МІЖНАРОДНИЙ ГУМАНІТАРНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, програмної інженерії та комп'ютерних наук
Кафедра інформаційних технологій

Пояснювальна записка

до кваліфікаційної роботи
першого (бакалаврського) рівня

на тему **РОЗРОБКА КОМПІЛЯТОРА СТРОГО-ТИПІЗОВАННОЇ МОВИ ПРОГРАМУВАННЯ ДЛЯ КОМАНДНОЇ ОБОЛОНКИ BASH У СЕРЕДОВИЩІ UNIX-ПОДІБНИХ ОПЕРАЦІЙНИХ СИСТЕМ**

Виконав: студент 4 курсу, групи КІ - 4
спеціальності
123 Комп'ютерна інженерія

______________Мемеге Деніз Емре________

Керівник__Григор'єва Т. І.__________

Рецензент__Русу О. Л.______________

Одеса – 2023 р.

Д О В І Д К А

кафедри ІТ про виконану бакалаврську роботу

студента 4 курсу ФКПІ групи КІ- 4

Мемеге Деніз Емре

на тему Розробка компілятора строго-типізованної мови програмування для командної оболонки BASH у середовищі UNIX-подібних операційних систем

Висновок нормоконтролера пояснювальна записка до кваліфікаційної роботи виконана з незначними порушеннями ДСТУ в т.ч. внутрішнього розпорядку ...

Нормоконтролер викл. каф. ІТ ______ 30.06.23 І.В. Клименко
(науковий ступінь, вчене звання, посада) (підпис, дата) (і. б. прізвище)

Висновок відповідального за наявність академічного плагіату згідно з сертифікатом ID унікальність підтверджено

Відповідальна особа викл. каф. ІТ ______ 30.06.23 І.В. Клименко
(науковий ступінь, вчене звання, посада) (підпис, дата) (і. б. прізвище)

Попередня експертиза (захист) бакалаврської роботи
(бакалаврської роботи чи магістерської роботи)

студ. Мемеге Деніз Емре проведена “ 30 ” 06 2023 р.
(прізвище і б.)

Висновки Виконана робота відповідає завданню, якість роботи задовільна. У роботі розроблено стандарт нової мови програмування що вирішує недоліки мови програмування BASH. Також у межах роботи було розроблено компілятор до цього стандарту. Цей компілятор опрацьовує програмний код написаний мовою програмування SBASH. Виконана робота бакалавра відповідає діючим стандартам та рекомендується до захисту в ЕК

Члени комісії ______ к.т.н., доц. Соловська І.М.
(підпис) (науковий ступінь, вчене звання,посада, прізвище і.б.)

______ к.т.н., доц. Григор'єва Т.Л.
(підпис) (науковий ступінь, вчене звання, посада, прізвище і.б.)

______ к.т.н., доц. Ільїна Л.Г.
(підпис) (науковий ступінь, вчене звання, посада, прізвище і.б.)

МІЖНАРОДНИЙ ГУМАНІТАРНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, програмної інженерії та комп'ютерних наук

Кафедра інформаційні технології

Освітній ступінь бакалавр

Галузь знань 12 Інформаційні технології

Спеціальність 123 Комп'ютерна інженерія

ЗАТВЕРДЖУЮ
Завідуючий кафедрою ІТ
к.т.н., доц.
Т.І. Григор'єва
"7" 04 2023 року

З А В Д А Н Н Я

НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТУ

1. Тема роботи Розробка компілятора строго-типізованної мови програмування для командної оболонки BASH у середовищі UNIX-подібних операційних систем
керівник роботи Григор'єва Тетяна Ігорівна, к.т.н., доцент каф. ІТ
затверджені наказом закладу вищої освіти від

2. Строк подання студентом роботи 9 червня 2023 року

3. Вхідні дані до роботи Книга Advanced compiler design and implementation. (1997) за авторством S. Muchnick; Книга Lex and yacc. (1997) за авторством John R. Levine, John Mason, John R Levine, B.A., Ph.D., Tony Mason, Doug Brown, John R. Levine, Paul Levine.

4. Зміст розрахунково-пояснювальної записки 1 АНАЛІЗ ТА ВИРІШЕННЯ ПРОБЛЕМИ РОЗРОБКИ BASH СКРИПТІВ У EMBEDDED СИСТЕМАХ; 2 СПЕЦИФІКАЦІЯ ВИМОГ ДО МОВИ ПРОГРАМУВАННЯ SBASH; 3 ПРОЕКТУВАННЯ МОВИ ПРОГРАМУВАННЯ SBASH; 4 ПРОГРАМНА РЕАЛІЗАЦІЯ КОМПІЛЯТОРА SBASH; 5 ЕКСПЛУАТАЦІЯ ОБЛАСТЬ ЗАСТОСУВАННЯ ТА ПОДАЛЬШИЙ РОЗВИТОК.

5. Перелік графічного матеріалу (з зазначенням обов'язкових креслень)

Слайд 1 – Додаток А, Рисунок 5 – Загільні відомості – оболонка терміналу BASH

Слайд 2 – Додаток А, Рисунок 7 – Мова програмування SBASH – загальний огляд

Слайд 3 – Додаток А, Рисунок 9 – Мова програмування SBASH – нові можливості

Слайд 4 – Додаток А, Рисунок 11 – SBASH компілятор

Слайд 5 – Додаток А, Рисунок 14 – Приклад скомпільованого коду – SBASH у BASH

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 15 березня 2023

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Аналіз предметної області, пошук аналогів, дослідження проблематики задачі у межах даної теми.	18.03.2023 – 23.03.2023	
2	Розробка базового стандарту мови програмування, планування основних особливостей мови програмування.	24.03.2023 – 28.03.2023	
3	Розробка синтаксису мови програмування.	29.03.2023 – 6.04.2023	
4	Розробка архітектури компілятора.	7.04.2023 – 12.04.2023	
5	Імплементація компілятора: загальні конструкції.	13.04.2023 – 27.04.2023	
6	Імплементація компілятора: імплементація лексичного аналізатора.	28.04.2023 – 4.05.2023	
7	Імплементація компілятора: імплементація граматичного аналізатора.	5.05.2023 – 12.05.2023	
8	Імплементація компілятора: імплементація семантичного аналізатора.	13.05.2023 – 20.05.2023	
9	Тестування програмного продукту.	21.05.2023 – 26.05.2023	

Студент __________ __________
(підпис)

Керівник роботи Григор'євський І. __________
(підпис)

ВІДГУК КЕРІВНИКА

на бакалаврську роботу студента Мемеге Деніз Емре.

на тему: «Розробка компілятора строго-типізованної мови програмування для командної оболонки bash у середовищі unix-подібних операційних систем»

Бакалаврська робота студента Мемеге Деніз Емре присвячена розробці стандарту строго-типізованої мови програмування та компілятора із цього стандарту до BASH скрипту у межах UNIX-подібних систем.

У бакалаврській роботі розроблено стандарт нової мови програмування, що вирішує недоліки мови програмування BASH. Таку мову програмування названо SBASH. Також у межах роботи було розроблено компілятор до цього стандарту. Ций компілятор опрацьовує програмний код написаний мовою програмування SBASH та компілює його у скрипт написаний мовою програмування BASH. Таким чином розроблений проект одночасно вирішує визначені проблемні сторони мови програмування BASH, що унеможливлюють розробку великих скриптів на ній. А також зберігає її переваги, такі як найвища росповсюдженість серед всіх UNIX-подібних операційних систем.

Завдання на бакалаврську роботу виконано. Під час виконання роботи студент Мемеге Деніз Емре показав уміння користуватися навчальною технічною літературою та навиками програмування.

Робота студента Мемеге Деніз Емре відповідає вимогам щодо кваліфікаційних робіт бакалаврського рівня та заслуговує оцінки «задовільно».

Студент Мемеге Деніз Емре заслуговує присвоєння кваліфікації бакалавр з комп'ютерної інженерії за заявленою спеціальністю 123 Комп'ютерна інженерія.

Керівник
к.т.н., доц. кафедри
інформаційних технологій Т.І.Григор'єва

РЕЦЕНЗІЯ

на бакалаврську роботу студента Мемеге Деніз Емре.

з теми: «Розробка компілятора строго-типізованної мови програмування для командної оболонки BASH у середовищі UNIX-подібних операційних систем»

Бакалаврська робота виконана на 83 с. текстової частини та містить відповідні розділи згідно з завданням на бакалаврську роботу. Робота складається з 5 основних розділів, вступу, висновків та рекомендацій, переліку джерел посилання та двох додатків. У вступі студент обґрунтовує вибір теми та її актуальність. Тема роботи актуальна, стосується сучасних проблем розробки. У бакалаврській роботі розроблено стандарт нової мови програмування, що вирішує недоліки мови програмування BASH. Таку мову програмування названо SBASH. Також у межах роботи було розроблено компілятор до цього стандарту. Цей компілятор опрацьовує програмний код написаний мовою програмування SBASH та компілює його у скрипт, написаний мовою програмування BASH. Таким чином, розроблений проект одночасно вирішує визначені проблемні сторони мови програмування BASH, що унеможливлюють розробку великих скриптів на ній. А також зберігає її переваги, такі як найвища розповсюдженість серед всіх UNIX-подібних операційних систем.

