкт класу якому він належить вже не існує.

У композиції, якщо володіючий об’єкт видалений збирачем сміття, збирається і об'єкт що належить йому, що в агрегації не відбувається.

Підсумовуючи асоціація - це дуже загальний термін, який використовується для представлення випадку, коли клас використовує функції, що знаходяться в іншому класі. Ми говоримо, що це композиція, якщо одному об’єкту батьківського класу належить інший об'єкт дочірнього класу, і цей дочірній об’єкт класу не може значимо існувати без об'єкта батьківського класу. Якщо це можливо, це називається агрегацією [9].

1.5.8. Переваги та недоліки об’єктно-орієнтованої парадигми. Об'єктно-орієнтоване програмування пропонує ряд переваг як для дизайнера програми, так і для користувача. Об’єктна орієнтація сприяє вирішенню багатьох проблем, пов'язаних з розробкою та якістю програмних продуктів. Ця парадигма обіцяє більшу продуктивність програміста, кращу якість програмного забезпечення та менші витрати на обслуговування. Основними перевагами є:

– За рахунок спадкування ми можемо усунути зайвий код, розширивши використання уже написаного.

– Класи.

– Можна будувати програми зі стандартних робочих модулів, які спілкуються один з одним, замість того, щоб починати писати код з нуля. Це призводить до економії часу розробки та підвищення продуктивності.

– Принцип приховання даних допомагає програмісту створити захищену програму, яка не можна вторгнутись кодом в інші частини програм.

– Можливе спільне існування кількох примірників об'єкта без жодного конфлікту.

– Робота в проекті легко розділяється на основі об'єктів.

– Підхід до дизайну, орієнтованого на даних, дозволяє отримати більш детальну інформацію про модель, що реалізується.

– Об'єктно-орієнтовану систему можна легко модернізувати від малої до великої системи.

– Техніка передачі повідомлень для зв'язку між об'єктами значно спрощує описи інтерфейсу із зовнішніми системами.

– Складністю програмного забезпечення можна легко керувати.

Хоч і можливо включити всі ці функції в об'єктно-орієнтовану систему, їх значення залежить від типу проекту та уподобань програміста. Існує ряд питань, які необхідно вирішити, щоб отримати деякі з перерахованих вище переваг. Наприклад, бібліотеки об'єктів повинні бути доступні для повторного використання [11].

Недоліком об’єктно-орієнтованого програмування можна вважати складний процес навчання: мисельний процес, що бере участь в об'єктно-орієнтованому програмуванні, для деяких людей може бути неприродним, і для його звикання може знадобитися час. Інколи складно створювати програми на основі взаємодії об’єктів. Деякі ключові методи програмування, такі як успадкування та поліморфізм, можуть бути складними для розуміння на початку [12].

Ще один недоліком об’єктно-орієнтованого програмування, є те що ця парадигма не підходить для всіх типів проблем. Є проблеми, які добре піддаються функційному стилю програмування, логічному стилю програмування або стилю програмування, заснованому на процедурах, і застосування об'єктно-орієнтованого програмування в таких ситуаціях не призведе до ефективних програм.

1.6. Функційна парадигма програмування

Функційне програмування являє собою парадигму, в корені відмінну від інших парадигм програмування. Дана парадигма програмування вирішує проблеми шляхом перенесення даних з функції у функцію, що призводить до серії перетворень. Функційне програмування визнає, що джерелом складності є стан, оскільки призначення змінних означає, що розробник програми і сама програма не можуть бути впевнені, у якому стані знаходиться змінна, коли вона використовується [13]. У чисто функційному програмуванні змінні взагалі не використовуються, і все робиться шляхом передачі значень між функціями.

Мови функційного програмування поділяються на чисті функційні мови, які підтримують лише функцій ні парадигми (Haskell, Mercury та ін.), та на нечисті функційні мови, які підтримують функційні та імперативні парадигми(LISP, Python, Erlang, Kotlin, Mathematica, Java, C++, C#, Clojure, Ruby, Perl, Elixir та ін.).

Програми на традиційних мовах програмування, таких як Сі, Паскаль і т.п. складаються їх послідовності модифікацій значень деякого набору змінних, який називається станом. Якщо не розглядати операції введення-виведення, а також не враховувати того факту, що програма може працювати безперервно (тобто без зупинок, як у випадку серверних програм), можна зробити наступну абстракцію. До початку виконання програми, стан має деяке початкове значення σ_0, в якому представлені вхідні значення програми. Після завершення програми стан має нове значення σ^1, що включає в себе те, що можна розглядати як «результат» роботи програми. Під час виконання кожна команда змінює стан; отже, стан проходить через деяку кінцеву послідовність значень:

$$\sigma = \sigma_0 \rightarrow \sigma_1 \rightarrow \cdots \rightarrow \sigma_n = \sigma^1 \qquad (1.1)$$

Стан модифікується за допомогою команд присвоювання, що записуються у вигляді v = E або v: = E, де v - змінна, а E - деякий вираз. Ці команди йдуть одна за одною; оператори, такі як if і while, дозволяють змінити порядок виконання цих команд в залежності від поточного значення стану. Такий стиль програмування називають імперативним або процедурним [14].

Функційна програма являє собою певний вираз(в математичному сенсі); виконання програми означає обчислення цього виразу. Вважаючи що результат роботи імперативної програми повністю і однозначно визначений її вхідними данними, можна сказати, що фінальне стан (або будь-який проміжний) являє собою деяку функцію (в математичному сенсі) від початкового стану, тобто:

$$\sigma^1 = f(\sigma) \tag{1.2}$$

У функційному програмуванні використовується саме ця точка зору: програма являє собою вираз, відповідний функції f. Функційні мови програмування підтримують побудову таких виразів.

При порівнянні функційного і імперативного підходу до програмування можна помітити такі властивості функційних програм:

– Функційні програми не використовують змінні в тому сенсі, в якому це слово вживається в імперативний програмуванні. Зокрема в функційних програмах не використовується оператор присвоювання.

– Як наслідок з попереднього пункту, в функційних програмах немає циклів.

– Виконання послідовності команд у функційній програмі безглуздо, оскільки одна команда не може вплинути на виконання наступної.

– Функційні програми використовують функції набагато більш хитромудрими способами. Функції можна передавати в інші функції в якості

аргументів і повертати в якості результату, і навіть в загальному випадку проводити обчислення, результатом якого буде функція.

– Замість циклів функційні програми широко використовують рекурсивні функції.

Перш за все, імперативний стиль в програмуванні не є жорстко заданою необхідністю. Багато характеристики імперативних мов програмування є результатом абстрагування від низькорівневих деталей реалізації комп'ютера, від машинних кодів до мов асемблера, а потім до мов типу Фортрана і т.д. Однак немає причин вважати, що такі мови відображають найбільш природний для людини спосіб повідомити машині про свої наміри [15]. Можливо, більш правильний підхід, при якому мови програмування народжуються як абстрактні системи для запису алгоритмів, а потім відбувається їх переклад на імперативну мову комп'ютера.

У порівнянні з об’єктно орієнтованими мовами, функційне програмування та об'єктно-орієнтоване програмування - це різні концепції мови програмування. Мета обох парадигм забезпечити код без багів, який може бути легко зрозумілим та читабельним, та швидким у розробці.

Функційне програмування та об'єктно-орієнтоване програмування використовують різні метод зберігання та маніпулювання даними. У чисто функційному програмуванні дані не можуть зберігатися в об'єктах, а їх можна трансформувати лише шляхом створення функцій. В об'єктно-орієнтованому програмуванні дані зберігаються в об'єктах [16]. Об'єктно-орієнтоване програмування широко використовується програмістами і також успішно. У табл. 1.1 винесене порівняння об’єктно орієнтованої і функційної парадигм та описані елементи кожної парадигми.

Таблиця 1.1

Порівняння функційної та об’єктно-орієнтованої парадигм

Вид порівняння	Функційне програмування	Об’єктно-орієнтоване програмування
Дефініція	Наголошує на виконанні функцій	Базується на концепції об’єктів
Дані	Використовує незмінні дані	Використовує змінні дані
Модель	Дотримується декларативної моделі програмування	Дотримується імперативної моделі програмування
Підтримка	Паралельне програмування підтримується у функційній парадигмі	Паралельне програмування не підтримується у об’єктно-орієнтованій парадигмі
Виконання	Оператори можуть виконуватися в будь-якому порядку	Оператори повинні виконуватися в певному заданому порядку
Ітерації	Для ітерації даних використовується рекурсія	Для ітерації даних використовуються цикли
Основний елемент	Основним елементом в функційному програмуванні є функції	Основними елементами в об’єктно-орієнтованому програмуванні є об’єкти, методи та змінні
Використання	Функційне програмування використовується тоді, коли є кілька речей з більшою кількістю операцій.	Об'єктно-орієнтоване програмування застосовується тоді, коли є багато речей, з якими проводиться менше операцій

1.7.Висновки до розділу

У даному розділі було чітко визначено різницю між мікрокомп’ютерами, мікропроцесорами та мікроконтролерами, описано різницю між ними. Описано

сучасні методи які використовуються для програмування мікроконтролерів та описані їхні обмеження, на які потрібно звертати при програмуванні.

Описане та проаналізоване поняття парадигми програмування, для чого воно існує і де застосовується. Проаналізовані найпопулярніші парадигми програмування, які використовуються у програмуванні мікроконтролерів, а саме імперативна та об’єктно орієнтована та описані основні елементи кожної парадигми які характеризують їх, з прикладами та описаними недоліками та перевагами кожної з них.

РОЗДІЛ 2
МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ ФУНКЦІЙНОЇ ПАРАДИГМИ

2.1. Компаративний аналіз функційної парадигми програмування

Функційний підхід має ряд переваг перед імперативним. Перш за все, функційні програми більш безпосередньо відповідають математичним об'єктам, і отже, дозволяють проводити строгі міркування. Встановити значення імперативній програми, тобто тієї функції, обчислення якої вона реалізує, може виявитися досить важко, а значення функційної програми, навпаки може бути виведено практично безпосередньо. Наприклад, розглянемо наступну програму на мові Haskell на рис. 2.1.

```
factorial n = if n == 0 then 1 else n * factorial (n - 1)
```

Рис. 2.1. Приклад функції на Haskell

```
int f (int n)
{
    int x = 1;
    while (n > 0)
    {
        x = x * n;
        n = n - 1;
    }
    return x;
}
```

Рис. 2.2. Аналог функції Haskell на C

Практично відразу видно, що ця програма відповідає наступній часткової функції:

$$f(n) = \begin{cases} n! & n \geq 0 \\ \perp & n < 0 \end{cases}$$

Символ ⊥ означає невизначеність функції, оскільки при негативних значеннях аргументу програма не закінчується.

Однак для програми на мові С, яка зображена на рис. 2.2, ця відповідність не очевидна.

Слід також зауважити щодо вживання терміна «функція» в таких мовах як Сі, Java і т.п. В математичному сенсі «функції» мови Сі не є функціями, оскільки:

– Їх значення може залежати не тільки від аргументів.

– Результатом їх виконання можуть бути різноманітні побічні ефекти наприклад, зміна значень глобальних змінних.

– Два виклики однієї і тієї ж функції з одними і тими ж аргументами можуть привести до різних результатів.

Функції в функційних програмах дійсно є функціями в тому сенсі, в якому це розуміється в математиці. Відповідно, ті зауваження, які були зроблені вище, до них не можна застосувати. З цього випливає, що обчислення будь-якого виразу не може мати ніяких побічних ефектів, і значить, порядок обчислення його подвиразів не впливає на результат. Таким чином, функційні програми легко піддаються розпаралеленню, оскільки окремі компоненти виразів можуть обчислюватися одночасно [13].

2.1.1. Основи лямбда числень. Подібно до того, як теорія машин Тюринга є основою імперативних мов програмування, лямбда-числення служить базисом і математичним «фундаментом», на якому засновані всі функйні мови програмування.

В даний час лямбда-числення є основною з таких формалізацій, застосовуваної в дослідженнях пов'язаних з мовами програмування. Пов'язано це, ймовірно, з наступними факторами:

– Це єдина формалізація, яка, хоча і з деякими незручностями, дійсно може бути безпосередньо використана для написання програм.

– Лямбда-числення дає просту і природну модель для таких важливих понять, як рекурсія і вкладені середовища.

– Більшість конструкцій традиційних мов програмування може бути більш-менш безпосередньо відображено в конструкції лямбда-числення.

– Функційні мови є в основному зручною формою синтаксичного записи для конструкцій різних варіантів лямбда-числень. Деякі сучасні мови (Haskell, Clean) мають 100% відповідність своїй семантиці з семантикою конструкцій лямбда-числень.

В математиці, коли необхідно говорити про будь-яку функцію, прийнято давати цій функції деяке ім'я і згодом використовувати його, як, наприклад, в наступному твердженні:

Нехай $f: \mathbb{R} \to \mathbb{R}$ визначається наступним виразом:

$$f(x) = \begin{cases} 0, & x = 0 \\ x^2 \sin(\frac{1}{x^2}), & x \neq 0 \end{cases} \qquad (2.1)$$

Тоді $f^1(x)$ не інтегрована на інтервалі [0, 1].

Багато мови програмування також допускають визначення функцій тільки з привласненням їм деяких імен. Наприклад, в мові Сі функція завжди повинна мати ім'я. Це здається природним, проте оскільки в функційному програмуванні функції використовуються повсюдно, такий підхід може призвести до серйозних ускладнень.

Лямбда-нотація дозволяє визначати функції з тією ж легкістю, що і інші математичні об'єкти. Лямбда-виразом називається конструкція виду $\lambda x.\,E$, де E - деякий вираз, можливо, використовує змінну x. Наприклад. $\lambda x.\,x^2$ являє собою функцію, що зводять свій аргумент в квадрат.

Використання лямбда-нотації дозволяє чітко розділити випадки, коли під виразом виду $f(x)$ розуміється сама функція f і її значення в точці x. Крім того, лямбда-нотація дозволяє формалізувати практично всі види математичної нотації. Якщо почати з констант і змінних і будувати висловлювання тільки з допомогою лямбда-виразів і застосувань функції до аргументів, то є можливість уявити дуже складні математичні вирази [16].

Існує операція каррірування, що дозволяє записати функції в звичайній лямбда-нотації. Ідея полягає в тому, щоб використовувати вирази виду $\lambda xy.x + y$. Такий вираз можна розглядати як функцію $\mathbb{R} \to (\mathbb{R} \to \mathbb{R})$, тобто якщо його застосувати до одного аргументу, результатом буде функція, яка потім приймає інший аргумент. Таким чином:

$$(\lambda xy.x + y)1\ 2 = (\lambda y.1 + y)2 = 1 + 2 \qquad (2.2)$$

Змінні в лямбда-виразах можуть бути вільними і пов'язаними. У виразі виду $x^2 + x$ змінна x є вільною, а значення виразу залежить від значення змінної x і в загальному випадку її не можна перейменувати. Однак в таких виразах як $\sum_{i=1}^{n} i$ або $\int_o^x \sin y\, dy$ змінні i та y являються пов’язаними, тобто якщо замість i повсюди використовувати позначення j значення виразу не зміниться.

В будь-якому підвиразі змінна може бути вільною як у натуральному виразі під інтегралом, однак у всьому виразі вона пов'язана будь-якою операцією зв'язування змінної, такою як операція сумовування. Та частина виразу, яка знаходиться всередині операції зв'язування, називається областю видимості змінної.

В лямбда численні вирази $\lambda x.\mathrm{E}\,[x]$ і $\lambda y.\mathrm{E}\,[\mathrm{y}]$ вважаються еквівалентними та називаються α-еквівалентними, і процес перетворення між такими парами

називають α-перетворенням. Але необхідно накласти умову, що y не є вільною змінною в $\mathrm{E}\ [x]$.