За результатами проведеного аналізу студ. Мемеге Деніз Емре провів дослідження проблеми розробки на мові програмування BASH у embedded системах, а також розробив нову мову програмування SBASH із однойменним компілятором до неї.

Текстова частина бакалаврської роботи викладена послідовно, чітко, технічно грамотно. Робота виконана відповідно до завдання.

До недоліків роботи варто віднести:

- не в повному обсязі проведено аналіз подібних мов програмування;
- немає суттєвих пояснень процесу компіляції.

Зазначені недоліки суттєво не знижують якості виконаної роботи.

Робота студента Мемеге Деніз Емре відповідає вимогам щодо кваліфікаційних робіт бакалаврського рівня та заслуговує оцінки «задовільно».

Студент Мемеге Деніз Емре заслуговує присвоєння кваліфікації бакалавр з комп'ютерної інженерії за заявленою спеціальністю 123 Комп'ютерна інженерія.

Рецензент

к.т.н., доцент кафедри комп'ютерних наук О. П. Русу

Имя пользователя:
Анна Серединко

Дата проверки:
29.06.2023 08:25:16 EEST

Дата отчета:
29.06.2023 08:28:25 EEST

ID проверки:
1015701911

Тип проверки:
Doc vs Internet + Library

ID пользователя:
100001433

Название файла: Мемеге_Деніз_Емре_Диплом

Количество страниц: 85 **Количество слов:** 15292 **Количество символов:** 119996 **Размер файла:** 2.75 MB **ID файла:** 1015345525

2.35%
Совпадения

Наибольшее совпадение: 1.37% с источником из Библиотеки (ID файла: 1015327943)

1.96% Источники из Интернета | 426 .... Страница 87

1.45% Источники из Библиотеки | 56 .... Страница 88

0% Цитат

Исключение цитат выключено

Исключение списка библиографических ссылок выключено

0% Исключений

Нет исключенных источников

Модификации

Обнаружены модификации текста. Подробная информация доступна в онлайн-отчете.

Замененные символы | 2

РЕФЕРАТ

Текстова частина бакалаврської роботи: 83 с., 48 рис., 4 табл., 2 додатка, 12 джерел.

Об`єкт розробки – строго-тіпізована мова програмування та компілятор.

Мета роботи – розробити стандарт строго-типізованої мови програмування та компілятора із цього стандарту до BASH скрипту у межах UNIX-подібних систем.

У бакалаврській роботі розроблено стандарт нової мови програмування, що вирішує недоліки мови програмування BASH. Таку мову програмування названо SBASH. Також у межах роботи було розроблено компілятор до цього стандарту. Цей компілятор опрацьовує програмний код написаний мовою програмування SBASH та компілює його у скрипт написаний мовою програмування BASH. Таким чином розроблений проект одночасно вирішує визначені проблемні сторони мови програмування BASH, що унеможливлюють розробку великих скриптів на ній. А також зберігає її переваги, такі як найвища росповсюдженість серед всіх UNIX-подібних операційних систем.

МОВА ПРОГРАМУВАННЯ, КОМПІЛЯТОР, BASH, C++, EMBEDDED РОЗРОБКА, UNIX, DEV-TOOLS

ABSTRACT

The text part of the bachelor's work: 83 pp., 48 figures, 4 tables, 2 appendices, 12 sources.

The object of development is a strictly typed programming language and a compiler.

The goal of the work is to develop a standard for a strictly typed programming language and a compiler from this standard to a BASH script within UNIX-like systems.

In the bachelor's work, a new programming language standard was developed, which solves the shortcomings of the BASH programming language. Such a programming language is called SBASH. A compiler for this standard was also developed as part of the work. This compiler processes program code written in the SBASH programming language and compiles it into a script written in the BASH programming language. The project developed in this way simultaneously solves certain problematic aspects of the BASH programming language, which make it impossible to develop large scripts on it. And also retains its advantages, such as the highest prevalence among all UNIX-like operating systems.

PROGRAMMING LANGUAGE, COMPILER, BASH, C++, EMBEDDED DEVELOPMENT, UNIX, DEV-TOOLS

ЗМІСТ

ВСТУП 12

1 АНАЛІЗ ТА ВИРІШЕННЯ ПРОБЛЕМИ РОЗРОБКИ BASH СКРИПТІВ У EMBEDDED СИСТЕМАХ 14

1.1 Необхідність розробки на мові програмування BASH для embedded систем .. 14

1.2 Проблеми розробки на мові програмування BASH 15

1.3 Вирішення проблеми та пошук аналогів 19

1.4 Методологія розробки нової мови програмування 21

2 СПЕЦИФІКАЦІЯ ВИМОГ ДО МОВИ ПРОГРАМУВАННЯ SBASH 24

2.1 Глосарій 24

2.2 Мова програмування SBASH 26

2.3 Основні відмінності мови програмування SBASH та її особливості 28

2.4 Функції у мові програмування SBASH 34

2.5 Головна специфікація компілятора мови програмування SBASH 36

2.6 Технології до застосування у реалізації компілятора мови програмування SBASH 38

3 ПРОЕКТУВАННЯ МОВИ ПРОГРАМУВАННЯ SBASH 40

3.1 Проектування базових конструкцій 41

3.2 Проектування конструкцій тіла функції 48

3.3 Базова структура програми 55

4 ПРОГРАМНА РЕАЛІЗАЦІЯ КОМПІЛЯТОРА SBASH 57

4.1 Глобальна архітектура компілятора 57

4.2 SDK компілятора 58

4.3 Архітектура ядра компілятора 60

4.4 Процес компіляції 70

4.5 Загальний огляд тестування 71

5 ЕКСПЛУАТАЦІЯ ОБЛАСТЬ ЗАСТОСУВАННЯ ТА ПОДАЛЬШИЙ РОЗВИТОК 73

5.1 Використання компілятора SBASH 73
5.2 Поточні ліміти компілятора 76
5.3 Подальший розвиток, та його методологія 77
ВИСНОВКИ ТА РЕКОМЕНДАЦІЇ 81
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 83
Додаток А 84
Додаток Б 92

ВСТУП

На сьогодення галузь інформаційних технологій має велику кількість різних напрямків. Вони різняться як проблемами які вирішують програмні продукти розроблені у межах цих галузей, так і технологіями у них використовуваними. Однією із найскладніших (з точки зору розробки) серед них є embedded (від англ. «вбудована») розробка, тобто розробка програмних продуктів, які будуть використовуватись на обладнанні що має жорсткі обмеження щодо обладнання на якому воно буде використовуватись. Зазвичай, це системи працюючі на SOC (від англ. «system on chip» - система на чіпі).

Embedded програмні продукти [1], це, зазвичай, ті які найменше помічає людство у повсякденному використанні, такі як вбудовані екрани у холодильник, що демонструють стан та наявність продуктів у ньому, елементи керуючі світлом, вентиляцією, сигналізацією у системах smart house (від англ. «розумний дім»), чи навігаційні та розважальні системи у новітніх моделях автомобілів.

Сучасна embedded розробка, має набагато меньше спільного із тим, чим вона була десятиріччя тому, а саме сучасне обладнання набагато сильніше що дозволяє використовувати комплексні операційні системи як базу для розробки цих програмних продуктів.

Саме операційні системи сім’ї UNIX використовуються як ядро embedded систем [2], на базі якого розробляються складні програмні продукти із великою кількістю залежностей та використаних технологій (залежність модулів програмного продукту один від одного, використання мережі інтернет, гео-датчиків, датчиків температурі, вологості повітря, тиску, тощо). Операційні системи побудовані на ядрі Linux є найпопулярнішими серед зазначених.

Під час розробки таких програмних продуктів у межах максимальних обмежень (які притаманні embedded розробці) виникає необхідність створення та використання devtools (від англ. «development tools» - «інструменти розробки»), тобто невеликих програм, що допоможуть працюючим над програмним комплексом ефективно виконувати свою роботу. Зазвичай, такі програми пишуть на скриптових мовах програмування (хоча це не справедливо для всіх devtools). Ця необхідність може виникати через зручність редагування коду програми без її компіляції.

Одним із обмежень embedded розробки може бути використання деяких програм, через відсутність чи недостатню потужність (розмір) ресурсів для їх підтримки, таких як:

- постійна пам’ять;
- оперативна пам’ять;
- центральний процессор;
- інколи графічний процесор.

Такими програмамми бувають і самі інтерпретатори скриптових мов програмування. Саме це є одним із головних аргументів щоб розробляти, якщо не всі, то більшість devtools на скриптовій мові програмування яка є вбудованою у цільову операційну систему. Самою популярною оболонкою терміналу UNIX подібних систем є BASH, і однойменна скриптова мова програмування.

Мова програмування BASH має довгу історію розробки (розроблена у 1989 році). Через надмірну спрямованість у портативність, велика кількість конструкцій мови програмування не має спільного вигляду і може мати несподівану для користувача поведінку. У купі із динамічну типізацію це призводить до складності визначення помилки, що зазвичай виникає у момент виконання програми.