2.1.2. Комбінатори. В теорії комбінаторів встановлено, що за допомогою кілька базових комбінаторів і змінних можна виразити будь-який терм без застосування операції лямбда-абстракції. Зокрема, замкнутий терм можна висловити тільки через ці базові комбінатори. Визначаються ці комбінатори наступним чином:

$$I = \lambda x.x$$
$$K = \lambda x\ y.x \qquad (2.3)$$
$$S = \lambda f\ g\ x.f\ x\ (g\ x)$$

I є функцією ідентичності, яка залишає свій аргумент незмінним. K служить для створення постійних (константних) функцій: застосувавши його до аргументу a, отримаємо функцію $\lambda x.a$, яка повертає a незалежно від переданого їй аргументу. Комбінатор S є розділяющим: він бере дві функції і аргумент і розділяє аргумент між функціями. Для будь-якого лямбда-терма t існує терм t', що не містить лямбда-абстракцій і складений з комбінаторів S, K, I і змінних, такий що $FV(t') = FV(t)\ u\ t' = t$. Це можна посилити, оскільки комбінатор I може бути виражений в термінах S і K. Дійсно, для будь-якого A виконується:

$$S\ K\ A\ x = Kx\ (Ax)$$
$$= (\lambda y\ x)\ (A\ x) \qquad (2.4)$$

$= x$

Застосовуючи η-конверсію, отримуємо, що $I = SKA$ для будь-якого A. З причин, які стануть зрозумілі надалі, використовується $A = K$. Таким чином, $I = SKK$, і в подальшому символ I можна виключати з виразів, складених з комбінаторів.

Лямбда-числення може розглядатися як проста функційна мова програмування, що становить ядро справжніх мов, таких як ML або Haskell. Тоді можна сказати, що наведена вище теорема показує, що лямбда-числення може бути в певному сенсі скомпільованою в машинний код комбінаторів [18]. Комбінатори дійсно використовуються як метод реалізації функційних мов не тільки на рівні програмного, а й на рівні апаратного забезпечення.

2.1.3. Рекурсивні функції. Можливість визначати рекурсивні функції - характерна особливість функційного програмування.

Ключовим моментом є існування так званих комбінаторів нерухомої точки. Замкнуте лямбда-терм *Y* називається комбінатором нерухомої точки, якщо для будь-якого лямбда-терма $\mathfrak{f}$ виконується: $\mathfrak{f}(Y\,\mathfrak{f}) = Y\,\mathfrak{f}$. Таким чином, комбінатор нерухомої точки за заданим терму $\mathfrak{f}$ повертає нерухому точку $\mathfrak{f}$, тобто терм *x*, такий що $\mathfrak{f}(x) = x$. Перший такий комбінатор, знайдений Каррі, зазвичай позначається як *Y*. Часто його називають «парадоксальним комбінатором». Він визначається наступним чином:

$$Y \triangleq \lambda\mathfrak{f}.\left(\lambda x.\mathfrak{f}\,(x\,x)\right)\,(\lambda x.\mathfrak{f}\,(x\,x)) \tag{2.5}$$

цей вислів справді визначає комбінатор нерухомої точки:

$$
\begin{aligned}
Y\,\mathrm{f} &= \left(\lambda \mathrm{f}.\left(\lambda x.\mathrm{f}\,(x\,x)\right)\left(\lambda x.\mathrm{f}\,(x x)\right)\right)\mathrm{f} = \\
&= \left(\lambda x.\mathrm{f}\,(x\,x)\right)\left(\lambda x.\mathrm{f}\,(x\,x)\right) = \\
&\mathrm{f}\left(\left(\lambda x.\mathrm{f}\,(x\,x)\right)\left(\lambda x.f\,(x\,x)\right)\right) = \\
&= \mathrm{f}(Y\,\mathrm{f})
\end{aligned}
\tag{2.6}
$$

З математичної точки зору все вірно, з обчислювального боку таке визначення викликає труднощі, оскільки вищенаведене міркування використовує лямбда-рівність, а не редукції. З цієї причини можна вважати за краще наступне визначення комбінатора нерухомої точки, що належить Тюрингу:

$$
T \triangleq \left(\lambda x\,y.y(x\,x\,y)\right)\left(\lambda x\,y.y\,(x\,x\,y)\right) \tag{2.7}
$$

Позначення комбінатора нерухомої точки залишається як і раніше *Y*. Для прикладу розглянуто функцію факторіала. Потрібно визначити функцію *fact*, таку, що

$$
fact(n) = if\ ISZERO\ n\ then\ 1\ else\ n * fact(PRE\ n)
$$

Спершу це перетворюється до наступного еквівалентного вигляду:

$$
fact = \lambda n.if\ ISZERO\ n\ then\ 1\ else\ n * fact(PRE\ n)
$$

Цей вислів, в свою чергу, еквівалентний

$$
fact = (\lambda f\ n.if\ ISZERO\ n\ then\ 1\ else\ n * f(PRE\ n))\ fact
$$

Звідси можна зробити висновок, що fact є нерухомою точкою деякої функції *F* такого вигляду:

$$
F = \lambda f\ n.if\ ISZERO\ n\ then\ 1\ else\ n * f(PRE\ n)
$$

Таким чином, fact = Y F.

Схожа техніка використовується для визначення взаємно рекурсивних функцій, тобто набору функцій, визначення яких взаємно залежать один від одного. визначення виду

$$\begin{aligned} \mathfrak{f}_1 &= F_1\, \mathfrak{f}_1 \ldots \mathfrak{f}_n \\ \mathfrak{f}_2 &= F_2\, \mathfrak{f}_1 \ldots \mathfrak{f}_n \\ \ldots &= \ldots \\ \mathfrak{f}_n &= F_n\, \mathfrak{f}_1 \ldots \mathfrak{f}_n \end{aligned} \tag{2.8}$$

можна перетворити, використовуючи кортежі, до однієї рівності:

$$(\mathfrak{f}_1, \mathfrak{f}_2, \ldots, \mathfrak{f}_n) = (F_1\, \mathfrak{f}_1 \ldots \mathfrak{f}_n, F_2\, \mathfrak{f}_1 \ldots \mathfrak{f}_n, \ldots, F_n\, \mathfrak{f}_1 \ldots \mathfrak{f}_n) \tag{2.9}$$

Тепер, якщо ми записувати $t = (\mathfrak{f}_1, \mathfrak{f}_2, \ldots, \mathfrak{f}_n)$, то кожна з функцій $\mathfrak{f}_i$ в правій частині рівності може бути представлена через відповідний селектор: $\mathfrak{f}_i = (t)_i$. Таким чином, рівняння можна записати в канонічному вигляді $t = F\ t$, що дає рішення $t = Y\ F$. Звідси, знову за допомогою селекторів, можна отримати окремі компоненти кортежу t.

2.1.4. Поліморфізм. Система типів по Каррі вже дає деяку форму поліморфізму, в тому сенсі, що терм може мати різні типи. Необхідно розрізняти схожі концепції поліморфізму і перегрузки. Обидва цих терміни означають, що вираз може мати кілька типів. Однак в разі поліморфізму всі типи зберігають деяку систематичну схожість один з одним і допустимі всі типи, які дотримуються деякого зразкоу. Наприклад, функція ідентичності може мати тип $\sigma \to \sigma, \tau \to \tau$ або $(\sigma \to \sigma) \to (\sigma \to \tau)$, але всі ці типи мають схожу

структуру. На противагу цьому, при перегрузці функція може мати різні типи, не пов'язані один з одним структурною подібністю. Також можлива ситуація коли функція просто визначається для деякого набору типів. Наприклад, функція складання може мати тип int → int → int або float → float → float, але не bool → bool → bool.

2.1.5. Відкладені обчислення. Теоретично нормальний порядок редукції виразів найкращий, оскільки якщо будь-яка стратегія завершується, завершиться і вона. Така стратегія відома і в традиційних мовах програмування в Алгол 60 її називають викликом за іменем, за такими ж правилами викликаються параметризрвані макроозначення в мові Сі [19]. Однак з практичної точки зору така стратегія має суттєві недоліки. Для прикладу дано вираз $(\lambda x.x + x + x)\ (10 + 5)$, Нормальна редукція зводить цей вислів до $(10 + 5) + (10 + 5) + (10 + 5)$.

Таким чином, необхідно тричі обчислювати значення одного і того ж вирази. На практиці це, зрозуміло, неприпустимо. Існують два основних рішення цієї проблеми і вони поділяють світ функційного програмування на два табори.

У першому підході відмовляються від нормальної стратегії і обчислюють аргументи функцій до того, як передати їх значення в функцію. Це - звичайна практика в таких мовах, як Сі, Паскаль і т. Д. Такий підхід називають передачею за значенням. При цьому обчислення деяких виразів, що завершується при нормальній стратегії, може привести до нескінченної послідовності редукцій. Однак на практиці цих випадків можна легко уникнути. Зазвичай така стратегія дозволяє отримувати досить ефективний машинний код. Також вона краща в разі гібридних мов, тобто функційних мов, що містять імперативні конструкції.Мови сімейства ML дотримуються такого порядку обчислень.

При іншому підході, що використовується в Haskell і деяких інших мовах, нормальна стратегія редукцій зберігається. Однак при цьому різні виникаючі підвирази поділяються і ніколи не обчислюються більш ніж один раз. У внутрішньому представленні вирази стають не деревами, а спрямованими

ациклічними графами. Така дисципліна виклику називається ледачою або викликом по необхідності, оскільки вирази обчислюються тільки тоді, коли їх значення дійсно необхідні для роботи програми.

Наприклад, в мові Haskell можна зробити наступне визначення:

```
bottom = bottom
```

Згідно з цим визначенням, обчислення виразу bottom ніколи не завершитися. Однак при такій функції:

```
const1 x = 1
```

Значення виразу const1 bottom дорівнює 1. Оскільки функція const1 не потребує значення свого аргументу, і він не обчислюється. Аналогічна ситуація і з наступним виразом:

```
const1 (1/0)
```

Значення цього виразу також дорівнює 1. Ділення на 0 не відбувається, оскільки аргумент функції ніколи не обчислюється.

Перевагою такого підходу є те, що він дозволяє добиватися більшої виразності при записі функцій. Платою за це є складність реалізації і деяке зниження ефективності. Замість того, щоб безпосередньо обчислити вираз, доводиться зберігати інформацію про те, як його обчислювати. Зрозуміло, якщо значення цього виразу не знадобилося в подальших обчисленнях, ми отримуємо деяку економію, однак в іншому випадку безпосереднє обчислення здається кращим. Оптимізуючі компілятори мови Haskell в багатьох випадках можуть самі виявити місця програми, в яких можна обійтися без відкладених обчислень.

Конструктори даних в Haskell також є функціями з відмінністю від «справжніх» функцій, яка полягає в тому, що їх можна використовувати в шаблонах при зіставленні зі зразком. У поєднанні з «ледачим» викликом це дозволяє визначати нескінченні структури даних. Наприклад, таке визначення задає нескінченний список одиниць:

```
ones = 1 : ones
```

Більш цікавим прикладом служить нескінченний список цілих чисел, починаючи з числа *n*:

```
numsFrom n = n : numsFrom (n + 1)
```

Термін «нескінченний» тут - не перебільшення. Визначення, подібні наведеним, дійсно задають потенційно нескінченні структури даних. З ними можна працювати так само, як і з кінцевими, наприклад, отримати (нескінченний) список квадратів натуральних чисел:

```
squares = map (^2) (numsFrom 1)
```

Зрозуміло, в реальності ми працюємо тільки з кінцевою частиною нашого списку, однак використання відкладених обчислень дозволяє відокремити генерацію нескінченної структури даних від виділення її кінцевої частини. Для цих прикладів кінцеву частину списку можна виділити, наприклад, за допомогою функції take, яка за заданою кількістю *n* і списку повертає перші *n* елементів цього списку. Так, значення виразу:

```
take 5 (numsFrom 1)
```

дорівнює [1,2,3,4,5], а результатом обчислення

```
take 5 squares
```

буде [1,4,9,16,25]

Представлений приклад дає зразок типової структури програми, що використовується в Haskell. Програма представляється у вигляді конструкції g (f input), де g і f - деякі функції. Вони виконуються разом строго синхронно. f запускається тільки тоді, коли g намагається прочитати деяке введення, і виконується рівно стільки, щоб надати дані, які намагається читати g. Після цього f призупиняється, і виконується g, до тих пір, поки знов не спробує прочитати наступну групу вхідних даних. Якщо g закінчується, не прочитавши весь вивід f, то f переривається. f може навіть бути нескінченною програмою, що створює нескінченний вивід, таквона буде зупинена, як тільки завершиться g. Це дозволяє відокремити умови завершення від тіла циклу, що є потужним засібом модуляризації.

Метод називається "ледачими обчисленнями" так як f виконується настільки рідко, наскільки це можливо. Він дозволяє здійснити модуляризацію програми як генератора, який створює велику кількість можливих відповідей, і селектора, який вибирає підходящі. Деякі інші системи дозволяють програмам виконуватися разом подібним способом, але тільки функційні мови використовують ледачі обчислення однорідно при кожному зверненні до функції, дозволяючи модуляризовати таким чином будь-яку частину програми. Ледачі обчислення, можливо, найбільш потужний інструмент для модулярзації в наборі функційного програміста.

2.1.6. Монади. Поняття монад є одним з найважливіших в мові Haskell. Однією з найпростіших монад є тип Maybe. Його визначення виглядає так:

```
data Maybe a = Nothing | Just a
```

Він використовується для повернення результату з функцій в разі, якщо цього результату може і не бути. Наприклад, функція *f* з сигнатурою

```
f :: a -> Maybe b
```

приймає значення типу a і повертає результат типу b (обгорнутий в конструктор Just), або може не обчислити значення і тоді, як сигнал про помилку, поверне значення Nothing.

Типовим прикладом такого роду функцій служать функції для здійснення запиту до бази даних. У разі, якщо дані, що задовольняють критеріям запиту, існують, їх слід повернути; в іншому випадку повертається Nothing.

Для прикладу розглянуто базу даних, що містить інформацію про адреси людей. Вона встановлює відповідність між повним ім'ям людини і його адресою [17]. Для простоти припускається, що ім'я та адреса задаються рядками. Тоді базу даних можна описати таким чином:

```
type AddressDB = [(String,String)]
```

Таким чином, база даних являє собою список пар, першим компонентом яких буде ім'я, а другим - адреса. Тоді функція getAddress, по заданому імені повертає адресу, визначається як на рис. 2.3.

```
getAddress :: AddressDB -> String -> Maybe String
getAddress [] _ = Nothing
getAddress ((name,address):rest) n | n == name = Just address
                                   | otherwise = getAddress rest n
```

Рис. 2.3. Визначення функції getAddress

Для імен, присутніх в базі, функція повертає відповідну адресу. Якщо такого імені в базі немає, повертається Nothing.

Проблеми починаються, коли необхідно здійснювати послідовність запитів. Припустимо, у нас є ще одна база даних, яка містить відповідність між адресами і номерами телефонів:

```
type PhoneDB = [(String,Integer)]
```

Далі, нехай є функція getPhone, за адресою повертає телефон, реалізована також за допомогою типу Maybe. Реалізація цієї функції повністю аналогічна функції getAddress.

У людини може не виявитися телефону, отже, функція повинна повертати значення типу Maybe Integer. Далі, значення Nothing ця функція може повернути в наступних випадках:

– зазначене ім'я не міститься в базі адрес;

– адреса, відповідний вказаному імені, існує, однак він не міститься в базі телефонів.

Виходячи з цих міркувань, функцію getPhoneByName можна визначити так як на рис. 2.4.

```
getPhoneByName :: AddressDB -> PhoneDB -> String -> Integer
getPhoneByName a p n = case (getAddress a n) of
        Nothing -> Nothing
        Just address -> case (getPhone p address) of
                    Nothing -> Nothing
                    Just phone -> Just phone
```

Рис. 2.4. Визначення функції getPhoneByName

Такий стиль програмування не надто витончений, крім того, він провокує помилки. У разі, коли рівень вкладеності запитів зростає, зростає і обсяг повторюваного коду. Програміст, завжди прагне до повторного використання коду, визначить допоміжну функцію, яка відображає використовуваний в функції getPhoneByName шаблон зв'язування функцій, які повертають значення типу Maybe. Яку можна назвати thenMB, зображена на рис. 2.4.

```
thenMB :: Maybe a -> (a -> Maybe b) -> Maybe b
thenMB mB f = case mB of
    Nothing -> Nothing
    Just a -> f a
```

Рис. 2.4. Допоміжна функція для повернення типу Maybe

Розглянемо приймає два аргументи: значення типу Maybe a і функцію, яка буде показувати значення типу a в значення типу Maybe b. Якщо перший аргумент містить значення Nothing, другий аргумент ігнорується. Якщо ж перший аргумент містить реальне значення, огорнуте в конструктор Just, воно витягується з нього і передається в функцію, яка є другим аргументом. В мові Haskell лямбда-абстракція записується у вигляді \x -> expr, функцію getPhoneByName можна записати як на рис. 2.5.

```
getPhoneByName a p n =
    (getAddress a n `thenMB`
        (\address -> getPhone p address))
        `thenMB` (\phone -> Just phone)
```

Рис. 2.5. Функція getPhoneByName з використанням лямбда-абстракції

Або, опускаючи дужки і записуючи з більш наочним розташуванням коду як на рис. 2.6.

```
getPhoneByName a p n = getAddress a n `thenMB`
\address ->
    getPhone p address `thenMB` \phone ->
    Just phone
```

Рис. 2.6. Функція getPhoneByName з більш наочним розташуванням коду

Цей запис слід читати так, ніби результат лівого аргументу оператора 'thenMB' присвоюється імені змінної з лямбда-абстракції правого аргументу.

Для прикладу була визначена функція thenMB, що комбінує обчислення, які можуть або повернути результат, або відмовитися його обчислювати. Сама функція thenMB не залежить від того, які саме обчислення вона комбінує, аби вони задовольняли її сигнатурі. Її можна використовувати не тільки для даного прикладу, а й для будь-яких інших аналогічних завдань. Вона визначає деяке правило комбінування обчислень в ланцюжок, що полягає в тому, що якщо одне з обчислень не виповнилося, не виконується і весь ланцюжок.

Даний приклад можна вдосконалити. Були перераховані випадки, в яких функція getPhoneByName може не знайти телефон. Однак в будь-якому випадку вона поверне значення Nothing. В реальності може цікавити, чому саме вона не знайшла телефон. Нехай функції getPhone, getAddress і getPhoneByName повертають значення типу Value, який можна визначити наступним чином:

```
data Value a = Value a | Error String
```

Значення типу Value a являє собою або значення типу a, огорнуте в конструктор Value, або стрічкове повідомлення про помилку, що міститься в конструкторі Error. Функцію getAddress можна визначити тоді так як на рис. 2.7.

```
getAddress :: AddressDB -> String -> Value String
getAddress [] _ = Error "no address"
getAddress ((name,address):rest) n | n == name = Value address
                                   | otherwise = getAddress rest n
```

Рис. 2.7. Функція getAddress з перевіркою на помилки

У разі помилки getAddress буде повернуто значення Error "no address". Аналогічно можна визначити і функцію getPhone, яка в разі помилки поверне

значення Error "no phone". Тоді функцію getPhoneByName можна визначити як на рис. 2.8.

```
getPhoneByName :: AddressDB -> PhoneDB -> String -> Value Integer
getPhoneByName a p n = case (getAddress a n) of
    Error s -> Error s
    Value address -> case (getPhone p address) of
        Error s -> Error s
        Value phone -> Value phone
```

Рис. 2.8. Функція getPhoneByName з перевіркою на помилки

Тут можна побачити аналогічну проблему, що і з попереднім визначенням. Для її вирішення можна скористатися тим же прийомом, а саме визначити допоміжну функцію як на рис. 2.9.

```
thenV :: Value a -> (a -> Value b) -> Value b
thenV mV f = case mV of
    Error s -> Error s
    Value v -> f v
```

Рис. 2.9. Допоміжна функція

З використанням цієї функції можна спростити визначення на рис. 2.10.

```
getPhoneByName a p n = getAddress a n `thenV` \address ->
    getPhone p address `thenV` \phone ->
    Value phone
```

Рис. 2.10. Спрощене визначення

Не можна не відзначити деяку схожість у функціях thenMB і thenV, а також у визначеннях функції getPhoneByName. Тип Value також є приклад монади.

До цього часу ми припускалося, що записи в базах даних унікальні, тобто кожній людині відповідає тільки одна адреса, а кожній адресі тільки один телефон. Припускаючи, що це не так, тобто одній людині може відповідати декілька адрес, а одній адресі кілька телефонів. Тоді функції getPhone, getAddress і getPhoneByName повинні повертати списки, і їх сигнатури можна записати в спосіб, який зображено на рис. 2.11.

```
getAddress :: AddressDB -> String -> [String]
getPhone :: PhoneDB -> String -> [Integer]
getPhoneByName :: AddressDB -> PhoneDB -> String - [Integer]
```

Рис. 2.11. Сигнатури функцій getPhone та getAddress

Припускаючи, що функції getPhone і getAddress вже визначені, у разі невдачі вони повертають порожні списки, а в разі успішного пошуку списки, що складаються з довільної кількості елементів. Для того щоб, використовуючи ці функції, визначити функцію getPhoneByName, потрібно для кожної адреси, повернутої функцією getAddress, вона повинна викликати функцію getPhone, результати всіх викликів цієї функції необхідно об'єднати в один список. Визначення, що враховує ці особливості, я зображено на рис. 2.12.

```
getPhoneByName a p n =
    case (getAddress a n) of
        [] -> []
        (address:rest) -> getPhone p addresses ++ getPhones p rest
    where getPhones _ [] = []
        getPhones p (x:xs) = getPhone p x ++ getPhones p xs
```

Рис. 2.12. Об’єднання в один список викликів getPhone

Тут також можна визначити допоміжну функцію, яка зображена на рис 2.13.

```
thenL :: [a] -> (a -> [b]) -> [b]
thenL mL f = case mL of
        [] -> []
        (x:xs) -> f x ++ getRest xs f
    where getRest [] _ = []
getRest (x:xs) f = f x ++ getRest xs f
```

Рис. 2.13. Допоміжна функція

З використанням комбінатора визначення набуде вигляду як на рис. 2.14.

```
getPhoneByName a p n = getAddress a n `thenL` \address ->
    getPhone p address `thenL` \phone -> [phone]
```

Рис. 2.14. Допоміжна функція з використанням комбінатора

Монада - це певний тип даних, що передбачає певну стратегію комбінування обчислень значення цього типу [18]. Так, монада Maybe описує таке комбінування обчислень, що обчислення яке не виконалися змушує не виконуватись і весь ланцюжок обчислень. Монада Value передбачає при цьому, що ланцюжок обчислень повинен повернути повідомлення про помилку від обчислення яке не відбулося. Монада список відображає концепцію обчислень, які можуть повернути неоднозначний результат, і ланцюжок обчислень повинен враховувати всі можливі результати. З іншого боку, монада можна розглядати як контейнерний тип, тобто як тип, значення якого містять в собі деяку кількість елементів іншого типу. Особливо яскраво це проявляється на прикладі списків, однак неважко помітити, що типи Maybe і Value також є контейнерними і можуть містити нуль або одне значення типу, що є їх параметром [14].

У стандартній бібліотеці мови Haskell визначено клас типів, які є монадами. Визначення цього класу зображено на рис. 2.15.

```
class Monad m where
return :: a -> m a
(>>=) :: m a -> (a -> m b) -> m b
(>>) :: m a -> m b -> m b
fail :: String -> m a

p >> q = p >>= \ _ -> q
fail s = error s
```

Рис. 2.15. Визначення класу типів, які є монадами

Дане визначення говорить про те, що тип m є монадою, якщо для нього визначені вказані функції і оператори. При цьому необхідно визначити тільки функцію return і оператор >> =, для інших функцій і операторів є визначення за замовчуванням.

2.2. Висновки до розділу

Таким чином, більшість функційних мов програмування мають теоретичну основу в лямбда численні та дуже тісно пов’язані з математикою.

Через характер обчислення лямбди можливі певні докази поведінки систем, побудованих на її принципах. Насправді, здатність бути доведеним (тобто правильним) є важливим поняттям в лямбда численні і надає можливість певних міркувань та висновків щодо систем лямбда числення. Обчислення лямбда також пов'язане з теорією типів та теорією категорій (і покладається на них).

Моделі Тюринга, навпаки менше покладаються на теорію типів і більше на структурування обчислень як серії переходів стану в базовій моделі. Моделі обчислень машини Тюринга складніше робити твердження і не піддаються тим самим видам математичних доказів і маніпуляцій, як це роблять програми, засновані на ламбда численні. Однак це не означає, що такий аналіз неможливий. Деякі важливі аспекти моделей лямбда числення використовуються при вивченні віртуалізації та статичного аналізу програм.

Оскільки функційне програмування покладається на ретельний підбір типів та перетворення між типами, функційне програмування можна сприймати як «математичне».

РОЗДІЛ 3
РОЗРОБКА ПРОГРАМ НА ФУНКЦІЙНИХ МОВАХ

3.1. Цільова платформа

В якості нашої цільової платформи ми вибрали мікроконтролери з 8-бітовим процесором ATmega328, такі як Arduino Nano, Arduino UNO та подібні пристрої, які зображено на рис. 3.1. Вибір Arduino яка базується на мікроконтролері ATmega328, обумовлений великою популярністю та легкістю використання цієї платформи.

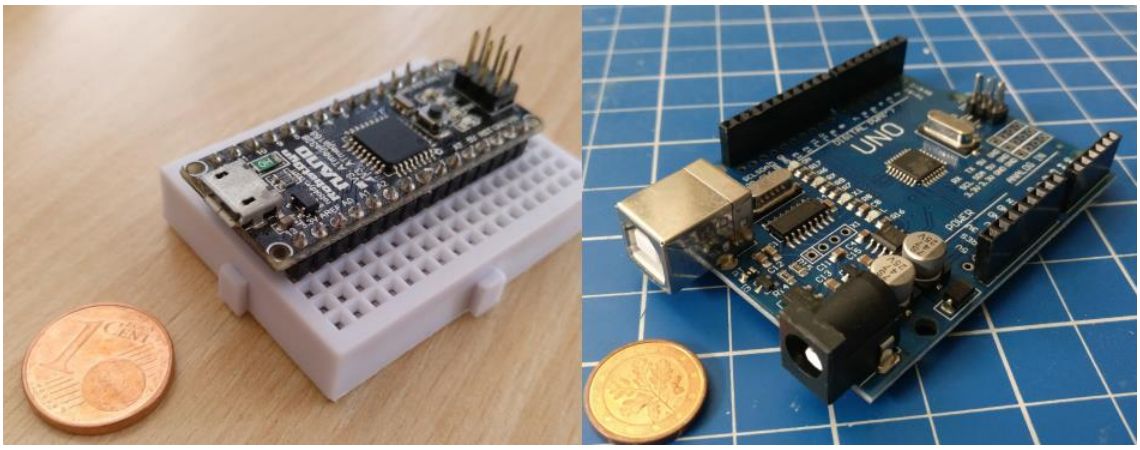

Рис. 3.1. Сумісні плати Arduino Nano та Uno

Платформи Arduino порівняно дешеві і широко використовуються у комерційних та навчальних проектах. Ця цільова платформа поставляється з набором обмежень, які потрібно враховувати під час розробки програмного забезпечення для них:

– Доступно лише 2 Кб SRAM, яка використовується як для хіпу, так і для стека. Це означає, що ми обмежені в оперативній пам’яті для зберігання отриманих даних та в дизайні алгоритму щодо глибини виклику функцій.

– 32 Кб флеш-пам'яті можна використовувати для зберігання програми. Це може здатися багато в порівнянні, але ця пам’ять також

використовується для зберігання додаткових бібліотек для периферійного доступу, які потрібні користувачеві. Це також досить обмежує враховуючи дизайн алгоритму та кількість вихідного коду, який ми можемо генерувати. Файли заголовка Arduino.h з введенням і виведенням і записом на послідовний порт вже використовують 2 КБ цієї пам'яті, коли компілюються з увімкненою оптимізацією розміру.

– Для прошивки використовується завантажувач розміром близько 2 Кб.

– Процесор ATmega328 має робочу частоту 16 МГц, що багато в порівнянні з кількістю даних, над якими ми маємо працювати.

– Існує додаткова пам’ять EEPROM 1 Кб. Зберігання у цю память повільне і обмежене кількістю циклів запису.

Обрана цільова платформа дає нам обмеження щодо використання ресурсів, на які потрібно звертати увагу. Оскільки ми під час компіляції генеруємо C-код, наш підхід може бути застосований без великих змін і для інших платформ та мікроконтролерів. Такий підхід також корисний тим що, вже існує величезна екосистема для розробки Arduino, для якої уже створено багато бібліотек для роботи з різноманітними датчиками, зовнішніми пристроями та шинами і немає жодної причини, щоб ці бібліотеки дублювати.

3.2. Програмне забезпечення та мови програмування для функційного програмування мікроконтролерів

У цьому підрозділі ми розглянемо інструменти для функційного програмування мікроконтролерів ATmega328, такими інструментами є Juniper та frp-arduino.

Juniper підтримує багато ознак типових для функційних мов програмування, включаючи алгебраїчні типи даних, записи, відповідність шаблону, незмінні структури даних, параметричні поліморфні функції і анонімні

функції (lambda). Деякі імперативні концепції програмування також присутні в Juniper, наприклад, для циклів “while”, “do while”, можливість позначити змінні як “mutable” і змінні посилання. А саме головне код написаний на Juniper перекладається на стандартний C ++ і компілюється за допомогою існуючих інструментами розробки Arduino, що дозволяє програмам Juniper застосовуватися до пристроїв, обмежених ресурсами, і забезпечує безперебійну взаємодію з існуючими бібліотеками C ++ для цих пристроїв.

Frp-arduino з іншого боку використовує високорівневі конструкції. мова Frp вбудована в мову Haskell, це означає, що програми написані на Frp мові, насправді програми на Haskell. Однак ці програми не будуть виглядати як стандартний Haskell, оскільки вони використовують спеціальні оператори, які більше підходять для парадигми FRP. Для того, щоб виконувати на Arduino FRP програму, потрібно спочатку скомпілювати її до вихідного файлу C, який потім потрібно перетворити в код збірки avr, використовуючи avr gcc toolchain.

Для мікрокомп’ютерів створювати програми на функційних мовах набагато легше, адже на мікрокомп’ютер можна встановити операційну систему, яка надає змогу створювати, виконувати та відлагоджувати програми. Для цього існує безліч інструментів та середовищ розробки.

3.3. Мова Juniper та її застосування

На рис. 3.2 показана програма Juniper, яка щосекунди вмикає та вимикає світлодіод. У цьому базовому прикладі код Juniper здається значно складнішим, ніж код аналогічної програми на C++. Це справедливо лише для простих проектів [19]. Зі збільшенням складності проекту, код C++ збільшується та ускладнюється набагато швидше, ніж код Juniper. Що ще важливіше, код показаний на рис. 3.2, є композиційним і може бути використаним багато разів, тоді як код C++ - ні.

```
module Blink
open(Prelude, Io, Time)

let boardLed : int16 = 13
let tState : timerState ref = Time:state()
let ledState : pinState ref = ref low()

fun blink() : sig<pinState> = (
  let timerSig = Time:every(1000, tState);
  Signal:foldP<uint32, pinState>(
    fn (currentTime : uint32,
          lastState : pinState) : pinState ->
      Io:toggle(lastState),
    blinkState, timerSig)
)

fun setup() : unit =
  Io:setPinMode(boardLed, Io:output())

fun main() : unit = (
  setup();
  while true do
    Io:digOut(boardLed, blink())
  end
)
```

Рис. 3.2. Базова програма на Juniper, для блимання світлодіода

На рис. 3.2 main функція викликає функцію setup, яка встановлює вбудований світлодіод для виведення. Програма Juniper потім входить у нескінченний цикл, який виводить сигнал, який блимає світлодіодом. Функція блимання створює сигнал таймера, по якому значення часової позначки рухається кожні 1000 мілісекунд. Цей сигнал використовується як параметр до функції foldP. Лямбда, передана foldP, приймає значення, яке вона раніше повернула разом зі значенням на вхідному сигналі. Ця операція, залежить від певного стану, тому використовується посилання для зберігання значень між викликами функції foldP. Тип pinState має два конструктори значень: Io:low() та Io:high(). Функція Io:toggle перемикається між цими двома конструкторами значень. Кінцевим результатом функції блимання є сигнал типу sig <pinState>.

Сумісність з існуючими бібліотеками C++ є критично важливою для успіху мови, орієнтованої на програмування мікроконтролерів Arduino. Бібліотеки, що

управляють кожним датчиком, приводом та іншими пристроями виводу, підключеними до Arduino, створюються на C++. Якщо потрібно використовувати бібліотеку C++, потрібно писати власну обгортку бібліотеки на Juniper.

Оскільки Juniper компілюється на C++, мова дозволяє робити вставки C++ коду між рядки. Після компіляції вбудований код C++ загортається всередину негайно викликаної функції, а це означає, що неможливо ввести змінні в поточну область функції. Повернене значення функції - Prelude::unit, тобто значення, що повертається будь-яким вставленим кодом C++, є одиницею. Вбудований код C++ записується між двома символами # хештега.