Одним із рішенням цієї проблеми є ця дипломна робота, що пропонує альтернативну мову програмування, що буде проста та інтуїтивна у використанні. А результатом компіляції програми написаної цією мовою програмування буде BASH script (від англ. «сценарій» – так називають програми написані для інтерпретованної мови програмування).

У той самий час, ця мова програмування буде позбавлена основних проблем мови програмування BASH, а саме буде мати:

- строгу типізацію;
- складні структури даних із можливістю спадкування;
- структури даних відомі як перерахування;
- компільованною.

Саме такі ознаки дозволяють звести до мінімуму виникнення помилок у BASH script під час виконання, та вирішити їх на етапі компіляції, а з іншого боку, це дозволить зберегти портативність програм написаних на мові програмування BASH, без необхідності у додаткових ресурсах embedded системи.

1 АНАЛІЗ ТА ВИРІШЕННЯ ПРОБЛЕМИ РОЗРОБКИ BASH СКРИПТІВ У EMBEDDED СИСТЕМАХ

1.1 Необхідність розробки на мові програмування BASH для embedded систем

Сучасна embedded розробка має все менше спільного із початковим терміном. Основною різницею стає можливість використання повноцінних операційних систем, які виконують роботу з менеджментом програмних процесів, забезпеченням обробки багатьох сигналів із різних датчиків, шин даних, забезпечують базову роботу із обладнанням низького рівня, тощо.

Найпопулярнішими системами embedded розробки є сім’я операційних систем UNIX, що стала такою через безкоштовний для комерційного використання Linux (назва системи за однойменною назвою ядра цієї системи), що є аналогом системи UNIX. Зазвичай Linux називають GNU/Linux, через те, що більша частина програм, що використовуються та знаходяться у дистрибутивах (дистрибутив, це збірка програм та ядра системи) Linux є програми написані у межах проекту GNU. У назві даної роботи зазначено, що розробка проводиться у межах UNIX подібних систем, саме через те, що проект GNU був розроблений як безкоштовна альтернатива операційній системі UNIX, та має багато спільних елементів. Відносніть до UNIX операційної системи можно побачити у самій назві проекту, адже GNU це рекурсивний акронім, що означає фразу «GNU’s NOT UNIX».

Розробка програмних комплексів під embedded системи зазвичай використовує компільовані мови програмування, адже вони швидко працюють, дають велику свободу використання ресурсів обладнання, та дають можливість перевірити велику кількість помилок на етапі компіляції. Ці особливості дають можливість написати більш надійну, та швидку у виконанні програму без додаткових зусиль на використання стороннього програмного забезпечення, чи написання тестів, ручного тестування, що призводить до додаткових фінансових витрат, тощо.

Попри те, розробка devtools ведеться здебільш на інтерпретованих мовах програмування, через іншу специфіку. Бо зміна основного програмного продукту, може вимагати зміни поведінки devtools, що потрібно зробити швидко та просто.

Треба зазначити, що у корпоративних проектах, для впровадження використання нової технології на цільовому обладнанні, треба отримати дозвіл на

це від керівництва, що займає багато часу, через бюрократичні проблеми. Тим паче, що devtools можуть бути ініціативою розробника у проекті, щоб оптимізувати свою роботу, але може не оплачуватися із сторони замовника програмного продукту. Але все це не має ваги, якщо цільове обладнання не має достатньо потужностей щоб використовувати ті чи інші додаткові програми, адже у embedded розробці все ще суворо контролюється використання ресурсів, що є її ознакою. Такі особливості розробки логічно призводять до необхідності використання вже вбудованих у дистрибутив системи рішень.

Однією із програм проекту GNU є оболонка терміналу BASH із однойменною мовою програмування. Мова програмування BASH є дуже портативною, хоча має багато команд та конструкцій, що використовуються тільки у декільках дистрибутивах. Саме проблема підтримки одних та відсутність підтримки інших команд у різних дистрибутивах називається bashism [3], та є однією із тих, зо вирішує дана робота.

Маючи таку відмінність як портативна мова програмування, що працює здебільш на всіх компьютерах під керуванням UNIX подібних систем, та відсутність у додаткових програмних та обчислювальних ресурсах, BASH стає ідеальним рішенням у розробці devtools для embedded систем.

1.2 Проблеми розробки на мові програмування BASH

Як було зазначено вище, однією із найпопулярніших проблем сучасної форми BASH, є bashism, що переслідує її вже десятиріччя. Вирішенням цієї проблеми може бути тільки відмова від старих стандартів, та прийняття нового єдиного стандарту. Колись таке рішення може бути прийняте, але на даний чай, занадто велика кількість програм вже написані на BASH, та зміна стандарту може призвести до несумісності старих скриптів із новим інтерпретатором. Через це, старі стандарти все ще використовуються і у нових скриптах, та призводять тільки повтору вже давно відомих проблем кожний раз при написанні на мові програмування BASH.

Розглянемо декілька конструкцій написаних на мові програмування BASH, які мають неоднозначне трактування чи додаткові «підводні камені» у їх використанні, або просто не зручні у використанні.

Першим що зустрічається у BASH і що заважає комфортному використанню мови програмування це призначення змінній якого-небудь літералу, чи значення

іншої змінної. На Рисунку 1.1, нижче приведено приклад правильного (на рядку 1) призначення змінній значення і неправильне (на рядках 2-4).

```
1 string_var1="text"      #OK
2 string_var2 ="text"     #Error
3 string_var3= "text"     #Error
4 string_var4 = "text"    #Error
```

Рисунок 1.1 - BASH код, операція присвоювання

Незручність написання цього виразу об’єктивна, якщо зазначити, що у найпопулярніших мовах програмування таких як Java, C, C++, Python, Javascript, та інші, - немає значення кількість пробілів між змінною та оператором присвоювання.

Друге, що не дає комфортно писати код на мові програмування BASH є різні форми операторів порівняння у конструкції умовного переходу, наприклад у таких, як «if». На Рисунку 1.2 нижче приведено код, де порівнюються на тотожність значення змінні із числовим значенням, та текстовим.

```
1 integer_var1=1
2 integer_var2=2
3 string_var1="text 1"
4 string_var2="text 2"
5
6 if[ $integer_var1 -eq $integer_var2 ];
7 # if statement code here...
8 if[ "$string_var1" = "$string_var2" ];
9 # if statement code here...
```

Рисунок 1.2 - Різні операторів порівняння

.

Треба зазначити, що саме різні форми дужок використаних самого оператору може надавати додаткові можливості (див. Рисунок 2.3), наприклад такі як порявняння рядка із regex (від англ. «регулярний вираз»).

```
1 if[[ "string" =~ ^[a-z]+$ ]];
2 # if statement code here...
```

Рисунок 1.3 - Додаткові можливості подвійного оператора квадратних дужок

Третім каменем спотикання у мові програмування BASH є складність синтаксису деяких доволі простих конструкцій як оператор умовного переходу «switch», де в BASH він зазначен як «case». Подвійна точка з комою, не зрозумілі означення кожних із опцій оператора, та суцільний безлад у символах синтаксису призводить до тяжкого для читання коду. Приклад цієї конструкції можна знайти на Рисунку 1.4, нижче.

```
1 case $condition_variable in
2     case_value_1)
3     #case code here...
4     ;;
5     case_value_2)
6     #case code here...
7     ;;
8     *)
9     #case code here...
10    ;;
11 esac
```

Рисунок 1.4 - Оператор умовного переходу «case» (або більш відомий як «switch»)

Четвертою, та найбільш неоднозначною конструкцією є цикл «for», або більш відомий у інших мовах програмування як «for each» (бо перебирає усі елементи колекції без індексування).

Як можна побачити на Рисунку 1.5, цикл «for» може виглядати по різному у місці де йому передається массив для ітерування. Проблема у тому, що при відсутності подвійних лапок, ітерація у текстових колекціях буде відбуватись не по елементам колекції, а по елементам розділеним по пропускам у тексті кожного із елементів.

```
1 integer_array+=(1)
2 integer_array+=(2)
3 string_array+=("first string")
4 string_array+=("second string")
5
6 for integer_entry in ${integer_array[@]};
7 #for statement code here...
8
9 for string_entry in "${string_array[@]}";
10 #for statement code here...
```

Рисунок 1.5 - Різні форми циклу «for» («for each»), через тип массивів

П’ятою неоднозначністю можна назвати відсутність означення де котрий параметр у визначенні функції. Як видно із коду на Рисунку 1.6 нижче, параметри можна отримати через позиціонування, чи індекс переданного параметра, але це не однозначно а ні для того, хто визначає функцію, а ні для того, хто її викликає.

```
1 function foo
2 {
3     local parameter_1="$1"
4     local parameter_2="$1"
5 }
```

Рисунок 1.6 - Приклад визначення функції з параметрами

Фінальним, можно зазначити те, що різні конструкції використовують різні оператори щоб визначити їх межі функціональних блоків, а саме:

- функції використовують фігурні дужки «{» та «}»;
- цикли використовують ключові слова «do» та «done»;
- оператори умовного переходу використовують ключові слова «then» та «fi», інколи «esac».

Такий синтаксис нагадую суміш синтаксису із мови програмування BASIC, та C, і немає ніякої узгодженості.