У обгортках Juniper тип вказівника використовується для вказівки на місце пам'яті. Цей тип вказівника насправді є об'єктом розумного вказівника C++. Розумний вказівник відстежує кількість посилань на об'єкт C ++ і автоматично звільняє пам'ять, коли на нього більше немає посилань. Внутрішньо розумний вказівник відстежує об'єкт C ++, використовуючи вказівник void *. Це означає, що під час взаємодії з розумним вказівником у C ++ коді потрібно використовувати приведення типу. Вираз null використовується для створення нового розумного вказівника:

```
let p : pointer = null
```

Тепер p - змінна вказівника типу, яка після компіляції буде Juniper::shared ptr<void> C ++ типу. Ключове слово "null" вказує на те, що розумний вказівник в даний час вказує на значення C ++ NULL. Можна змінити те, на що вказує розумний вказівник, використовуючи метод set спільного класу ptr C ++. Встановлений метод просто приймає один параметр типу void *.

```
#p.set((void *) new MyClass(...));#
```

Для доступу до вмісту розумного вказівника може бути використаний метод get з класу Juniper::shared ptr. Метод get не містить параметрів і повертає вказівник у вигляді void* типу C ++. Потім можна взаємодіяти з об'єктом, перетворивши його до потрібного типу:

```
#((MyClass *) p.get())-> ...; #.
```

Juniper не виконує жодного декорування імен змінних, назв типів або імен функцій. Це означає, що вбудований C ++ може безпечно використовувати ці об'єкти без обмежень. Наприклад, ми можемо отримати ціле число, збережене в MyClass, використовуючи наступний код:

```
(let mutable x : int32 = 0; #x = ((MyClass *) p.get())-> getX(); #; x)
```

Декларація включення дозволяє включати файли заголовків з бібліотек C ++ у вихідний файл C ++:

```
include(" < heaser1.h > ", "\" < header2.h >\"",...)
```

Компіляція програми Juniper - досить простий процес. Програміст має записати у .jun файли, в яких міститься код для одного модуля. Щоб полегшити написання коду Juniper, для текстового редактора Atom можна використовувати плагін для синтаксичної розмальовки коду. Потім ці модулі передаються компілятору, який також включає стандартні модулі бібліотеки. Код аналізується, перевіряється і потім компілюється в один файл C ++ .cpp. Цей файл C ++ потім компілюється та завантажується в Arduino. Компілятор Juniper написаний на F # і доступний для платформ Windows та Linux [19].

3.4. Приклади функційних програм на мові Haskell для мікроконтролерів та їх компіляція

Перш ніж писати будь який код, потрібно для початку встановити кілька залежностей, а саме:

– makefile для Arduino скетчів, який знає, як будувати скетчі Arduino. Він визначає цілі робочі процеси для компіляції коду, прошивання його Arduino і навіть спілкування через послідовний порт;

– платформу Haskell, для компіляції коду на Haskell;

– платформу Frp-arduino для компіляції Haskell в його до вихідного файлу C, який потім потрібно перетворити в код збірки avr, використовуючи avr gcc toolchain [18].

Для компіляції та загрузки коду в мікроконтролер Arduino необхідно скористатися командою:

BOARD=[name of board] ./make [name of example] upload

Лістининг коду для мигання світлодіоду зображено на рис. 3.3.

```
import Arduino.Uno
main = compileProgram $ do
    digitalOutput pin13 =: clock ~> toggle
```

Рис. 3.3. Лістининг коду для мигання світлодіоду

`import Arduino.Uno` - імпортує функції, які дозволяють визначити програму в EDSL.

`main = compileProgram $ do` - основна функція є стандартною main функцією в Haskell.

Код для мигання парою світлодіодів, який показує, як об'єднати два значення разом і вивести їх на два різних виходи зображено на рис. 3.4.

```
    import Arduino.Uno
    import Data.Tuple (swap)
    main = compileProgram $ do
        setupAlternateBlink pin11 pin12 (createVariableTick a0)
    setupAlternateBlink :: GPIO -> GPIO -> Stream a -> Action ()
    setupAlternateBlink pin1 pin2 triggerStream = do
        output2 (digitalOutput pin1) (digitalOutput pin2) =: alternate
triggerStream
        where
            alternate :: Stream a -> Stream (Bit, Bit)
            alternate = foldpS2Tuple (\_ -> swap) (bitLow, bitHigh)
    createVariableTick :: AnalogInput -> Stream ()
    createVariableTick limitInput = accumulator limitStream timerDelta
        where
            limitStream :: Stream Arduino.Uno.Word
            limitStream = analogRead limitInput ~> mapS analogToLimit
            analogToLimit :: Expression Arduino.Uno.Word -> Expression
Arduino.Uno.Word
            analogToLimit analog = 1000 + analog * 20
```

Рис. 3.4. Лістининг коду для мигання парою світлодіодів

Приклад відправлення байтів у послідовний порт зображено на рис. 3.5.

```
import Arduino.Uno
main = compileProgram $ do
    digitalOutput pin13 =: clock ~> toggle
    uart =: timerDelta ~> mapSMany formatDelta ~> flattenS
formatDelta :: Expression Arduino.Uno.Word -> [Expression [Byte]]
formatDelta delta = [ formatString "delta: "
                    , formatNumber delta
                    , formatString "\r\n"
                    ]
```

Рис. 3.5. Лістининг коду для відправлення байтів у послідовний порт

Приклад програми відображення тексту на РК-дисплеї зображено на рисунку 3.6.

```
import Arduino.Uno
import qualified Arduino.Library.LCD as LCD
main = compileProgram $ do
    tick <- def clock
    digitalOutput pin13 =: tick ~> toggle
    setupLCD [ bootup ~> mapSMany (const introText)
             , timerDelta ~> mapSMany statusText
             ]
introText :: [Expression LCD.Command]
introText = concat
    [ LCD.position 0 0
    , LCD.text "hello, world!"
    ]

statusText :: Expression Arduino.Uno.Word -> [Expression LCD.Command]
statusText delta = concat
    [ LCD.position 1 0
    , LCD.text ":-)"
    ]
setupLCD :: [Stream LCD.Command] -> Action ()
setupLCD streams = do
    LCD.output rs d4 d5 d6 d7 enable =: mergeS streams
    where
        rs     = digitalOutput pin3
        d4     = digitalOutput pin5
        d5     = digitalOutput pin6
        d6     = digitalOutput pin7
        d7     = digitalOutput pin8
        enable = digitalOutput pin4
```

Рис. 3.6. Лістининг коду для відображення тексту на РК-дисплеї

Для визначення раціональності заміни C++ на Haskell із семантичної точки зору необхідно проаналізувати чи дійсно аналогічний проект на Haskell має менший об’єм коду, а також чи на його написання витрачається стільки ж зусиль.

Отже, для порівняння на основі цих двох критеріїв обрано проект що спочатку код був написаний за допомогою C++, а потім переписаний на Haskell.

Для оцінки зусиль витрачених при написанні проектів було взято час виконання типових завдань протягом розробки проектів. А саме сума таких показників як: створення коду для активності (ініціалізація елементів, логіка взаємодії з користувачем), створення моделей, виправлення помилок, пошук інформації в інтернеті. Ці показники вимірювались кількістю завдань виконаних кожного тижня. Завдання в даному випадку є наприклад якийсь алгоритм чи форматування даних. Виміри при написанні коду Haskell з урахуванням вивчення програмістом нової мови.

Для порівняння об’єму коду та складності структури пораховано кількість файлів у проекті, а також кількість рядків.

На рис. 3.7 і 3.8 зображено що кількість файлів на Haskell трошки менша за C++, а також у півтора рази менше кількість рядків

В основному, різке зменшення об’єму коду це результат використання лямбда числень у Haskell, що дозволяють уникнути рутинних дій при оголошенні структури класів моделей і описати всю структуру інформації, що передається й отримується, в одному файлі.

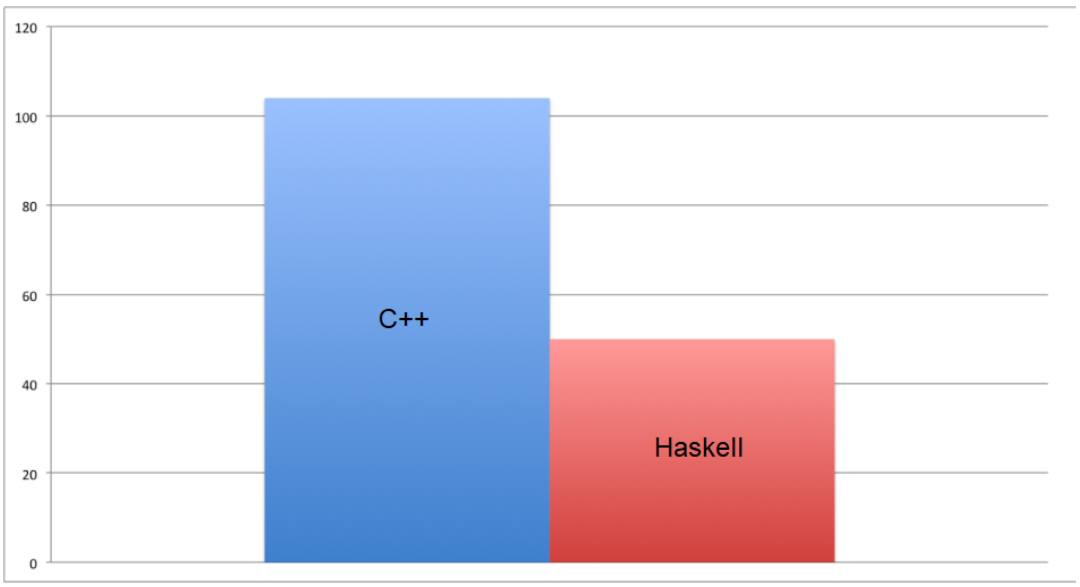

Рис. 3.7. Кількість файлів у проектах

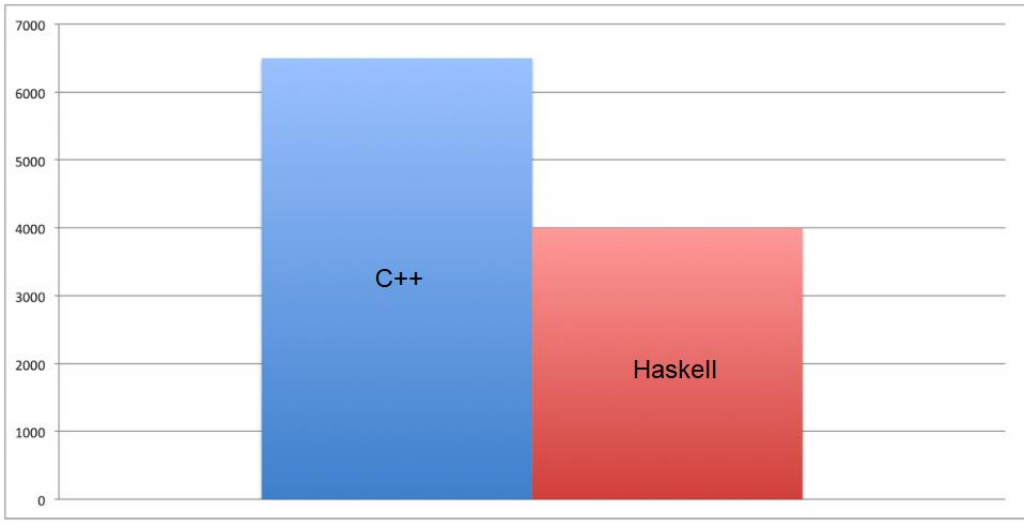

Рис. 3.8. Кількість рядків коду у проектах

Звісно, це не єдина причина. Також відчутний вплив монад при роботі з масивами, вони дозволяють зменшити кількість написаного коду в двічі, в тричі.

На рис. 3.9 видно що C++ має стабільну швидкість написання коду, кожного тижня було зроблено як мінімум 1 завдання. Haskell ж має трохи інші показники. Так як до уваги взято спільні для обох мов завдання, ми бачимо що перші 3 тижні прогрес майже не рухався. Це результат того що в ці тижні створювалась математична модель алгоритму та проводилась адаптація інструментів для роботи з сторонніми бібліотеками. Після цих робіт спостерігається різкий скачок, він пояснюється тим, що готову математичну

модель легко переводити у код мови Haskell. Вже на шостий тиждень весь функціонал попередньої версії був переписаний на Haskell.

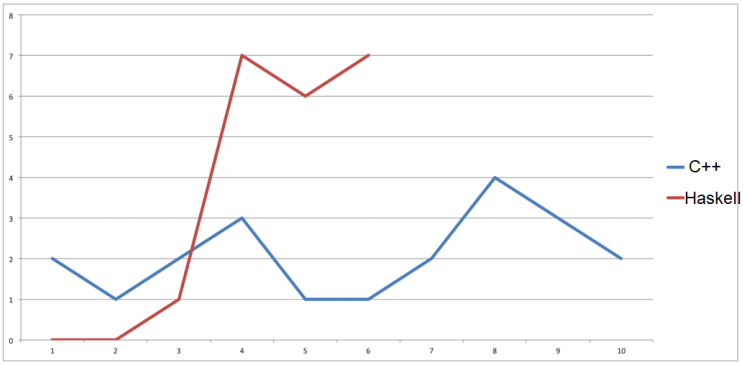

Рис. 3.9. Щотижнева кількість виконаних завдань при написанні проектів

З цього можна зробити висновок про те що Haskell потребує менше зусиль у рівних умовах написання програмного забезпечення для мікроконтролера. Також необхідно зазначити, що за час роботи над проектом, кількість помилок при тестуванні зменшилась у півтора рази. Покращилась надійність при роботі, адже при написанні програми на Haskell не використовувалися вказівники, звичайні цикли та масиви.

3.5. Перелік рекомендацій для написання функційних програм

Варто Уникати бібліотек, які роблять все за вас. Функційне програмування полягає у створенні складних рішень з невеликих, легко зрозумілих частин. Це означає уникати монолітних, фреймворків і бібліотек, які роблять все за вас на користь того, щоб вирішити більшість частин проблеми самостійно, що полегшить відладку програми.

Пам'ятайте, що функції відображають набори значень до наборів значень. В основному функційна мова програмування працює з функціями. Вони набагато більше схожі на математичні функції з алгебри, на відміну від роботи подібних процедур з такої мови, як C або Python. Ці функції приймають певний набір значень як вхідні дані і повертають певний набір значень як вихідні. Які значення функція повертає, задані конкретними вхідними даними, це визначає

поведінку функції. Як приклад можна розглянути функцію, яка повертає довжину рядка. Як її вхідні дані вона може отримувати будь-який рядок, а як її вихідні може повертати будь-яке додатне ціле число, включаючи 0. Вона не може повернути від'ємне число, а також не може повертати числа з плаваючою комою. Крім того, її поведінка є не визначеною, якщо їй на вхід подати щось інше, наприклад булеве значення. Пам'ятаючи про цей принцип, допоможе зрозуміти, коли композиції "просто працюватимуть".

Варто думати про властивості своїх функцій. Є всілякі властивості, яким функція може підкорятися. Вони можуть бути внутрішніми для самої функції, визначаючи її поведінку ізольовано, або вони можуть бути зовнішніми, визначаючи поведінку стосовно інших функцій. Внутрішні властивості визначатимуть, які вхідні дані відображаються на які вихідні, а отже, визначають поведінку функції. Зовнішні властивості дозволять переконатися, що ваша функція не має дивної поведінки. Прикладом внутрішньої властивості є референсна прозорість, або функція повертає те саме на вихід, коли викликається з тими же вхідними даними. Прикладом зовнішньої властивості є асоціативність, тобто можна оцінити функцію в будь-якому порядку.

Абстрактна математика - ваш друг. абстрактна математика може бути дещо непростою, але вчені математики вивчають, як описувати математично речі чи події протягом сотень років. Ви можете ставитися до концепцій, які вони виявили, як шаблони дизайну, які слід застосувати у власному коді.

3.6. Висновки до розділу

У даному розділі ми розглянули інструменти за допомогою яких можливо створювати програмне забезпечення для мікроконтролерів. До недоліків цього підходу можна віднести те, що за допомогою цих інструментів можна програмувати тільки мікроконтролери платформи Arduino, але ці проекти є у

вільному доступі, і кожен бажаючий може доповнити їх необхідним функціоналом.

У даному розділі, описаний експеримент, який показав ефективність використання функційного підходу при створенні двох аналогічних програм для мікроконтролера з використанням різних парадигм та мов програмування, і результатом було те, що написання програми використовуючи функційний підхід, у два рази скоротився код та проект був закінчений швидше.

Також у даному розділі було складено перелік рекомендацій, яких слід дотримуватися при написанні програм функційними мовами програмування.

РОЗДІЛ 4
ОБҐРУНТУВАННЯ ЕКОНОМІЧНОЇ ЕФЕКТИВНОСТІ

4.1. Визначення стадій технологічного процесу та загальної тривалості проведення НДР

Економічне обґрунтування дипломної роботи магістра є суттю даного розділу, оскільки, дозволяє встановити доцільність проведення науководослідних робіт і економічно обґрунтувати доцільність застосування тих чи інших засобів.

Метою дипломної роботи магістра є дослідження методів та засобів для функційного програмування мікроконтролерів.

Як відомо, розробка надійного і ефективного програмного забезпечення для мікроконтролерів вимагає значних затрат часу, особливо якщо використовуються мови програмування низького рівня та імперативна парадигма програмування. На сьогоднішній день мови програмування, в яких можна писати код згідно функційної парадигми, все більше розвиваються і їх можна використовувати для програмування мікроконтролерів. Використовуючи функційну парадигму при програмуванні можна значно скоротити час написання якісного програмного забезпечення. Також варто зауважити, що затрати часу залежать від кваліфікації розробника і його можливостей. Розробник повинен у достатній мірі володіти навиками програмування, вміти адекватно застосовувати математичний апарат, бути добре обізнаним з об'єктом дослідження.

Розробку програмного забезпечення можна поділити на такі етапи:

а) постановка задачі;

б) збір потрібної інформації і наступне її опрацювання;

в) прийняття рішень щодо вибору оптимального шляху розв'язання поставленої задачі;

г) аналіз математичної моделі інформаційної системи;

д) розробка алгоритму програми інформаційної системи;

е) налаштування середовища розробки і роботи вже готової програми;

ж) написання програми;

з) написання і оформлення документації.

Для оцінки тривалості виконання окремих робіт використовують нормативи часу або попередній досвід. До таких нормативів відносять тривалість написання операцій (команд), які в деяких підприємствах становлять: для одної операції - 0,5-1,6 год та 8 годин для п'яти операцій (тривалість зміни).

У разі їх відсутності звертаються до експертних оцінок по встановленню тривалості кожного етапу (стадії):

при трьох оцінках:

Твс=(tmin+4тн.й+tmax)/6,

при двох оцінках:

Твс=(3tmin+2tmax)/5,

де Твс - очікуване (середнє) значення тривалості виконання етапу (стадії); tmin., тн.й., tmax- відповідно мінімальна, найбільш імовірна і максимальна оцінки тривалості виконання етапу (стадії).

Для порівняння тривалості розробки програмного продукту при використанні імперативної та функційної парадигм програмування доцільно взяти середній час розробки одного й того ж продукту з використанням різних парадигм програмування та дані витрат часу на виконання окремих стадій (етапів) звести у таблицю 4.1.

Витрати часу керівника проекту на виконання окремих стадій (етапів) при недостатній кількості інформації доцільно приймати в межах 5% сумарних витрат часу інженерів на виконання цих стадій (етапів).

Таблиця 4.1

Основні етапи і час їх виконання при використанні різних парадигм програмування

№ з/п	Етап	Середній час виконання етапу, год			
		Імперативна		Функційна	
		інженер	керівник	інженер	керівник
1	постановка задачі	3	10	3	10
2	збір потрібної інформації і наступне її опрацювання	10	5	10	5
3	прийняття рішень щодо вибору оптимального шляху розв'язання поставленої задачі	5	4	8	6
4	аналіз математичної моделі інформаційної системи	15	10	20	5
5	розробка алгоритму програми інформаційної системи	15	5	3	1
6	налаштування середовища розробки і роботи вже готової програми	2	1	2	1
7	написання програми	80	5	15	10
8	написання і оформлення документації	20	10	10	5
разом		150	50	71	43

Разом для створення повноцінного програмного продукту для мікроконтролера використовуючи імперативну парадигму програмування інженеру потрібно буде 150 годин а його керівнику 50. При використанні фінкційної парадигми програмування на розробку ідентичного програмного продукту інженеру знадобиться 71 година, що менше на 52.7%, а його керівнику 43 години що менше на 14%.

4.2. Визначення витрат на оплату праці та відрахувань на соціальні заходи при використанні різних парадигм програмування

Відповідно до Закону України “Про оплату праці” заробітна плата – це “винагорода, обчислена, як правило, у грошовому виразі, яку власник або уповноважений ним орган виплачує працівникові за виконану ним роботу”.

Розмір заробітної плати залежить від складності та умов виконуваної роботи, професійно-ділових якостей працівника, результатів його праці та господарської діяльності підприємства. Заробітна плата складається з основної та додаткової оплати праці.

Основна заробітна плата нараховується на виконану роботу за тарифними ставками, відрядними розцінками чи посадовими окладами і не залежить від результатів господарської діяльності підприємства.

Додаткова заробітна плата – це складова заробітної плати працівників, до якої включають витрати на оплату праці, не пов’язані з виплатами за фактично відпрацьований час. Нараховують додаткову заробітну плату залежно від досягнутих і запланованих показників, умов виробництва, кваліфікації виконавців. Джерелом додаткової оплати праці є фонд матеріального стимулювання, який створюється за рахунок прибутку.

Основна з/п складається із прямої з/п і доплати, яка при укрупнених розрахунках становить 25% - 35% від прямої з/п. При розрахунку з/п кількість робочих днів в місяці слід приймати – 25,4 дні/міс., що відповідає 203,2 год./міс. Розмір місячних окладів керівника та інженерів слід приймати згідно існуючих на даний час норм. Основна заробітна плата розраховується за формулою:

$$З_{осн} = Т_с \cdot К_г, \qquad (5.1)$$

де $Т_с$ – тарифна ставка, грн.;

$К_г$ - кількість відпрацьованих годин.

Посадові оклади (тарифні ставки) за розрядами Єдиної тарифної сітки визначаються шляхом множення окладу (ставки) працівника 1 тарифного розряду на відповідний тарифний коефіцієнт. У разі коли посадовий оклад (тарифна ставка) визначені у гривнях з копійками, цифри до 0,5 відкидаються, від 0,5 і вище - заокруглюються до однієї гривні. У 2019 році посадові оклади (тарифні ставки) розраховуються згідно з Законом України "Про Державний бюджет України на 2019 рік".

Мінімальна зарплата в 2019 р. прирівняна до прожиткового мінімуму для працездатних осіб (тобто з 01.01.2019 - 4173 гривень), в погодинному розмірі — 25,13 гривень.

Тарифні ставки програмістів які використовують імперативну парадигму програмування: керівник проекту – 295 грн./год., інженер – 95 грн./год.

Тарифні ставки програмістів які використовують функційну парадигму програмування, вищі ніж програмістів які використовують імперативну парадигму програмування, адже для цього потрібна вища кваліфікація та кращі знання математики: керівник проекту – 300 грн./год., інженер – 115 грн./год.

Основна заробітна плата становитиме:

$$З_{осн} = Т_{осн} \cdot К_{год}, \qquad (5.2)$$

При виконанні проекту програмістами які використовують імперативну парадигму програмування:

Керівник проекту $З_{осн}$=295х50=14250 грн.

Інженер $З_{осн}$=95х150=14750 грн.

При виконанні проекту програмістами які використовують функційну парадигму програмування:

Керівник проекту $З_{осн}$=300х43=12900 грн.

Інженер $З_{осн}$=115х71=8165 грн.

Додаткова заробітна плата становить 10–15 % від суми основної заробітної плати:

$$З_{дод} = З_{осн} \cdot К_{допл}, \quad (5.3)$$

При виконанні проекту програмістами які використовують імперативну парадигму програмування:

Керівник проекту $З_{дод}$=14250х0.1=1425 грн.

Інженер $З_{дод}$=14750х0.1=1475 грн.

При виконанні проекту програмістами які використовують функційну парадигму програмування:

Керівник проекту $З_{дод}$=12900х0.1=1290 грн.

Інженер $З_{дод}$=8165х0.1=817 грн. ,

де $К_{допл.}$ – коефіцієнт додаткових виплат працівникам 0,1.

Звідси загальні витрати на оплату праці ($В_{О.П.}$) визначаються за формулою, і становлять:

$$В_{О.П.} = З_{осн} + З_{дод}, \quad (5.4)$$

При виконанні проекту програмістами які використовують імперативну парадигму програмування:

Керівник проекту: $В_{О.П.}$=14250+1425=15675 грн.

Інженер : $В_{О.П.}$=14750+1475=16225 грн..

При виконанні проекту програмістами які використовують функційну парадигму програмування:

Керівник проекту: $В_{ОП}$=12900+1290=14190 грн.

Інженер : $В_{ОП}$=8165+817=8982 грн..

Таким чином загальна сума програмістам які використовують імперативну парадигму програмування становитиме 31900 грн. а при використанні функційної парадигми 23172 грн.

Крім того, слід визначити відрахування на соціальні заходи:

– податок на доходи фізичних осіб: 18%;

– військовий збір 1,5%;

– єдиний соціальний внесок 22%.

У сумі зазначені відрахування становлять 41,5%.

Отже, сума відрахувань на соціальні заходи буде становити:

$$В_{с.з.} = ФОП \cdot 0{,}415, \quad (5.5)$$

де ФОП – фонд оплати праці, грн.

Проведені розрахунки витрат на оплату праці зведемо у наступну табл. 4.2.

Таблиця 4.2

Зведені розрахунки витрат на оплату праці

№ п/п	Категорія працівників	Основна заробітна плата, грн.			Додаткова заробітна плата, грн.	Нарах. на ФОП, грн.	Всього витрати на оплату праці, грн.
		Тарифна ставка, грн.	К-сть відпрацьов. год.	Фактичне нарах. з/пл., грн.			
Імперативна парадигма програмування							
1.	Керівник проекту	295	50	14250	1425	6 505	22180
2.	Інженер	95	150	14750	1475	6 733	22958
Разом				29000	2900	13238	45138
Функційна парадигма програмування							
3.	Керівник проекту	300	43	12900	1290	5888	20079
4.	Інженер	115	71	8165	817	3728	12710
Разом				21065	2107	9616	32789

4.3. Розрахунок витрат на електроенергію

Затрати на електроенергію 1-ці обладнання визначаються за формулою:

$$З_е = W \cdot T \cdot S, \quad (5.6)$$

де W – необхідна потужність, кВт;

T– кількість годин роботи обладнання;

S – вартість кіловат-години електроенергії.

Згідно з постановою НКРЕ України від 01.01.2019 р. вартість електроенергії становить 142,50 коп./кВт.год.

Потужність комп'ютера – 450 Вт з підключеним маршрутизатором, кількість годин роботи обладнання згідно таблиці 4.1 – для програмістів які використовують імперативну парадигму програмування 200 годин, отже $З_е$=0,45х200х1.425=128,25 грн.

Для програмістів які використовують функційну парадигму програмування $З_е$=0,45х114х1,425=73,10 грн.

4.4. Розрахунок суми амортизаційних відрахувань

Характерною особливістю застосування основних фондів у процесі виробництва є їх відновлення. Для відновлення засобів праці у натуральному виразі необхідне їх відшкодування у вартісній формі, яке здійснюється шляхом амортизації.

Амортизація – це процес перенесення вартості основних фондів на вартість новоствореної продукції з метою їх повного відновлення.

Комп'ютери та оргтехніка належать до четвертої групи основних фондів. Для цієї групи річна норма амортизації дорівнює 60 % (квартальна – 15 %).

Для визначення амортизаційних відрахувань застосовуємо формулу:

$$А = \frac{Б_в \cdot Н_А}{100}, \quad (5.7)$$

де А – амортизаційні відрахування за звітний період, грн..

$Б_В$ – балансова вартість комп'ютера, на початок звітного періоду, грн..

$Н_А$ – норма амортизації, %.

Амортизаційні витрати становлять 3000 грн.

4.5. Обчислення накладних витрат

Накладні витрати пов'язані з обслуговуванням виробництва, утриманням апарату управління підприємства (фірми) та створення необхідних умов праці.

Накладні витрати можуть становити 20% від суми основної та додаткової заробітної плати працівників:

$$Н_В = В_{О.П.} \cdot 0,2, \quad (5.8)$$

При використанні імперативної парадигми програмування накладні витрати становитимуть $Н_В$=31900х0,2=6380 грн. При використанні функційної парадигми $Н_В$=23172х0,2=4634,4 грн.

4.6. Складання кошторису витрат та визначення собівартості програмного продукту

Результати проведених вище розрахунків зведемо у табл. 4.3. Собівартість ($С_В$) програмного продукту розрахуємо за формулою:

$$С_В = В_{О.П.} + В_{с.з.} + З_{м.в.} + З_е + Т_в + Н_в + А, \quad (5.9)$$

Таблиця 4.3

Кошторис витрат на програмний продукт при використанні різних парадигм

Зміст витрат	Сума, грн.	В % до загальної суми	Сума, грн.	В % до загальної суми
	Імперативна парадигма програмування		Функційна парадигма програмування	
Витрати на оплату праці (основну і додаткову заробітну плату)	31900	58,37	23172	57,22
Відрахування на соціальні заходи	13238	24,22	9616	23,74
Витрати на електроенергію	128,25	0,23	73,1	0,18
Амортизаційні відрахування	3000	5,49	3000	7,4
Накладні витрати	6380	11,67	4634,4	11,44
Собівартість	54646,25	100	40495,5	100

4.7. Розрахунок ціни НДР

Ціну НДР можна визначити за формулою:

$$Ц = \frac{C_В \cdot (1 + P_{рен}) + K \cdot B_{н.і.}}{K} \cdot (1 + ПДВ), \qquad (5.10)$$

де $P_{рен.}$ – рівень рентабельності, 30 %;

K – кількість замовлень, од. (встановлюється лише при розробці програмного продукту та мікропроцесорних систем);

$B_{н.і.}$ – вартість носія інформації, грн. (встановлюється лише при розробці програмного продукту);

ПДВ – ставка податку на додану вартість, (20 %).

Оскільки розробка є прикладною, і використовуватиметься тільки для одного підприємства, то для розрахунку ціни не потрібно вказувати коефіцієнти K та $B_{н.і}$, оскільки їх в даному випадку не потрібно.

Тоді, формула для обчислення ціни розробки буде мати вигляд:

$$Ц = C_B \cdot (1 + P_{рен}) \cdot (1 + ПДВ), \quad (5.11)$$

Звідси ціна на проект складе:

$$Ц = 54646{,}25 \cdot (1 + 0{,}3) \cdot (1 + 0{,}2) = 85248{,}15 \text{ грн.}$$

Таким чином при використанні імперативної парадигми ціна рівна 85248,15 грн.

Визначимо величину прибутку:

$$П = Ц - С_в, \quad (5.12)$$

Згідно даної формули отримаємо 30601,9 грн.

При використанні функційної парадигми ціна буде складати 63172,98 грн. та прибуток 22677,48 грн. але варто врахувати те що розробка з використанням функційної парадигми зайняла майже вдвічі менше часу ніж з використанням імперативної.

4.8. Визначення економічної ефективності і терміну окупності капітальних вкладень

Ефективність виробництва – це узагальнене і повне відображення кінцевих результатів використання робочої сили, засобів та предметів праці на підприємстві за певний проміжок часу.

Економічна ефективність ($E_р$) полягає у відношенні результату виробництва до затрачених ресурсів:

$$E_р = П/C_в, \quad (5.13)$$

де П – сумарна вартість виробленої продукції за рік;

$C_В$ – собівартість.

Враховуючи те що у році 250 робочих днів та те що у кожному з них по 8 робочих годин, за один рік можна створити 2000/150=13,3 програмних продукти використовуючи імперативну парадигму програмування та 2000/71=28,16 програмних продукти використовуючи функційну парадигму програмування.

Якщо програмний продукт створюється програмістами які використовують імперативну парадигму програмування, $E_Р$=1133800,39/726795,12=1,56.

Поряд із економічною ефективністю розраховують термін окупності капітальних вкладень ($T_р$):

$$T_р = \frac{1}{E_Р}, \quad (5.14)$$

$$T_Р = 1 / 1,56 = 0,64 \text{ р.}$$

При використанні функційної парадигми $E_Р$=1778951,11/1140353,28=1,56. Та $T_Р$ = 1 / 1,56 = 0,64 р.