Неоднозначність синтаксису чи його складність одних і тих самих конструкцій, таких як умовні оператори переходу; неочевидні рішення, такі як розширення функціоналу подвійними дужками в умовних операторах; різна поведінка пов'язана із неоднозначними синтаксичними змінами як у циклах; відсутність єдності у синтаксису операторів - перетворює потужний інструмент BASH на не комфортну мову програмування що має безліч місць для помилок. Більш того, ці помилки складно виявити, навить при виконанні програми, бо більшість із них не виводить ніяких попереджувальних повідомлень.

1.3 Вирішення проблеми та пошук аналогів

Рішенням проблеми синтаксису, та неоднозначності BASH, в умовах сьогодення неможливо обмеженням старого та впровадження нового стандарту, як зазначено у попередньому пункті. Тож можливим рішенням може бути тільки декілька варіантів.

Перший та найпростіший це розробити інтерпретатор мови програмування BASH, що не буде мати неоднозначностей та буде виводити на екран всі попередження якщо користувач намагається використати конструкцію, що може призвести до неочікуваного на перший погляд результату. Але в цьому разі такий інтерпретатор не буде дотримуватись стандарту BASH, крім того скрипти написані із особливостями не прийнятними у межах даного проекту будуть мати зміни у поведінці, що призводить цінність такого рішення до нулю, тим паче це не вирішить проблему із синтаксисом. Цей підхід має багато аналогів, що намагаються вирішити цю проблему, такі як інтерпретатори: sh, ash, zsh, ksh, ch, rc, та інші. Але вони тільки підтримують тенденцію bashism-a.

Другий варіант, це не змінюючи компілятор, створити програму-валідатор, що буде перевіряти на корректність написаний код, та видавати можливі підказки

щодо його покращення. Таке рішення дозволить перевірити старі скрипти, та писати нові у кращому вигляді. Та в цьому рішенні також немає можливості виправити синтаксис мови програмування BASH. Це вже реалізовано, наприклад у такому проекті як ShellCheck [4], але він не охоплює усіх можливих помилок, бо занадто велика варіація різних інтерпретаторів, із різними додатковими можливостями. Але стандарти BASH покриваються майже повністю.

Третім варіантом є написання компілятора для вже існуючої, чи для нової мови програмування, що буде транслюватися (компілюватися) у BASH код. Що дозволить вирішити проблемі із синтаксисом, додавати нові можливості на базі стандарту BASH, та зберегти сумісність із старими скріптами, бо інтерпретатор не модифікується. Даному рішенню поставленої проблеми аналогів немає.

Порівняння всіх варіантів наведено в таблиці 1.1, нижче. Найкращими є останні два, що стосуються написання компілятора, але найгнучкішим є той, що передбачає створення окремої мови програмування, бо надає можливість:

- створити її найбільш гнучкою відносно потреб;
- дозволить консолідувати знання із однієї галузі в одному місці.

Грунтуючись на цьому порівнянні, ця робота направлена на розробку нової мови програмування, та компілятора до неї, що вирішить зазначені проблеми (зазначені вище), та привнесе додаткові можливості використання мови програмування BASH.

Таблиця 1.1 - Порівняння різних рішень проблем мови програмування BASH

Опис виникшої проблеми	Написання нового інтерпретатору для мови програмування BASH	Написання програми-валідатора для мови програмування BASH	Написання компілятора існуючої мови програмування в BASH	Написання компілятора нової мови програмування в BASH
Сумісність старих скриптів	Ні	Так	Так	Так
Вирішення проблеми синтаксису	Ні	Ні	Так (залежить від вибору мови)	Так

Продовження таблиці 1.1

Вирішення проблеми неоднозначності конструкцій	Так	Так	Так	Так
Можливість розширити функціонал	Так	Ні	Ні	Так
Опис виникшої проблеми	Написання нового інтерпретатору для мови програмування BASH	Написання програми-валідатора для мови програмування BASH	Написання компілятора існуючої мови програмування в BASH	Написання компілятора нової мови програмування в BASH
Необхідність вивчати додаткові знання	Ні	Ні	Ні (за умови, якщо компільована мова програмування вже відома)	Так
Наявність аналогів	Так	Так	Ні	Ні

1.4 Методологія розробки нової мови програмування

Нова мова програмування, тим паче для скриптів, що має бути простою, та швидкою у засвоєнні, повинна базуватись на популярних знаннях, доступні широкому колу спеціалістів, щоб не викликати труднощів у рішенні задач. Також вона повинна бути доволі простою, не складнішою, а краще простішою за BASH. У той самий час, вона повинна мати більш комплексні конструкції, щоб дати можливість простіше, і швидше вирішувати задачі що аналогічно написати у BASH займало б багато часу. Найкраще рішення для цього може бути тільки використання вже широко відомих, та перевірених часом рішень із інших мов програмування.

По перше для обрання орієнтиру, необхідно розуміти які саме ключові особливості, нова мова програмування привнесе, яких немає у BASH, але є у більшості мов сучасних програмування, такими є:

- согласований та однотипний синтаксис;
- функції із визначеними параметрами;
- модифікатори типів даних;
- структури даних.

Окрім цього, так як ця мова програмування буде трансльована у BASH, який не вищначає помилок щодо сумісності типів, типізація у цієї мови програмування буде строгою.

Провівши аналіз найпопулярніших мов програмування у 2020, отримаємо наступні діаграми за популярністю скриптових (див. Рисунок 1.7) та компільованих (див. Рисунок 1.8) мов програмування загальної направленості (діаграми ґрунтуються на даних інтернет порталу stackoverflow [5]).

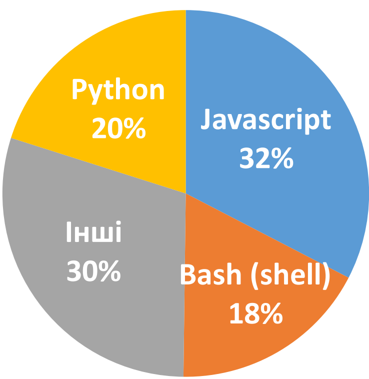

Рисунок 1.7 - Співвідношення скриптових мов програмування у 2020 році

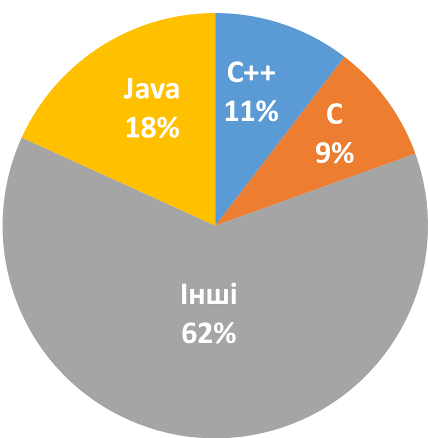

Рисунок 1.8 - Співвідношення компільованих мов програмування у 2020 році

Грунтуючись на отриманих даних, можна стверджувати, що найпопулярнішими компільванними мовами програмування є Java, C++, C, що мають так званий C-like (від англ. «сі-подібний», мається на увазі мова програмування) синтаксис. Так само і найпопулярніша скриптова мова програмування Javascript має C-like, крім Python і BASH.

Оптимальним рішенням буде розробити мову програмування із C-like синтаксисом, можливо із елементами позиченими із синтаксиса мови програмування Python.

2 СПЕЦИФІКАЦІЯ ВИМОГ ДО МОВИ ПРОГРАМУВАННЯ SBASH

2.1 Глосарій

Таблиця 2.1 - Глосарій

Термін	Опис терміну
Операційні системи	
Дистрибутив	Форма розповсюдження програмного забезпечення. Збірка певного переліку програм та ядра операційної системи.
GNU	Проект, що розробляє безкоштовні аналоги програм оточення операційної системи UNIX.
GNU/Linux	Назва дистрибутиву Linux, у котрий включено програми із проекту GNU.
Linux	UNIX-подібна операційна система та її однойменне ядро.
UNIX	Однойменна операційна система, а також сім'я багатозадачних операційних систем, розроблених компанією AT&T у 1970-х роках.
Програми та технології	
BASH	Оболонка терміналу для UNIX-подібних операційних систем, розроблена у межах проекту GNU.
Devtools	Допоміжні програми, що використовуються від час розробки програмного забезпечення.
Embedded система	Система чи що працює на дуже обмеженому обладнанні з точки зору обчислювальної потужності.
Regex	Технологія, що дозволяє аналізувати великі масиви даних (тексту), завдяки формування так званих шаблонів пошуку із послідовності службових символів. Найпоширенішим застосуванням є пошук заздалегідь відомих лексичних конструкцій. Має декілька схожих між собою – стандартів.
SBASH	Розроблена у межах цього проекту – мова програмування, та компілятор до неї. Є процедурною, компільованою мовою програмування, та належить до класу із строгою типізацією.
SDK	Комплекс програмного забезпечення, бібліотек, програм розробки, тощо, зібраних у одному пакеті призначених для спрощення розробки специфічних програмних продуктів.
Мови програмування	
Інстанс змінної	Створення змінної, яка має значення та іменування.
Колізія імен	Неоднозначність яка постає під час семантичного аналізу, за умов, що в одній і тій самій області імен з'являється дві різні змінні із одним і тим самим ім'ям. Це унеможливлює успіх семантичного аналізу.