Про доцільність використання імперативної чи функційної парадигми можна сказати при врахуванні наступних критеріїв:

Таблиця 4.4

Техніко-економічні показники НДР

№ п/п	показник	Імперативна парадигма	Функційна парадигма
1.	Собівартість, грн.	54646,25	40495,5
2.	Плановий прибуток, грн	30601,9	22677,48
3.	Ціна, грн	85248,15	63172,98
4.	Економічна ефективність	1,56	1,56
5.	Термін окупності, рік	0,64	0,64

4.9. Висновки до розділу

У результаті проведення розрахунків можна зробити висновок: використання функційної парадигми є ефективнішим, тому що витрачається майже удвічі менше часу на розробку програмного продукту ніж при використанні імперативної парадигми програмування, що призводить до зменшення його ціни майже на 25%. Розробка матиме оптимальну економічну ефективність 1,56 і термін окупності становитиме майже рік років (0,64 року). Варто зазначити, що дані розрахунки носять номінальний характер і основна їх мета оцінити приблизну вартість дослідження та створення програмного продукту. Номінальний характер розрахунків зумовлений тим, що даний програмний продукт має дослідницьке призначення.

РОЗДІЛ 5
ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

5.1. Охорона праці

Усі дослідження здійснювалося з дотриманням правил та норм охорони праці і вимог техніки безпеки, що є невід'ємною частиною виконання всіх видів робіт при проведенні даного дослідження. Робоче приміщення та місце має відповідати вимогам щодо охорони праці при організації роботи з візуальними дисплейними терміналами електронно обчислювальних машин (ВДТ). У даному підрозділі розглядаються умови в приміщенні, де розроблявся дипломний проект.

Згідно з санітарними нормами приміщення, що розглядається, має природне і штучне освітлення. Денне (природне) освітлення приміщення відбувається за системою однобічного бічного освітлення. Природне світло проникає у приміщення через три віконні отвори. Також наявні жалюзі з можливістю захисту працюючих від прямого попадання сонячних променів і регулювання рівня освітленості в приміщенні. Вікна приміщення орієнтовані на північний схід. Оскільки будинок розташований у відносній віддаленості від прилеглих будівель, то які-небудь перешкоди природному освітленню розглянутого приміщення відсутні.

Всередині приміщення стіни обклеєні світлими шпалерами, стеля побілена (переважає білий колір), у якості підлогового покриття використаний лінолеум світло – жовтого кольору.

В приміщенні використовується система загального рівномірного штучного освітлення.

У приміщенні маються внутрішні джерела постійного шуму:

– вентилятори блоків ЕОМ;

– принтери;

– дисководи.

що відповідає нормам державних санітарних правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин ДСанПІН 3.3.2.007-98, затверджених постановою Головного державного санітарного лікаря України від 10 грудня 1998 року № 7 [22].

Зовнішніми джерелами шуму і вібрації в приміщенні є проїжджаючі транспортні засоби.

Шум, створюваний усіма перерахованими джерелами, можна кваліфікувати як постійний.

Наявність постійного шуму в робочій зоні приводить до розладу центральної нервової системи і до таких захворювань як неврози, однак фактичний обмірюваний рівень шуму в робочій зоні склав 43 дБА, що задовольняє нормативному рівню шуму (не повинний перевищувати 50 дБА), та не перевищує санітарних норм виробничого шуму, ультразвуку та інфразвуку ДСН 3.3.6.037-99, затверджених постановою Головного державного санітарного лікаря України від 01 грудня 1999 року № 37 [23].

Аналіз стану електробезпеки в робочому приміщенні показав що:

– Всі прилади в кабінеті використовують напругу 220 В.

– Електропроводка захована і ізольована від працівників спеціальним коробом.

– Кожне робоче місце з ПЕОМ обладнане окремими розетками по 220 В.

– Споживачі електроенергії - 4 ПЕОМ + 4візуально дисплейні термінали + 8 світильників (по 4 лампи).

– Відносна вологість повітря – 60%, температура повітря 22-24 °С, струмопровідний пил і хімічно активні речовини в повітрі відсутні.

– Підлога: ізолююча – лінолеум.

Проаналізувавши наведене вище, можемо сказати, що кабінет відноситься до приміщень без підвищеної електронебезпеки.

ПЕОМ, що використовуються в даному кабінеті підключаються до трифазної мережі і мають захисне занулення (за допомогою окремого захисного нульового провідника). Корпуси ВДТ та принтера виготовлені з пластику і не являються струмопровідними. Щодо корпусів самих ПЕОМ, вони виготовлені зі струмопровідного матеріалу, крім передньої панелі, що виготовлена з пластику.

При виконанні робіт по ремонту і обслуговуванню ПЕОМ обслуговуючий персонал зобов'язаний керуватися “Правилами техніки безпеки при експлуатації електроустановок споживачами" [24]. До роботи не допускаються особи, які не пройшли навчання з техніки безпеки. Даний кабінет задовольняє вимоги щодо електробезпеки у приміщенні, в якому встановлені ЕОМ, відображені в НПАОП 0.00-1.28-10 [25].

З огляду на можливість виникнення пожежі слід з’ясувати, які речовини і матеріали можуть горіти. У приміщенні, що розглядається, можуть горіти вироби з дерева, пластмас, тканини і паперу. Горючі рідини, пил та волокна у приміщенні не використовуються і не виділяються. Тому приміщення, що аналізується, відноситься, відповідно до нормативної документації, до зони П-IIа і до категорії пожежної небезпеки В.

Ймовірними причинами виникнення пожару можуть бути несправність електрообладнання (кабелів, розеток), короткі замикання внаслідок виходу з ладу чи експлуатації несправного електроустаткування (ПЕОМ, периферійних пристроїв), порушення правил протипожежної безпеки тощо.

Експлуатація ліній електромережі практично повністю унеможливлює виникнення електричного джерела загоряння в наслідок короткого замикання та перевантаження проводів. Застосовуються дроти з важкогорючою і негорючою ізоляцією.

Для своєчасного попередження пожеж та підвищення оперативності реагування при їх виникненні у приміщенні використовується такий комплекс заходів:

– обов’язковий інструктаж персоналу з питань охорони праці;

– зокрема, правила пожежної безпеки у приміщеннях з ЕОМ;

– заборона використання відкритого вогню у приміщенні;

– наявність системи автоматичної пожежної сигналізації з димовими пожежними оповіщувачами;

– ступінь вогнестійкості будівлі, у якій розташовано приміщення – II;

– наявність шляхів евакуації при виникненні пожежі;

– розміщення схеми евакуації людей при пожежі і ознайомлення з нею персоналу.

Приміщення має один вихід, оскільки в ньому працює менше 25 чоловік. Ширина проходу між робочими місцями у приміщенні перевищує 1 м. Будинок має три виходи – головний і 2 запасних. Коридор між приміщеннями має два виходи на різні сходи, одні з яких ведуть до головного виходу, а другі - до спеціального евакуаційного виходу.

Для гасіння пожежі кожна кімната обладнана ручними вуглекислотними вогнегасниками ВВК-1,4. У загальному коридорі встановлені пінні вогнегасники ВВП. На сходах присутній спеціальний щит пожежного гідранта з відповідним рукавом. Розглянуте приміщення обладнане датчиками централізованої системи пожежної сигналізації. Призначена відповідальна особа, що відповідає за дотримання персоналом вимог пожежної безпеки.

Розроблено план евакуації персоналу і найбільш коштовного устаткування (майна).

Співробітники ознайомлені з порядком і планом евакуації. Отже, шляхи евакуації з приміщення повністю відповідають нормам правила пожежної безпеки в Україні, затверджені наказом Міністерства внутрішніх справ України від 30 грудня 2014 року № 1417 [26].

В даному підрозділі було проаналізовано основні проблеми охорони праці, що можуть виникнути підчас роботи працівника. Було виділено основні вимоги до приміщення, мікроклімату в приміщенні, освітлення та основних ергономічних характеристик.

5.2. Розрахунок потреби ОГД в захисних спорудах і їх оснащення

Оцінка стійкості роботи об’єкта господарювання під час надзвичайних ситуацій (НС) може бути проведена шляхом моделювання уразливості об’єкта до впливу уражаючих факторів НС на основні його елементи на основі використання результатів розрахунків.

У разі впливу ударної хвилі будівлі, споруди, обладнання, комунально-енергетичні мережі об’єкта можуть зазнати різного ступеню руйнування.

Зазначені руйнування поділяються на повні, сильні, середні та слабкі.

При повних руйнуваннях в будівлях і спорудах зруйновано всі основні несучі конструкції та є обваленими перекриття. Відновлення є неможливим. Обладнання, засоби механізації і техніка відновленню не підлягають. На комунально-енергетичних мережах (КЕМ) і технологічних трубопроводах розриви кабелів, руйнування великих ділянок трубопроводів, опор повітряних ліній електропередач і т.п.

У разі сильних руйнувань в будівлях та спорудах мають місце значні деформації несучих конструкцій, зруйновано більшу частину перекрить та стін. Відновлення будівель та споруд є можливим, але недоцільним, оскільки це призведе до фактично нового будівництва з використанням деяких конструкцій, що збереглися. Обладнання і механізми здебільшого зруйновано та значно деформовано. Окремі деталі та вузли обладнання можуть бути використані як запасні частини. На КЕМ і трубопроводах є розриви та деформації на окремих ділянках підземних мереж, деформації опор повітряних ліній електропередач і зв’язку, а також розриви технологічних трубопроводів.

Середні руйнування призводять до пошкодження у будівлях та спорудах в основному не несучих, другорядних конструкцій (легкі стіни, перегородки, дахи, вікна, двері). Можливі тріщини у зовнішніх стінах і вивали в окремих місцях. Перекриття і підвали не зруйновані, частина приміщень придатна до експлуатації. Деформовано окремі вузли обладнання та техніки. Техніка

виходить з ладу і потребує капітального ремонту. На КЕМ деформовано і зруйновано окремі опори повітряних ліній електропередач, є пориви і пошкодження технологічних трубопроводів. Для відновлення об’єкта (елементу), що отримав середні руйнування, необхідно провести його капітальний ремонт, для чого залучаються сили об’єкту.

При слабких руйнуваннях в будівлях і спорудах зруйновано частину внутрішніх перегородок, заповнення дверних і віконних отворів. Обладнання має незначні деформації другорядних елементів. На КЕМ є незначні руйнування і вихід з ладу конструктивних елементів. Для відновлення об’єкта (елементу), який отримав слабкі руйнування, як правило, є потреба у дрібному (поточному) ремонті.

При ядерних і звичайних вибухах основним уражаючим фактором, що діє на виробничі будівлі, споруди, обладнання тощо, є повітряна ударна хвиля, основним параметром якої є надмірний тиск $\Delta P_Ф$

Надлишковий тиск ударної хвилі в залежності від потужності вибуху ядерного боєзапасу, типу вибуху (наземний чи повітряний) та відстані від об’єкту до центру вибуху визначається за допомогою таблиць додатку1.

При вибухові ядерного боєзапасу, відстань від центру вибуху до об’єкту визначається за формулою:

$$R_x = R_r - r_{\text{відх}}, \qquad (5.1)$$

де $R_г$ – відстань від точки прицілювання (центру міста) до об’єкту, км;

$r_{відх}$ – максимально вірогідне відхилення боєзапасу від точки прицілювання, км.

Надмірний тиск на відстані r, що очікується на об’єкті в разі вибуху газоповітряної суміші вуглеводневих продуктів (ГПС), буде максимальним значенням надмірного тиску $\Delta P_Ф$ max і визначається розрахунковим шляхом

виходячи з того, що у разі вибуху газопровідної суміші формується три фізичні зони:

а) Зона I – зона детонаційної хвилі. Вона знаходиться в межах хмари вибуху. Радіус цієї зони можна розрахувати за формулою:

$$r_1 = 17.5\sqrt[3]{Q}, \text{ м} \qquad (5.2)$$

В межах цієї зони надмірний тиск складає ΔP_I=1700 кПа.

б) Зона II – зона дії продуктів вибуху. Ця зона охоплює всю територію, по якій розлетілись продукти ГПС в результаті дії детонації.

Радіус цієї зони становить:

$$r_{II}=1.7r_I. \qquad (5.3)$$

Надмірний тиск в межах зони змінюється від 1350 до 300 кПа і може бути розрахований за формулою:

$$\Delta P_{II} = 1300(\frac{r_1}{r})^3 + 50, \text{ кПа} \qquad (5.4)$$

де r – відстань від центру вибуху до вибраної точки (до об’єкту), м.

в) Зона III – зона дії повітряної ударної хвилі.

Надмірний тиск в зоні III на заданій відстані від центру вибуху r_{III} можна визначати за формулами:

$$\psi = 0{,}24\frac{r_{III}}{r_I}, \qquad (5.5)$$

Спочатку обчислюється відносна величина ,

при ψ <2

$$\Delta P_{III} = \frac{700}{3\left(\sqrt{1+29{,}8\psi^3}-1\right)}, \text{ кПа} \qquad (5.6)$$

при ψ >2 $$\Delta P_{III} = \frac{22}{\psi\sqrt{\log\psi + 0{,}158}}, \text{ кПа} \qquad (5.7)$$

Надмірний тиск $\Delta P_{Ш}$ на будь-якій відстані до ЦВ (r) для заданої маси продукту Q, а також осередку ураження і радіуси кожної зони руйнувань можна визначити за графіком на рис.5.1.

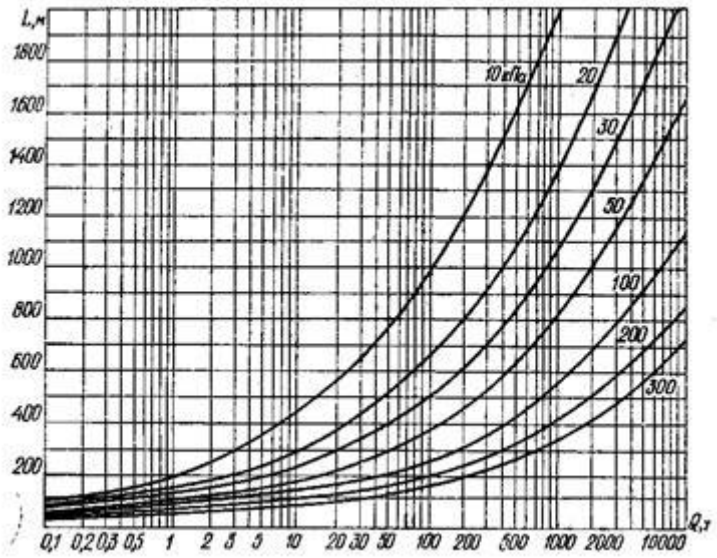

Рис. 5.1. Графік для визначення радіуса зон руйнувань в залежності від маси вуглеводного продукту Q

При вибуху звичайної вибухової речовини (тротилу) надмірний тиск УХ на відстані r (м) від центру вибуху в залежності від маси речовини q (кг) визначається за формулою:

$$\Delta Pф = \left(1{,}05\frac{\sqrt[3]{q}}{r} + 4{,}3\frac{\sqrt[3]{q^2}}{r^2} + 14\frac{q}{r^5}\right)100, \text{ кПа} \qquad (5.8)$$

Розміри осередку ураження при вибуху тротилу практично не відрізняються від розмірів осередку ураження, що утворюється при вибуху газоповітряної суміші вуглеводневого продукту(метану, пропану тощо).

Для інших ВР передбачається коефіцієнт перерахування К. Тоді маса ВР має бути виражена співвідношенням К·q, де К=0,94 - для амоніту, К=0,94 - пікрилової кислоти, К=1,08 –тетрилу, К=1,28 -гексогену.

За кількісний показник стійкості об’єкту до впливу ударної хвилі приймається значення надмірного тиску, при якому будівлі, споруди і обладнання об’єкту зберігаються або отримують слабке і середнє руйнування. Це значення надмірного тиску прийнято вважати межею стійкості об’єкту до ударної хвилі ($\Delta P_ф$ lim).

За межу стійкості будь-якого елементу об’єкту (будівля, споруда, обладнання) приймається мінімальне значення $\Delta P_ф$,при якому цей елемент отримає середнє руйнування.

5.3. Створення комфортних умов праці на промисловому підприємстві, яке може використовувати методи та засоби для функційного програмування мікроконтролерів

Промислове підприємство буде розташоване в офісному приміщенні, де працює 16 осіб. Реальні розміри приміщення становлять: довжина -10 м; ширина - 10 м; висота - 5 м.

Для забезпечення нормальної роботи користувачів ПК вибирається монітор, який відповідає Директиві 90/270 Європейської економічної Комісії "Мінімальні вимоги з охорони праці, які гарантують безпечні умови роботи".

Електронно-променева трубка монітора - це джерело практично всіх видів електромагнітного випромінювання, яке буває іонізуючим (рентгенівське) та неіонізуючим. Іонізуюче випромінювання, впливаючи на клітини організму людини, може викликати їх пошкодження за рахунок утворення іонів. Крім того, при досить сильному опроміненні можливий процес радіолізу (розпаду) води, що міститься в тканинах, а це приведе до утворення короткоживучих радикалів, здатних викликати рак і мутації клітин.

Крім опромінення, при тривалій роботі з ПК можна заробити також так званий КЗС - комп’ютерний зоровий синдром. Цей синдром супроводжується ріжучим болем очей, при цьому спостерігається їх почервоніння і сухість. Нерідко трапляються головні болі, особливо у людей, які багато працюють з набором тексту, з’являється швидка стомлюваність.

Під час роботи за комп’ютером необхідно вжити низку заходів, щоб вплив монітора на здоров’я був якнайменшим.

Монітор має антибликове покриття, що є позитивним фактором, тому що бліки розпорошують увагу і прискорюють втомлюваність організму.

Розмір зерна — 0,26 мм, що є досить непоганим показником. Цей параметр визначає якість картинки і, що дуже важливо - чіткість шрифтів. Якщо монітор "грубозернистий", то шрифти виглядатимуть розпливчастими і нечіткими, що неминуче спричинить високе навантаження на очі при читанні, а в перспективі - прогресуючу короткозорість.

Під час роботи за комп’ютером необхідно дотримувались таких вимог:

– оптимальна відстань від монітору до очей — 50- 70 см;

– кут зору між нормаллю до кінескопа і лінією погляду (не більше 60°);

– кут нахилу клавіатури дорівнює 15і;

– яскравість монітору відносно джерела світла не перевищувала 1:10.

Наприклад, вибраний монітор, який має наступні параметри, задовольняє вимоги наведеного вище нормативного документу:

– діагональ монітора 15 дюймів;

– частота оновлення екрану 100 Гц при роздільній здатності 1024x768;

– максимальна роздільна здатність 1600x1200;

– монітор відповідає стандарту ТСО’99 за електромагнітним випромінюванням;

– плоский екран;

– розмір зерна 0,24 мм.

Монітор встановлюють таким чином, щоб верхній край екрану знаходився на рівні очей. Екран монітору знаходиться від очей користувача на відстані не меншій ніж 60 см. Клавіатура розташовується таким чином, щоб на ній зручно працювати двома руками. Тобто повинна знаходитись на поверхні стола чи спеціальній підставці на відстані 10 - 30 см від краю стола чи підставки. Кут нахилу панелі клавіатури до столу регулюється в межах від 5 до 15 градусів.

На робочому місці мають бути передбачені заходи захисту від можливої дії небезпечних і шкідливих чинників виробництва. Рівні цих чинників не повинні перевищувати граничних значень, обумовлених правовими, технічними і санітарно-технічними нормами.

Робота з комп'ютером характеризується значною розумовою напругою і нервово-емоційним навантаженням операторів, високою напруженістю зорової роботи і достатньо великим навантаженням на м'язи рук при роботі з клавіатурою ЕОМ. Велике значення має раціональна конструкція і розташування елементів робочого місця, що важливе для підтримки оптимальної робочої пози людини-оператора.

В процесі роботи з комп'ютером необхідно дотримувати правильний режим праці і відпочинку. Інакше у персоналу виникає значна напруга зорового апарату з появою скарг на незадоволеність роботою, головні болі, дратівливість, порушення сну, утомленість і хворобливі відчуття в очах, в попереку, в області шиї і руках.

Облаштування робочих місць, обладнаних відеотерміналами, забезпечує:

належні умови освітлення приміщення і робочого місця, відсутність відблисків;

оптимальні параметри мікроклімату (температура, відносна вологість, швидкість руху, рівень іонізації повітря);

належні ергономічні характеристики основних елементів робочого місця,

а також враховує такі небезпечні і шкідливі фактори:

– наявність шуму та вібрації;

– м’яке рентгенівське випромінювання;
– електромагнітне випромінювання;
– ультрафіолетове і інфрачервоне випромінювання;
– електростатичне поле між екраном і оператором;
– наявність пилу, озону, оксидів азоту й аероіонізації.

Заземлені конструкції, що знаходяться в приміщенні (батареї опалення, водопровідні труби, кабелі із заземленим відкритим екраном тощо), надійно захищені діелектричними щитками. Для захисту від надмірної яскравості вікон застосовуються регульовані жалюзі. Для внутрішнього оздоблення приміщень з ВДТ використовуються дифузно-відбивні матеріали з коефіцієнтами відбиття для стелі 0,7- 0,8, для стін 0,5— 0,6. Покриття підлоги матове з коефіцієнтом відбиття 0,3- 0,5. Поверхня підлоги рівна, неслизька, з антистатичними властивостями. Виробниче приміщення обладнане шафами для зберігання документів, полицями, стелажами, тумбами. Щоденно проводиться вологе прибирання. Приміщення оснащене аптечкою першої медичної допомоги. При приміщенні з ВДТ обладнане побутове приміщення для відпочинку під час роботи, кімната психологічного розвантаження, в якій передбачено встановлення пристроїв для приготування й роздачі тонізуючих напоїв, а також місця для занять фізичною культурою.

Згідно ДСанПІН 3.3.2.007- 98, а також, беручи до уваги характер робіт, відповідно до яких, площа підлоги приміщення на одного працівника дорівнює 6 м приймається:

$$S_n = n \cdot S_o \,, \qquad (5.9)$$

де S_o - площа приміщення, що відводиться на одного працівника, м , n — кількість працівників.

Оскільки в приміщенні працює 16 чоловік, тоді необхідна площа для роботи становить 96м^2.

Реальні розміри приміщення становлять: довжина -10м; ширина- 10 м; висота- 5 м.

Тобто площа підлоги приміщення - 100 м , а об’єм - 500 м що відповідає вимогам "Державних санітарних правил та норм роботи з візуальними дисплейними терміналами електронно-обчислювальних машин ДСанПІН 3.3.2.007- 98 ", пункт 2 "Вимоги до виробничих приміщень для експлуатації ВДТ ЕОМ та ПЕОМ", згідно з якими для кожного працівника відводиться 20 м об’єму і 6 м площі.

Робоче місце та розташування всіх його елементів має відповідати антропометричним, фізичним і психологічним вимогам. Наприклад, дотримані наступні основні умови: оптимальне розміщення устаткування, достатній робочий простір, що дозволяє здійснювати всі необхідні рухи і переміщення. Враховані ергономічні вимоги: висота робочої поверхні, розмір простору для ніг, наявність і розміри підставки для документів, можливість різного розміщення документів, відстань від очей користувача до екрану, документа, клавіатури), характеристики робочого крісла, вимоги до поверхні робочого столу, можливість регулювання елементів робочого місця.

Конструкція робочого місця забезпечує підтримання оптимальної робочої пози з такими ергономічними характеристиками: ступні ніг - на підлозі або на підставці для ніг; стегна - в горизонтальній площині; передпліччя - вертикально; лікті - під кутом 70 - 90 град, до вертикальної площини; зап’ястя зігнуті під кутом не більше 20 град, відносно горизонтальної площини, нахил голови - 15 - 20 град, відносно вертикальної площини.

Висота робочої поверхні столу для відеотерміналу знаходиться в межах 680 -800 мм, а ширина - забезпечує можливість виконання операцій в зоні досяжності моторного поля. Розміри столу: висота - 725 мм, ширина - 600 - 1400 мм, глибина -800 - 1000 мм. Робочий стіл має простір для ніг висотою 600 мм, шириною 500 мм, глибиною на рівні колін 450 мм, на рівні витягнутої ноги -650 мм. Робочий стіл для відеотерміналу обладнаний підставкою для ніг шириною

300 мм та глибиною 400 мм, з можливістю регулювання по висоті в межах 150 мм та кута нахилу опорної поверхні - в межах 20 град. Підставка має рифлену поверхню та бортик на передньому краї заввишки 10 мм.

Робоче сидіння підйомно-поворотне, регулюється за висотою, кутом нахилу сидіння та спинки, відстанню спинки до переднього краю сидіння, висотою підлокітників. Регулювання кожного параметра є незалежне, має надійну фіксацію. Хід ступінчастого регулювання елементів сидіння становить для лінійних розмірів 15 — 20 мм, для кутових - 2 - 5 град. Зусилля під час регулювання не перевищує 20 Н. Ширина та глибина сидіння не менші за 400 мм. Висота поверхні сидіння регулюється в межах 500 мм, а кут нахилу поверхні — від 15 град, вперед до 5 град, назад. Поверхня сидіння плоска, передній край - заокруглений.

Висота спинки сидіння становить 300 мм, ширина - 380 мм, радіус кривизни в горизонтальній площині - 400 мм. Кут нахилу спинки регулюється в межах 0-30 град, відносно вертикального положення. Відстань від спинки до переднього краю сидіння регулюється у межах 260 - 400 мм.

Для зниження статичного напруження м’язів рук застосовуються стаціонарні або знімні підлокітники довжиною 250 мм, шириною - 50 мм, що регулюються по висоті над сидінням у межах 230 мм та по відстані між підлокітниками в межах 500 мм.

Поверхня сидіння, спинки та підлокітників напівм’яка, з неслизьким, ненаелектризовуючим, повітронепроникним покриттям та забезпечує можливість чищення від бруду.

Клавіатура розміщується на спеціальній робочій поверхні на відстані 100 - 300 мм від краю, ближчого до працівника. Кут нахилу клавіатури в межах 5 — 15 град.

Робоче місце оснащене рухомим пюпітром (тримачем) для документів, який встановлюється вертикально на тому ж рівні та відстані від очей користувача ПК, що і монітор.

Розміщення принтера забезпечує добру видимість екрану відеотерміналу, зручність ручного керування пристроєм введення-виведення інформації в зоні досяжності моторного поля: по висоті 900 - 1300 мм, по глибині 400 - 500 мм. Для забезпечення високої концентрації уваги під час виконання робіт з високим рівнем напруженості суміжні робочі місця з моніторами та ПК відокремлюються одне від одного перегородками висотою 1,5 м. Щодня перед початком роботи очищається екран монітора від пилу та інших забруднень.

Не допускається виконувати обслуговування, ремонт та налагодження ПК і периферійних пристроїв безпосередньо на робочому місці та зберігати біля ПК папір, дискети, інші носії інформації, запасні блоки, деталі тощо, якщо вони не використовуються для поточної роботи; відключати захисні пристрої, самочинно проводити зміни у конструкції та складі ПК і периферійних пристроїв або їх технічне налагодження; працювати з моніторами, у яких під час роботи з'являються нехарактерні сигнали, нестабільне зображення на екрані. Зони досяжності рук в горизонтальній площині зображені на рис. 5.2.

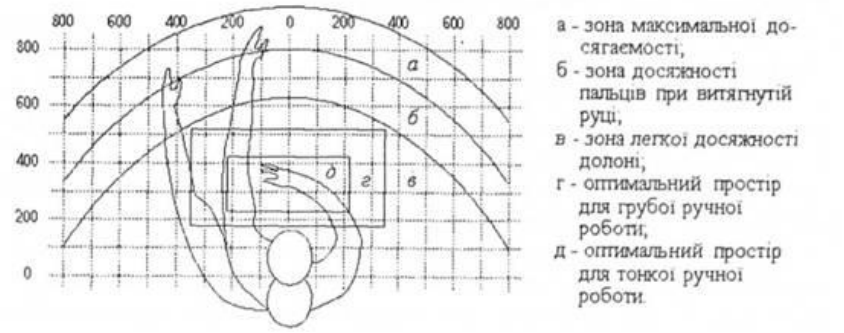

Рис. 5.2. Зони досяжності рук в горизонтальній площині.

Робочі місця розміщені так, що природне світло падає збоку, переважно зліва.

Дотримуються такі вимоги:

– робочі місця розмішуються на відстані не менше 1 м від стін зі світловими прорізами;

– відстань між бічними поверхнями відеотерміналів не менша за 1,2 м;

– відстань між тильною поверхнею одного відеотерміналу та екраном іншого не менша за 2,5 м;

– прохід між рядами робочих місць не менший за 1 м.

Колір приміщення і меблів сприяє створенню сприятливих умов для зорового сприйняття та гарного настрою. Через те, що вікна орієнтовані на північ - забарвлення стін жовтого кольору, підлоги - червонувато-оранжеве. В приміщенні забезпечуються наступні коефіцієнти віддзеркалення: для стелі: 60-70%, для стін: 40-50%, для підлоги: біля 30%. Для інших поверхонь і робочих меблів: 30- 40%.

5.4. Висновки до розділу

У даному розділі було розглянуто підприємство, яке займається методами та засобами функційного програмування мікроконтролерів, було розраховане приміщення для шістнадцятьох людей, які працюватимуть з а комп’ютерами, та розраховані усі необхідні параметри для безпечної та комфортної роботи працівників згідно з санітарними правилами та нормами.

РОЗДІЛ 6
ЕКОЛОГІЯ

6.1. Статистичний аналіз екологічності виробництва

Еколого-економічний аналіз виробництва підприємств-забруднювачів ґрунтується на системі показників та інформації, необхідних для прийняття оптимальних управлінських рішень у сфері раціоналізації природокористування й охорони навколишнього середовища, екологізації та екологічності виробництва.

Екологізація процес послідовного впровадження нової техніки і технології, нових форм організації виробництва, виконання управлінських та інших рішень, які дають змогу підвищити ефективність використання природних ресурсів з одночасним збереженням природного середовища та його поліпшення на різних рівнях. Екологізація економіки цілеспрямований процес перетворення економіки, пов'язаний зі зниженням інтегрального екодеструктивного впливу виробництва і споживання товарів і послуг у розрахунку на одиницю сукупного суспільного продукту. Екологічність виробництва характеризує частку екологічних витрат у сукупних витратах виробництва конкретного виду продукції. Важливим показником екологізації виробництва виступає екологоємність продукції, тобто сукупність екологічних витрат в одиниці вартості продукції [27].

Під оцінкою екологічності виробництва розуміють висновок про рівень екологічності господарської діяльності з урахуванням чинника техногенної безпеки у взаємозв'язку з виробничими ресурсами, умовами і фінансово-економічними результатами господарської діяльності. Можна також сказати, що оцінка екологічності підприємства являє собою його характеристику, отриману в результаті дослідження, і містить висновки про результати екологічної діяльності підприємства, галузі, регіону.

Оцінка екологічності виробництва може бути:

– інструментом обліку, аналізу, планування і регулювання;

– показником еколого-економічного стану господарського об'єкта;

– критерієм порівняльної оцінки екологічності виробництва різних об'єктів;

– показником ефективності прийнятих управлінських рішень у сфері природокористування й охорони навколишнього середовища, а також повноти їх реалізації;

– основою вибору можливих варіантів розвитку екологізації виробництва.

Таким чином, оцінювання екологічної діяльності суб'єктів господарювання, екологічності виробництва і стану соціально-еколого-економічної системи проводиться за одним показником, який характеризує всі сторони функціонування об'єкта. Отримання оцінки екологічної діяльності підприємства та екологічності виробництва на основі системи показників має елемент порівняння, як і комплексна оцінка господарської діяльності. Тобто вона по-суті виступає як порівняльна комплексна або рейтингова оцінка.

Вимоги до аналізу екологічності виробництва. Аналіз екологічності виробництва має задовольняти такі вимоги:

– виражати сутність виробничих та еколого-економічних відносин;

– охоплювати головні сторони виробничо-господарської та екологічної діяльності підприємства;

– використовувати обмежену кількості узагальнених еколого-економічних показників;

– бути еластичною побічно визначати динаміку суспільне необхідних (повних) витрат у сфері природокористування і охорони навколишнього середовища;

– забезпечувати порівнянність показників у часі та просторі;

– вибір показників має визначатися метою регулювання природокористування.