Продовження таблиці 2.1

Термін	Опис терміну
Композитний тип даних	Типи даних що характеризується можливістю створити новий тип даних агрегацією інших в унікальному наборі.
Bashism	Комплексна проблема, що виникає під час написання скриптів на мові програмування BASH, що характеризується набором різних службових програм у різних дистрибутивах UNIX-подібних операційних систем. А як наслідок – виключає можливість писати скрипти для різних дистрибутивів у єдиному стилі.
Built-in types	Так звані – примітиви у мові програмування. Це типи даних що вбудовані у мові програмування, тобто підтримуються нею без сторонніх залежностей.
C-like (синтаксис)	Синтаксис, що походить на синтаксис мови програмування C.
Python-like (синтаксис)	Синтаксис, що походить на синтаксис мови програмування Python.
Компілятор	
Валідатор	Програма чи частка програми, що проводить валідацію певних даних щодо певних вимог.
Валідація	Процес, результатом якого є логічне значення, що свідчить про рівень збіжності вхідних даних щодо певних правил.
Компілятор	Програма, що проводить валідацію певних вхідних файлів (написаних певною мовою програмування відповідною, для котрої призначено компілятор). Здійснює їх лексичний, граматичний та семантичний аналіз, успішним результатом котрого є вихідний файл, зберігаючий ідентичну програму, але написану у відмінній від оригінальної мові програмування.
Компіляція (семантичний аналіз)	Процес, що виконує компілятор, чи його частина – так званий: семантичний аналіз, тобто валідація AST щодо відповідності певним семантичним правилам. Іншими словами, це перевірка компілятором – відповідності AST до логічних щодо контексту, та можливостей мови програмування – конструкцій.
Лексинг (лексичний аналіз)	Процес аналізу (валідації) вхідних даних щодо певних лексичних даних, результатом котрого є послідовність так званих «токенів», що є частинами вхідних даних. Іншими словами, це розбиття вхідного файлу на зрозумілі, з точки зору пресингу, - «слова».
Парсинг (граматичний аналіз)	Процес аналізу (валідації) послідовності токенів щодо певних граматичних правил, результатом котрого є побудоване AST. Іншими словами, це перевірка компілятором – вхідних файлів на зрозумілі йому конструкції.
AST	Структура даних, що використовує компілятор, для зберігання програми у зрозумілому для нього форматі для семантичного

Продовження таблиці 2.1

Термін	Опис терміну
	аналізу. Вона є деревом, що складається із токенів отриманих під час лексичного аналізу та побудованих під час граматичного аналізу.
APS	Структура даних, розроблена у межах даного проекту, що зберігає програму написану мовою програмування SBASH у вигляді колекції стандартних команд мови програмування BASH.

2.2 Мова програмування SBASH

Як зазначено у назві цього розділу, мова програмування, шо буде розроблятися у межах цього проекту буде називатися SBASH, така назва була обрана через додавання латинської літери «S», до назви мови програмування, проблематика якої була розглянута у попередньому розділу, а саме BASH. Літера «S» означає скорочення від англійського слова strict, що означає «суворий» чи «вимогливий», та відображає мету (у межах вдосконалення BASH) якої домагається нова мова програмування, а саме:

- привнести строгу типізацію заради зниженню кількості помилок у скриптах на мові програмування BASH;
- надати можливість скомпілювати програму написану на мові програмування SBASH у вихідний код BASH з дотриманням універсального формату;
- знизити загальну можливість допущення помилки при написанні програми, що у результаті компілювання стане BASH скриптом.

З іншої точки зору, в мові програмування SBASH втілюється ідеологія збереження короткого та простого, та інклюзивного для пересічного програмісту - синтаксису. Це зроблено заради зниження так званого «порогу входження» у нову галузь для спеціаліста незнайомого із даною мовою програмування. Найголовніше, що дає така ідеологія - це збереження часу розробника, а знайомі прийоми із інших мов програмування, просто спрощують процес її вивчення.

Таким чином підсумовуючи зазначені напрямки розробки мови програмування SBASH, можна визначити її наступні ознаки:

- строго-типізована система типів;
- процедурна семантика;
- компільований тип використання;
- C-like синтаксис;

– сформований під впливом таких мов програмування як C, C++, Python.

Попри все перераховане вище, треба розуміти, що незважаючи на майбутню розробку нових можливостей у цій мові програмування є неможливі на теперішній час удосконалення.

Як було зазначено у попередньому розділу: загальний вигляд використання мови програмування SBASH, буде мати вигляд написання коду у текстових файлах, наприклад із розширенням файлу «*.sb», компілювання їх у однойменному компіляторі SBASH, а результатом буде скриптовий файл на мові програмування BASH.

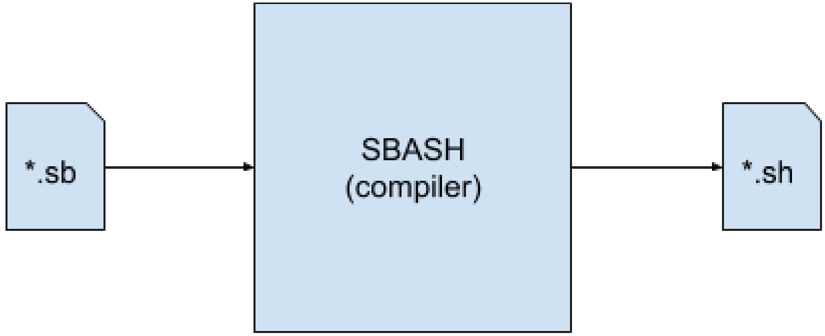

Рисунок 2.1 - Сценарій використання компілятора мови програмування SBASH

Як вказано на Рисунку 2.1, файл із розширенням «*.sh» - це результат компіляції вхідного файлу написаного на мові програмування SBASH.

Якщо зобразити на діаграмі як працює програми написані мовою програмування SBASH, то діаграма буде мати наступний вигляд (див. Рисунок 2.2 нижче).

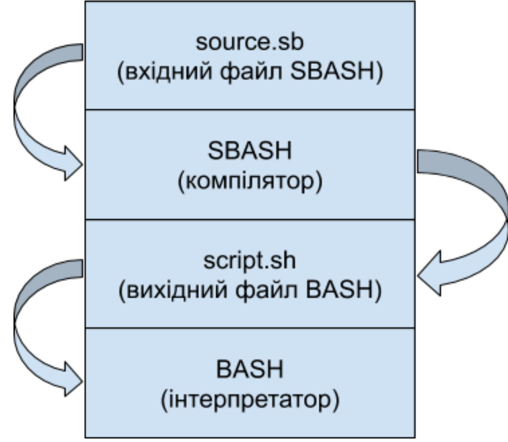

Рисунок 2.2 – Етапи виконання програми написаною мовою SBASH

Слід пам’ятати, що компіляцію треба виконувати тільки після модифікації вхідного файлу програми, а результат компіляції можна виконувати необмежену кількість разів на інтерпретаторі BASH.

Як було зазначено у даному підпункті, деякі вдосконалення мови програмування можуть бути неможливі, чи просто надмірні у межах транслювання коду у BASH. Основною причиною такого обмеження є сам інтерпретатор BASH, та його архітектура. Це виражається у тому, що BASH на відміну від, наприклад, мови програмування Python, виконує різні задачі завдяки різним програмам. Тобто виконання команд фізично виконується різними виконуючими файлами, бо сам інтерпретатор баш доволі обмежений у своїх можливостях. Прикладом такого обмеження для мови програмування SBASH є перехід до об’єктно орієнтованого классу мов програмування. Причина цьому у тому, що BASH не є повноцінною мовою програмування у звичному для пересічного програміста сенсі, а це саме «клей», що надає можливість комфортно використовувати дистрибутиви UNIX подібних систем без графічного інтерфейсу, тобто через консоль.

2.3 Основні відмінності мови програмування SBASH та її особливості

Згідно із попереднім пунктом розглянемо головні відмінності мови програмування BASH від SBASH на прикладах особливостей та можливостей останньої.

2.3.1 Типи даних у мові програмування SBASH

Як відомо, у мові програмування BASH типізація є динамічною, тобто не має наперед зазначених типів даних із визначеною поведінкою. У мові програмування SBASH типізація є статичною із заздалегідь визначеними типами, їх ще називають built-in [6] types (від. англ. «вбудовані типи», так звані «примітиви» [7]). Провівши порівняння різних мов програмування (зазначених у першому параграфі як найпопулярніші) із статичною типізацією, маємо наступну порівняльну таблицю 2.2. Слід зазначити, що тип void (від англ. «порожнеча») не вказано у порівнянні, він не надає інтересу у межах даної задачі, бо не може зберігати жодних даних (за звичайних умов). Також не враховуються складені типі даних такі як колекції чи посилання, вони будуть розглянуті нижче.