Методологічна основа аналізу екологічності виробництва. Методологічною основою аналізу складових екологічності виробництва виступає індексний метод. За допомогою індексів (у межах від 0 до 1) характеризується наближення того чи іншого показника до необхідного (оптимального).

Процедура аналізу порівняльної оцінки екологічності виробництва виконується у вигляді таких відносно самостійних етапів:

а) поставлення цілей і завдань комплексного аналізу екологічності виробництва, включаючи вибір підприємств і видів їх виробничо-економічної діяльності;

б) обґрунтування та вибір системи еколого-економічних і фінансово-економічних показників;

в) організація збирання вихідної інформації, розрахунку і оцінки окремих показників і вагових коефіцієнтів;

г) вибір об'єкта як бази для порівняння;

д) розроблення алгоритму і розрахунку комплексних показників екологічності виробництва;

е) перевірка адекватності комплексних узагальнених оцінок еколого-економічної ситуації;

ж) аналіз і використання порівняльних комплексних рейтингових оцінок у процесі прийняття управлінських рішень щодо екологізації промислового виробництва.

Реалізація методів порівняльної комплексного рейтингового аналізу передбачає наявність бази порівняння. В економічному аналізі використовуються поняття підрозділу-еталона, підприємства-еталона або об'єкта-еталона. Ряд авторів пропонує використовувати як підприємство-еталон так зване абсолютне підприємство, у якому всі розглянуті показники мають

найкраще значення серед даної сукупності підприємств галузі. У ряді випадків типовим об'єктом порівняння вважається об'єкт, значення показників якого дорівнюють середнім арифметичним або нормативним величинам досліджуваної сукупності підприємств.

6.2. Робота з банками екологічної інформації

Екологічна інформація — це будь-яка інформація про стан навколишнього природного середовища в письмовій, аудіовізуальній, електронній чи іншій матеріальній формі про події, явища, матеріали, факти, процеси і окремих осіб у сфері використання, відтворення та охорони природних ресурсів, природних компонентів та ландшафтів, охорони довкілля та забезпечення екологічної безпеки.

Інформацію про екологічні ситуації на окремих територіях чи об'єктах, про вплив антропогенної діяльності на стан довкілля та здоров'я людей можна отримати за даними екологічної експертизи, екологічний стан на окремих об'єктах описується в екологічних паспортах підприємства.

Банки екологічної статистичної інформації - це вторинна накопичена інформація, певним чином упорядкована чи опрацьована. Найчастіше така інформація подається у вигляді статистичних збірників, щорічників.

Статистичний щорічник України представляє собою накопичену інформацію за ряд років, тобто є по суті банком статистичної інформації про соціально-економічне становище держави. Він складається з таких розділів: національні рахунки, фінанси і кредит, ціни і тарифи, матеріально-енергетичні ресурси, промисловість, інвестиційна та будівельна промисловість, транспорт і зв'язок, торгівля і послуги, зовнішньоекономічна діяльність, структурні зміни в економіці, населення, зайнятість населення, доходи населення, освіта, наука та інформатика, культура і відпочинок, медичне обслуговування, правопорушення,

природні ресурси та охорона навколишнього середовища, міжнародні зіставлення.

Статистичні збірники «Довкілля України», «Довкілля Житомирщини» та аналогічні збірники інших регіонів України представляють собою накопичену інформацію за ряд років, тобто є по суті банком статистичної інформації про екологічне становище держави та різних її областей.

Оперативна, якісна і точна обробка великих масивів статистичної інформації з банків екологічної інформації може бути виконана лише з використанням сучасних засобів обчислювальної техніки. Наявність потужних, надійних і разом з тим простих в експлуатації програмних продуктів статистичного аналізу звільняє дослідника від рутинних операцій, розширює сферу застосування статистичних методів в різних галузях людської діяльності, сприяє появі якісно нових можливостей статистичного аналізу і моделювання даних. Використання пакетів прикладних програм - це єдиний реальний практичний інструмент розв'язування задач багатофакторного кореляційно-регресійного та аналізу в багатовимірному просторі.

Програмне забезпечення статистичних досліджень досить розвинуте. Сучасний ринок програмних продуктів пропонує різноманітні пакети програм для статистичної обробки даних - BMDP, SPSS, SAS, Systat, Minitab, S-Plus, Statgraphics, Statistica

Використання згаданих пакетів програм дає змогу автоматизувати процес статистичного дослідження в таких напрямах:

- створення файлів даних і таблиць;
- групування екологічних даних;
- графічний аналіз екологічних даних;
- розрахунок варіаційних характеристик вибіркових сукупностей;
- аналіз рядів динаміки і прогнозування їх майбутніх рівнів;
- кореляційно-регресійний аналіз;
- багатомірний аналіз.

6.3. Висновки до розділу

У даному розділі було розглянуто як проводити статистичний аналіз екологічності виробництва, як правильно проводити оцінку екологічності підприємства. Також описані способи роботи з банками екологічної інформації.

ВИСНОВКИ

У даній магістерській роботі проаналізовано та описано існуючі парадигми програмування для програмування мікроконтролерів, описано матиматичне забезпечення функційної парадигми програмування. Основні результати та висновки проведених теоретичних та експериментальних досліджень такі:

– проаналізовано імперативну, об’єктно орієнтовану та функційну парадигми програмування, порівняно ці парадигми та визначено переваги та недоліки кожної;

– проаналізовано задачі які стоять перед програмістом при програмуванні мікроконтролерів;

– проаналізовано функційну парадигму та проведено компаративний її аналіз з іншими парадигмами програмування, які застосовуються при програмуванні мікроконтролерів;

– обґрунтовано доцільність використання функційної парадигми;

– проведено верифікацію запропонованих підходів, шляхом створення прототипу;

– розроблено рекомендації, яких необхідно дотримуватись при програмуванні мікроконтролерів з використанням запропонованої парадигми.

Було здійснено економічні розрахунки, спрямовані на визначення економічної ефективності та вартості написання однакових програмних продуктів для мікроконтролерів використовуючи функційну та імперативну парадигми програмування, а також порівняні ці результати.

Розглянуто та описано вимоги з охорони праці та техніки безпеки відповідно до нормативних документів щодо: організації робочого місця, електробезпеки, шуму та вібрації, освітленості, мікроклімату та пожежної безпеки. Розглянуто поставлені питання екології які стосуються магістерської роботи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Alley P. Introductory Microcontroller Programming. Worcester, 2011. 205 p.

2. Introduction to Microcontrollers. 2018. URL: https://www.arrow.com/en/research-and-events/articles/engineering-basics-what-is-a-microcontroller (дата звернення: 07.06.2019).

3. Парадигмы программирования. 2013. URL: https://www.referat911.ru/Programmirovanie-i-kompyutery/paradigmy-programmirovaniya/95293-1866669-place1.html (дата звернення: 15.06.2019).

4. Gabbrielli M., Martini S. Programming Languages: Principles and Paradigms. Bologna, 2010. 440 p.

5. Banatre J., Fradet P., Giavitto J. Unconventional Programming Paradigms. Mont Saint-Michel, 2005. 336 p.

6. Veltkamp R. Programming Paradigms in Graphics. Blake, 2012. 172 p.

7. Watt D. Programming Language Concepts and Paradigms. Upper Saddle River, 1990. 322 p.

8. What, if anything, is a programming paradigm?. 2017. URL: http://www.cambridgeblog.org/2017/05/what-if-anything-is-a-programming-paradigm/ (дата звернення: 03.08.2019).

9. Overview of the four main programming paradigms. 2013. URL: http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overview-section.html (дата звернення: 09.08.2019).

10. IMPERATIVE PROGRAMMING . Chalmers, URL: https://www.eolss.net/Sample-Chapters/C15/E6-45-05-02.pdf (дата звернення: 11.08.2019).

11. Programming Paradigms for Dummies: What Every Programmer Should Know. 2009. URL: https://www.info.ucl.ac.be/~pvr/VanRoyChapter.pdf (дата звернення: 11.08.2019).

12. Richard B. PEARLS OF FUNCTIONAL ALGORITHM DESIGN. Cambridge, 2010. 277 p.

13. Graham H. Programming in Haskell. Nottingham, 2016. 318 p.

14. Petricek T., Skeet J. Real-World Functional Programming: With Examples in F# and C#. Michigan, 2010. 529 p.

15. O'Sullivan B., Goerzen J., Stewart D. Real World Haskell. Sebastopol, 2008. 671 p.

16. Функциональное программирование. URL: http://pv.bstu.ru/flp/fpLectures.pdf (дата звернення: 20.08.2019).

17. Bird R. Thinking Functionally with Haskell. Cambridge, 2014. 344 p.

18. Functional Reactive Programming for the Arduino. URL: http://www.juniper-lang.org/ (дата звернення: 25.08.2019).

19. frp-arduino. URL: https://github.com/frp-arduino/frp-arduino (дата звернення: 28.08.2019).

20. Audio Processing using Haskell. Bremen, 2004. URL: https://pdfs.semanticscholar.org/3119/84d20c462c84289bc2b319a1aaef38aab266.pdf (дата звернення: 02.09.2019).

21. Data Stream Algorithms. 2009. URL: https://www.cs.mcgill.ca/~denis/notes09.pdf (дата звернення: 04.09.2019).

22. Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин ДСанПІН 3.3.2.007-98. 1998. URL: https://zakon.rada.gov.ua/rada/show/v0007282-98 (дата звернення: 10.10.2019).

23. Санітарні норми виробничого шуму, ультразвуку та інфразвуку ДСН 3.3.6.037-99. 1999. URL: https://zakon.rada.gov.ua/rada/show/va037282-99 (дата звернення: 10.10.2019).

24. Наказ про затвердження Правил безпечної експлуатації електроустановок споживачів. URL: https://zakon.rada.gov.ua/laws/main/z0093-98 (дата звернення: 11.10.2019).

25. Наказ Про затвердження правил охорони праці під час експлуатації електронно-обчислювальних машин. 1998. URL: https://dnaop.com/html/31562/doc-%D0%9D%D0%9F%D0%90%D0%9E%D0%9F_0.00-1.28-10 (дата звернення: 11.10.2019).

26. Наказ Міністерства внутрішніх справ України «Про затвердження Правил пожежної безпеки в Україні». 2014. URL: https://zakon.rada.gov.ua/laws/show/z0252-15 (дата звернення: 11.10.2019).

27. Тарасова В. В. Екологічна статистика. Київ, 2008. 391 с.

Додаток А

Тези конференцій

Рис. А.1. копія титульної сторінки збірника тез конференції

Рис. А.2. копія сторінки змісту збірника, на якій зазначено прізвище автора магістерської роботи, назву тези та посилання на сторінку, де теза розміщена

II Міжнародна студентська науково - технічна конференція
"ПРИРОДНИЧІ ТА ГУМАНІТАРНІ НАУКИ. АКТУАЛЬНІ ПИТАННЯ"

УДК 004.41
Судомир В. - ст. гр. СІм-52
Тернопільський національний технічний університет імені Івана Пулюя

ВИКОРИСТАННЯ ФУНКЦІЙНОЇ ПАРАДИГМИ ПРИ ПРОГРАМУВАННІ МІКРОКОНТРОЛЕРІВ

Науковий керівник: к.т.н., доцент Луцків А. М.

Sudomyr V.
Ternopil Ivan Puluj National Technical University

USE OF FUNCTIONAL PARADIGM IN PROGRAMMING MICROCONTROLLERS

Supervisor: Candidate of technical sciences, docent Lutskiv A. M.

Ключові слова: функційна парадигма, мікроконтролер;
Keywords: functional paradigm, microcontroller.

У контексті бурхливого розвитку технології IoT дедалі більше створюється програм для мікроконтролерів.

На сьогодні при створенні програмного забезпечення, домінуючою є процедурна та іноді об'єктно об'єктно-орієнтована парадигми. Проте з урахуванням таких факторів як: робота мікроконтролерів у поєднанні з різними датчиками, необхідністю економії процесорного часу, написання легко читабельного коду та легкого і ефективного паралельного програмування для багатоядерних мікроконтролерів, оптимальним є використання функційної парадигми програмування.

Функційне програмування - це парадигма програмування, яка суттєво відрізняється від імперативної та об'єктно-орієнтованої парадигм програмування. Функційні програми, як правило, уникають станів та зміни даних, таких як встановлення змінної. Фактично, функційне програмування прагне використовувати підхід з одним ініціюванням тобто використовуються так звані незмінювані значення("immutable"). Функційна парадигма має багато корисних концепцій таких як : рекурсії та хвостові рекурсії, які замінюють звичайні цикли; чисті функції, в яких повернені данні залежать тільки від вхідних даних; функцій вищих порядків, які можуть приймати в якості аргументів, а також повертати як результат обчислень інші функції; лінивні обчислення в яких значення не потрібно обчислювати, якщо вони не будуть використовуватися. Таким чином функційне програмування не тільки забезпечує кращу масштабованість компактного, читабельного та ефективного коду. Тому такий вид програмування успішно використовується в широкому спектрі застосувань, тому що використання мов високого рівня є більш ефективним використанням часу для створення П.З., але часто може бути менш ефективним у використанні обчислювальних ресурсів. Тому функційна парадигма не у всіх випадках буде ефективною для програмування мікроконтролерів.

На сьогоднішній день існують лише декілька інструментів для функційного програмування мікроконтролерів різними мовами програмування, одними з таких інструментів є Juniper та Copilot.

Juniper підтримує багато ознак типових для функційних мов програмування, включаючи алгебраїчні типи даних, записи, відповідність шаблону, незмінні структури даних, параметричні поліморфні функції і анонімні функції (lambda). Деякі імперативні концепції програмування також присутні в Juniper, наприклад, для циклів "while", "do while", можливість позначити змінні як "mutable" і змінні посилання.

Copilot – це предметно-орієнтована мова на Haskell, яка компілюється в embedded C. Copilot подібний до мов типу Luster. Copilot містить інтерпретатор, декілька back-end компіляторів та інші засоби.

Рис. А.3. копія тексту тези зі збірника

Додаток Б

Приклад однакових програми написані з використанням різних парадигм

Лістинг коду програми, яка зчитує показники терморезистора та відображає їх на РК дисплеї для мікроконтролера Arduino на мові Haskell:

```
import Arduino.Uno
import qualified Arduino.Library.LCD as LCD
main = compileProgram $ do
     tick <- def clock
     digitalOutput pin13 =: tick ~> toggle
     setupLCD [ bootup ~> mapSMany (const introText), timerDelta ~>
mapSMany statusText]
     introText :: [Expression LCD.Command]
     introText = concat [ LCD.position 0 0 , LCD.text "FRP Arduino"]
     statusText :: Expression Arduino.Uno.Word -> [Expression
LCD.Command]
     statusText delta = concat[ LCD.position 1 0, LCD.text
(createVariableTick a0)]
     setupLCD :: [Stream LCD.Command] -> Action ()
     setupLCD streams = do
          LCD.output rs d4 d5 d6 d7 enable =: mergeS streams
          where
             rs     = digitalOutput pin3
             d4     = digitalOutput pin5
             d5     = digitalOutput pin6
             d6     = digitalOutput pin7
             d7     = digitalOutput pin8
             enable = digitalOutput pin4
createVariableTick :: AnalogInput -> Stream ()
createVariableTick limitInput = accumulator limitStream timerDelta
where
     limitStream :: Stream Arduino.Uno.Word
     limitStream = analogRead limitInput ~> mapS analogToLimit
```

```
analogToLimit :: Expression Arduino.Uno.Word -> Expression Arduino.Uno.Word
analogToLimit analog = log(((10240000/analog)-10000))
analog = ((1/(0.001129148 + (0.000234125 + (0.00000008767 * analog * analog)) * analog)) - 273.15)
```

Аналог попередньої програми на мові C++:

```
#include <LiquidCrystal_I2C.h>
#include <math.h>
LiquidCrystal_I2C lcd(0x3F, 2, 1, 0, 4, 5, 6, 7, 3, POSITIVE);
double Thermister(int RawADC) {
  double Temp;
  Temp = log(((10240000 / RawADC) - 10000));
  Temp = 1 / (0.001129148 + (0.000234125 + (0.0000000876741 * Temp * Temp))* Temp);
  Temp = Temp - 273.15;   // Kelvin to Celcius
  return Temp;
}
void setup()
{
  lcd.begin(20, 4);
  Serial.begin(9600);
}
void loop()
{
  lcd.setCursor(0, 0);
  lcd.print("c++ arduino");
  lcd.setCursor(1, 0);
  lcd.print(float(Thermister(analogRead(0))));
}
```