Таблиця 2.2 - Порівняння наявності різних типів даних у найпопулярніших мовах програмування із статичною типізацією

Тип даних	Призначення	Мова програмування		
		Java	C++	C
integer	Зберігає цілі числа	Так	Так	Так
short	Зберігає цілі числа (відрізняється від int меншою місткістю)	Так	Так	Так
long	Зберігає цілі числа (відрізняється від int більшою місткістю)	Так	Так	Так
boolean	Зберігає логічне значення («істина» чи «не істина»)	Так	Так	Так
double	Зберігає числа із десятковою частиною (має спеціальний формат)	Так	Так	Так
float	Зберігає числа із десятковою частиною (має спеціальний формат, відрізняється від double меншою місткістю)	Так	Так	Так
char	Зберігає один символ	Так	Так	Так
wide char	Зберігає один символ (відрізняється від char більшою місткістю)	Ні	Так	Ні
byte	Зберігає один байт	Так	Ні	Ні

В мові програмування BASH не має вбудованої підтримки стандартів double чи float, як і інших типів даних, межі обчислювальних можливостей залежать від імплементації інтерпретатора та додаткових службових програм, таких як bc (від англ. «best calculator» - найкращий калькулятор). Через це можна нівелювати цим показником і зосередитись виключно на їх поведінці зумовленой типами даних у них зберігаємих. Таким чином можна відмовитись від обробки диапазону підтримуючих значень, бо це буде обробляти інтерпретатор BASH чи інші службові програми.

Таким чином, можна обмежити необхідні для підтримки типи даних для мови програмування SBASH наступними:

- integer (втілює integer, short, та long);
- double (втілює float та double, назва може бути будь якою);
- boolean;
- string (пояснення про цей тип даних буде наведено нижче).

Відмова від типу даних byte зумовлена іншою направленностью мови програмування SBASH, бо вона не передбачає використання пам’яті комп’ютера

на низькому рівні. А тип даних string є аналогом комбінації char чи wide char у єдиний тип, а також із додавання ознаки коллекції до цього типу; це зумовлено вбудованою підтримкою інтерпретатора BASH цього типу даних, та відсутність потреби виражати його через колекцію char для користувача.

2.3.2 Модифікатори типів даних мови програмування SBASH

Згідно із попереднім пунктом, можна порівняти мови найпопулярніші мови програмування із статичною типізацією, але на цей час зробивши виборку по модифікаторам типів данних. Це додаткові конструкції які розширюють сенс зазначених вище built-in types, приклад та наявність котрих можна побачити на таблиці 2.3, нижче.

Таблиця 2.3 - Порівняння наявності різних модифікаторів типів даних у найпопулярніших мовах програмування із статичною типізацією

Модифікатор	Призначення	Мова програмування		
		Java	C++	C
* (вказівник)	Вказує на ділянку пам'яті із даними типу вказівника	Ні	Так	Так
& (посилання)	Зазвичай псевдонім змінної того ж типу	Так	Так	Так
[] (колекція)	Створює серію змінних вказаного типу	Так	Так	Так
const	Не дає можливості модифікувати значення змінної після її ініціалізації	Так	Так	Так

Необхідність маніпулювати із пам’яттю комп’ютера у мові програмування BASH а внаслідок SBASH немає, тож вказівника у мові програмування не буде, але мати можливість взяти посилання чи створити колекцію, як і можливість створювати константні змінні. Слід зазначити, що у порівнянні відсутні модифікатори типів, що змінюють так чи інакше місткість built-in types, причина цього вказана у попередньому пункті.

Мова програмування SBASH позиціонується як легка у освоєнні, тож використання в умовних позначеннях модифікаторів типів - символів не є легким для читання, тим паче складно для запам’ятання у випадку, якщо із цим не

працювати. Тож для вибраних модифікаторів типів данних були обрані скорочені вирази які використовуються у такій мові програмування як C# (наступною серед зазначених із найпопулярніших компільованих мов програмування):

- ref (вказує змінну як псевдонім до іншої із тим самим типом даних);
- const (не дозволяє змінити значення змінної після ініціалізації);
- [] (створює колекцію змінних із тим самим типом даних).

Як можна побачити, модификатор типу, що створює колекцію змінних - залишається символьним, бо використовується у більшості найпопулярніших мов програмування у такій формі.

2.3.3 Структури даних у мові програмування SBASH

Однією із провідних особливостей мови програмування SBASH є структури даних, яких немає у мові програмування BASH. Їх відсутність унеможливлює організувати зрозуміло код, та ускладнює його читання (розуміння) через надмірну заплутаність. Саме із вирішенням цієї проблематики розроблені дві структури даних у мові програмування SBASH доступні для модифікації.

Якщо основні типи даних мають назву built-in, через те що вони присутні за умовчанням у мові програмування, перелічені у цьому пункту структури даних називаються доступними для модифікації через те, що це «шаблони» структур даних, котрі визначає користувач.

2.3.4 Структури

Структури у мові програмування SBASH, це максимально спрощена презентація організованих даних. Найближчий аналог, що може бути підібраний, це структури даних із мови програмування C чи C++, де ними виступають консолідація певного набору змінних під спільною назвою. Зазвичай структури задаються ключовим словом «struct» (скорочення від англ. «structure» - структура), через незмінність даного скорочення майже у всіх мовах програмування де є така конструкція - вона буде використана і у мові програмування SBASH.

Іншими словами, структура даних в SBASH, це спеціальна синтаксична конструкція, що у якій декларуються змінні які називають атрибутами даної структури даних. Сама по собі - структура даних не є екземпляром перелічених змінних, а є лише прототипом, чи користувальницький типом даних. Екземпляр структури можна створити, а безпосередньо через нього мати доступ до усіх перелічених у неї атрибутах.

Як було зазначено у попередньому розділу, мова програмування SBASH не може набути ознак класу об'єктно орієнтованих мов програмування через певні обмеження проектування та реалізації самої мови програмування, так і через відсутність необхідності вирішувати таку задачу у межах даного проекту. Але деякі принципи можливо позичити у об'єктно орієнтованих мов програмування, наприклад такі як «наслідування». Цей термін, взятий із такої галузі знань інформаційних технологій як «теорія типів», він свідчить про те, що типи, чи у даному випадку - об'єкти, можуть набувати особливостей і ознак інших типів, тобто спадкувати їх. У випадку із структурами даних мови програмування SBASH, можна визначити тільки один шлях роботи цього принципу, а саме: спадкування однією структурною - списку атрибутів її батьківської структури чи структур, що робить її композитним типом даних.

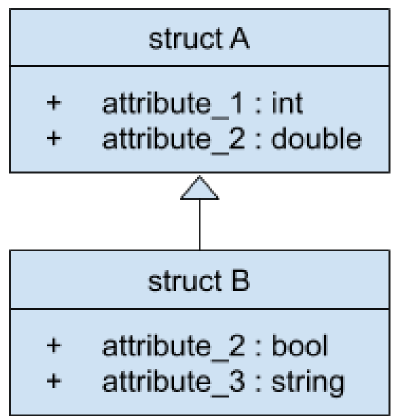

Рисунок 2.3 - Приклад спадкування структур даних у мові програмування SBASH

Приклад спадкування структур можно подивитися на Рисунку 2.3, віще. Слід розуміти, що на спадкування накладається деякі обмеження. Поперше, це неможливість називати одними і тими самими ім'ями атрибути у ланцузі наслідування структур даних, бо це призведе до неможливості визначення, до котрого атрибуту бажає звернутися користувач при необхідності (а мета мови програмування SBASH зменшити кількість неоднозначностей у даній галузі). Подруге може постати так звана проблема «ромбовидного успадкування», яка полягає у площині дублікації імен структури при подвійному спадкуванні такої.

Як видно із схеми спадкування на Рисунку 2.4, нижче - фактично структура «D» спадкує структуру «A» двічі, тобто можлива так звана «колізія імен». Різні мови програмування вирішують цю проблему по різному, наприклад C++ має такий інструмент як «віртуальне наслідування», що допомагає впоратися із даною

проблемою. Але, оскільки мова програмування SBASH орієнтована на простоту у використанні, це потрібно вирішувати автоматично у компіляторі, тим паче, що це не є складною задачею.

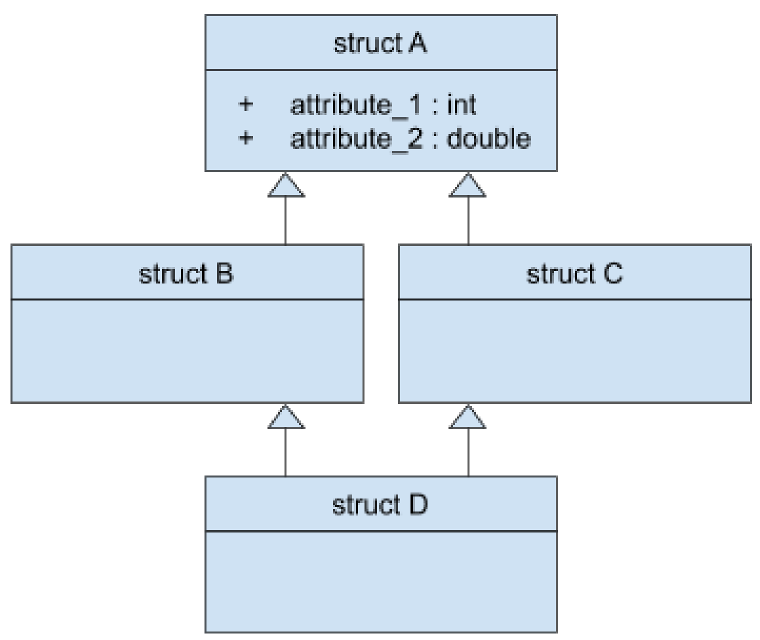

Рисунок 2.4 - Приклад ромбовидного успадкування

Все що потрібно зробити компілятору, це оцінити, чи є двічі унаслідувана структура «A» - спільним предком для наслідуваних структур «B» та «C». Проте, потрійне наслідування структури «A» структурою «D» безпосередньо і за умови що вона вже є у ланцюжку наслідування - є семантичною помилкою.

2.3.5 Перерахування

Перерахування це одна із найпростіших структур даних, доступних для створення користувачем. Основною її особливістю є вказання вичерпного, відмінного від нуля (за довжиною) списку значень (літералів). Тобто іншими словами, це у результаті буде така сама змінная наприклад і built-in types, але яка може набувати тільки вказаних у її декларації значень (визначених користувачем літералів).

Такі перерахування зазвичай прив’язують до деяких значень, наприклад як у мовах програмування як C та C++, можливо завдати завчасно та отримати - унікальне ціле число притаманне конкретному значенню літерала цього перерахування. У той самий час у мовах програмування Java та C#, можна завдати заздалегідь та отримати - унікальне текстове значення для літералів перерахування.

Мова програмування SBASH може дати більше та об'єднати всі built-in types літерали доступни для завдання заздалегідь значень літералам перерахування, але із можливістю дублювати значення та звертатись до них за ім'ями їх типів. Але як запобіжник для доброго стилю коду написаного на мові програмування SBASH, та збереження узгодженості у кожному типі, компілятор повинен перевіряти, щоб у межах кожного із перерахувань - все літерали мали однакові набори значень за типом. Таким чином проста структура даних - перерахування, набуває нових можливостей, що допоможуть як зберегти кількість написаних рядків коду, так і додадуть новий досвід використання відомої структури даних.

2.4 Функції у мові програмування SBASH

Функції у мові програмування SBASH мають таку ж саму функціональність як і у референцних аналогів, тобто мовах програмування: Java, C, C++, Python, тощо. Це означає що вони повинні мати заздалегідь задекларовані параметри, із вказаними типами, а також можливість повернути деяке значенні із функції. Більш того, функції у мові програмування SBASH, отримають можливість задавати список параметрів, що вони можуть отримати, через те, що строга типізація сама на це натякає (із нею виникає необхідність валідувати передані параметри).

Так як мова програмування SBASH розробляється у межах удосконалення написання скриптів на мові програмування BASH, то однією із очевидних рис мови програмування може стати безпосереднє написання коду BASH у вхідних файлах написаних на мові програмування SBASH.

Грунтуючись на вище перерахованим, можна стверджувати, що вірогідним рішенням заданих задач є створення двох типів функцій: функцій що валідуютьсяс компілятором SBASH, та написані на одноіменній мові програмування; функції, що мають таку саму декларацію як і звичайні функції написані на мові програмування SBASH, але містять написаний користувачем BASH код, котрий буде виконаний без валідації. Аналогом останньої є вставки асмеблерного коду (ASM) у мові програмування C++. Така функціональність може надати багато волі користувачу, навіть без стандартної бібліотеки функцій, адже дає йому можливість використовувати вже відомі йому команди та програми із оболонки терміналу BASH, та зв'язувати їх із кодом написаним мовою програмування SBASH, тим самим розширюючи її потенціал. Слід пам'ятати, що мова програмування BASH не підтримує різних типів даних, а тому параметри таких функцій повинні мати тільки string типа даних. Виконання самої функції повинно повертати так званий exit status

(від англ. «статус виходу» чи «статус завершення»), тобто ціле число, що зберігає інформацію про статус завершення виконання програми, у даному випадку - функції. Ці особливості функцій написанних із вставкою BASH коду повинні валідуватися компілятором.

Якщо при обранні синтаксису спиратися на найпопулярніші мови програмування, то на теперішній час він поділяється на C-like, та Python-like (за аналогією із C-like, Python-like означає: синтаксис подібний Python). Оскільки всі конструкції у мові програмування SBASH повинні бути уніфіковані, то це унеможливлює використання синтаксису декларації функцій такої як в мові програмування C (див. Рисунок 2.5), розглянемо як вона задається:

```
1 void function_name()
2 {
3     //function code here...
4 }
```

Рисунок 2.5 - Приклад декларації функції мовою програмування C

Питання у тому, що мова програмування SBASH немає вбудованого типу даних «void», внаслідок цього використання такої декларації функцій - неможливо. Альтернативою цьому є використання декларації функції, схоже з мовою програмування Python (див. Рисунок 2.6), але за умови того, що тип повернене значення буде зазначений після декларації параметрів функції - опціонально, згідно до необхідності строгої типізації у мові програмування SBASH.

```
1 def function_name()
2 {
3     #function code here...
4 }
```

Рисунок 2.6 - Приклад декларації функції мовою програмування Python

2.5 Головна специфікація компілятора мови програмування SBASH

Сам по собі, компілятор [8] - це дуже проста програма, яка все що може робити, це приймати на вхід файли, валідувати їх згідно із відомими їй правилами, та на вихід видавати файл написаний у іншій мові програмування. Через відсутність велико кількості функцій, немає необхідності розробляти графічний інтерфейс, біль того, він буде зайвим, якщо згадати, що програма розробляється для UNIX подібних систем, в яких основним маніпулятором є термінал, тим паче у embedded розробці.

Слід пам’ятати, що розробка компілятора починається із самого початку та не має вже готової кодової бази. Як наслідок цього, неможливо розробити за доволі короткий термін часу повноцінний компілятор: який не буде мати жодних помилок, чи буде мати підтримку всього стандарту мови програмування, як наслідок цього, розробка компілятора буде вестись до так званого POC етапу (від англ. «Proof Of Concept» - «доказ концепту»), з можливою підтримки у майбутньому, вже поза межами даної роботи, наприклад, під час здобування наступного рівня освіти.

2.5.1 Варіанти використання компілятору SBASH

Як зазначено вище, основним варіантом використання компілятору є компіляція коду, і тільки, додатковими функціями програми компілятора можуть бути задання параметрів компіляції (так званих «флагів компілятора»), виведення версії компілятора, тощо (див. Рисунок 2.7).

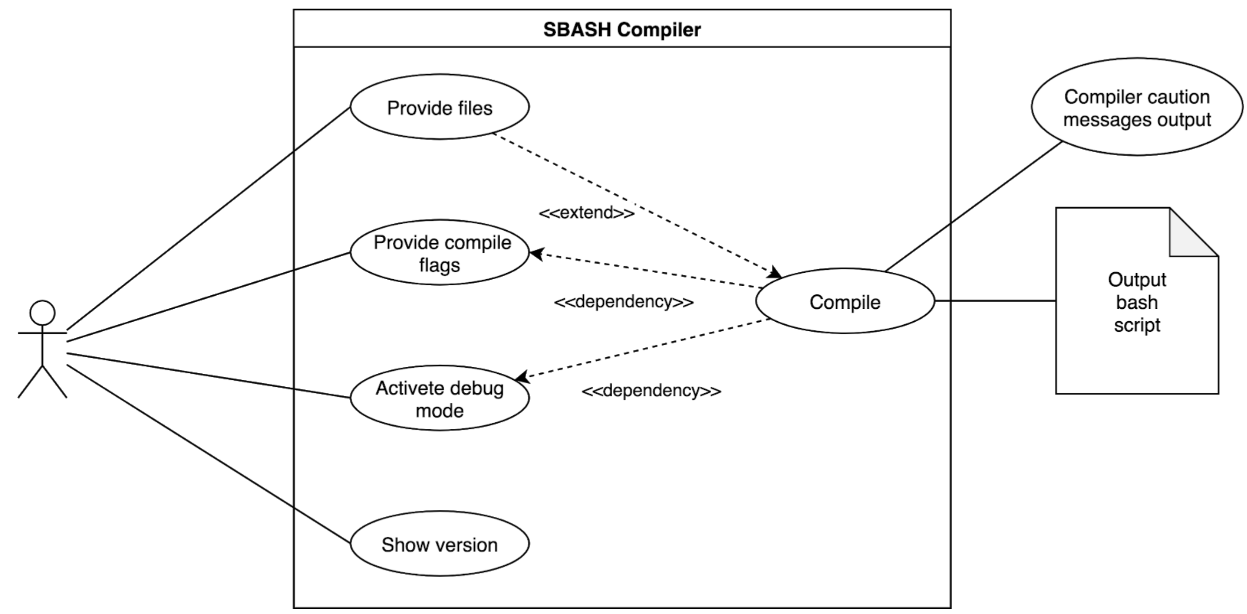

Рисунок 2.7 - Головна діаграма Use-Case

2.5.2 Базова структура компілятора SBASH

Можна зазначити що у процесі компіляції програма може використовувати різні функціональні модулі, відповідають за такі етапи компіляції як:

- лексичний аналіз;
- граматичний аналіз;
- побудова абстрактного синтаксичного дерева;
- семантичний аналіз;
- побудова абстрактної структури програми;
- генерація коду.

Більшість із них, можна групувати, та залишити основні групи такі як: відповідальні за парсинг, компіляцію, та генерацію. Детально їх можна побачити на Рисунку 2.8, нижче.

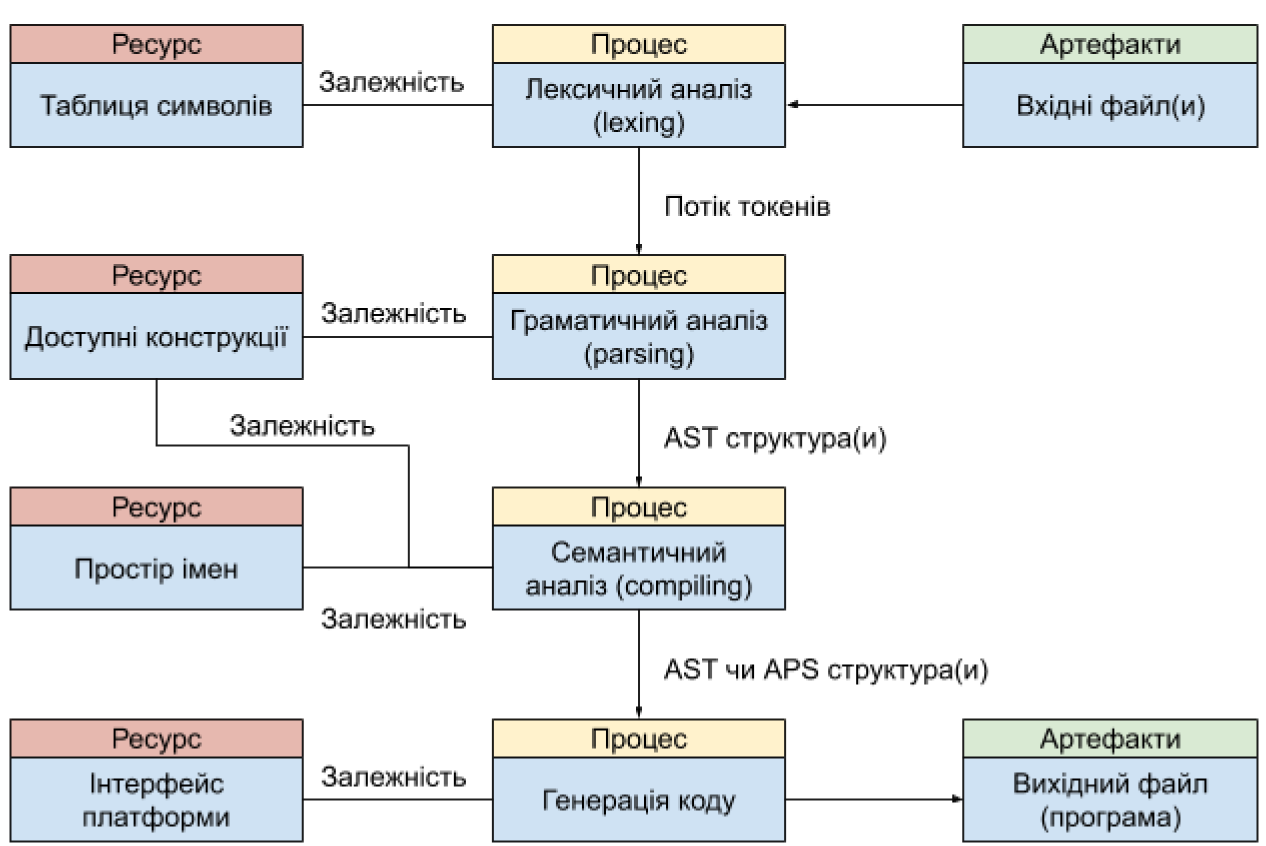

Рисунок 2.8 - Базова структура роботи компілятора

Кожну з них можна розробити як окремий модуль (наприклад динамічну бібліотеку). Що надасть можливість оновлювати та вдосконалювати поодинці кожен не впливаючі на інші. Це знизить пов'язаність архітектури та підвищить зачепленя що вважається ознакою добре спроектованих архітектури.

2.6 Технології до застосування у реалізації компілятора мови програмування SBASH

2.6.1 Вибір мови програмування для написання компілятора

Вибір мови програмування для написання компілятора - доволі очевидний, якщо подивитися на вибір мови у провідних проектах цієї галузі. Звісно, реалізувати компілятор можна будь якою мовою програмування, але очікуваними від компілятора вимогами буде швидкість роботи, що може забезпечити виключно компільовані мови програмування, а також достатній рівень гнучкості обраної мови програмування та обсягом бібліотек можливих для використання нею.

Більшість найпопулярніших інтерпретаторів мов програмування реалізовано на мовах програмування C чи C++, а через більшу гнучкість та обсяг різних бібліотек, а також за швидкість розробки буде обрана остання.

На даний момент існує багато стандартів мови програмування C++, зазвичай, кожний із котрих включає у себе попередній та додаткові можливості розроблені у межах нового стандарту, тобто зберігають зворотню сумісність. Тож є сенс у обранні останнього стандарту який повністю підтримується всіма його провідними компіляторами по всіх доступних в ньому можливостях, таким є C++17. Таким компілятором є Clang версії 9.

2.6.2 Вибір бібліотек для розробки компілятора

Очевидним вибором до C++, є комплекс бібліотек під назвою BOOST, головна перевага котрої полягає у сумісності із стандартною бібліотекою STL та STD для C++, та передбачає не тільки розширення можливостей останньої, а також нові алгоритми, та функції, що будуть використані в проекті та прискорять його розробку [9].

Такоє слід зазначити, що однією із библиотек BOOST є бібліотека BOOST TEST, що надає функціонал для тестування коду, таким чином, є майже всі необхідні компоненти для розробки компілятора без використання сторонніх технологій.

2.6.3 Вибір додаткових технологій для розробки компілятора

Попри те, що комплекс бібліотек BOOST має у собі бібліотеку BOOST SPIRIT, що представляє функціонал парсеру (синтаксичного аналізатору), її використання у межах даної роботи надмірне та не виправдане. Основним мінусом

даної бібліотеки є дуже комплексний опис синтаксису аналізатора, більш того, який використовує C++, а ні спеціально розроблений для цього синтаксис.

Щоб полегшити розробку та поскорити її темп було обрано на роль синтаксичного та лексичного аналізаторів програми BISON та FLEX відповідно (це аналоги програм із UNIX, що відомі як YACC та LEX [10]). Але, через те, що в оригіналі вони розроблені для мови програмування C, були знайдені їх аналоги написання для мови програмування C++, а саме BISON C++ та FLEX C++ відповідно. Ці програми генерують код C++ на основі спеціально написанних для них файлів граматики лексичного та синтаксичного аналізаторів, що прискрює розробку: зв’язавши вперше згенеровані файли та проект на C++, немає необхідності редагувати код написанний на C++, а лише просто та швидко редагувати файли граматики.

2.6.4 Вибір системи збірки проекту

Виходячи із необхідності швидко розробляти проект та використовувати найбільш широко відомі технології, очікуваним рішенням буде обрання системи збірки CMAKE. Вагомим аргументом у її користь є можливість прив'язати до неї тестування, збір звіту по ньому, а найголовніше - можливість простої організації збірки у «облаці», тобто на серверу.

3 ПРОЕКТУВАННЯ МОВИ ПРОГРАМУВАННЯ SBASH

3.1 Проектування базових конструкцій

В даному пункті описуються базові конструкції мови програмування SBASH. Такими конструкціями називаються вирази із котрих складається основна семантика мови програмування SBASH. Іншими словами, це найменші блоки коду із котрих складається програма написана цією мовою програмування. Тобто цей розділ є повною документацією мови програмування, що розробляється у межах даного проекту. Треба зауважити, що у цьому розділі надається інформація щодо розробленого:

- синтаксису (безпосередньо – правил лексичного та граматичного аналізу);
- компіляції (семантичного аналізу).

Слід зазначити, що при використання одних і тих самих конструкцій за назвою у різних конструкціях – на них поширюються одні і ті самі правила компіляції, але не вказуються на пряму, заради зберігання лаконічності документування. Виключенням є тільки випадки очевидно зазначені у тих чи інших конструкціях.

3.1.1 Змінні та колекції

Змінні це найпростіші конструкції мови програмування SBASH. Вони дають можливість створювати іменовані ділянки пам’яті через інтерпретатор BASH, та мати до них доступ. Як зазначено у попередніх розділах, мова програмування SBASH відноситься до строго-типізованого класу. А Через це, вона вимагає вказати тип змінної, над якою буде проводитися маніпуляції алгоритму програми. Окрім цього, існують додаткові модифікатори змінних, що дозволяють сховати за ім’ям одної змінною іншу (посилання) чи створити серію змінних використовуючи одне ім’я (колекцію). Також, мова програмування SBASH має типи даних, що програміст може створити сам, такі як структури та перерахування. Виходячи з того, що кількість змінних можуть варіюватись від одної (не має сенсу писати декларування змінної без жодного її інстансу) та більше (без обмеження зі сторони граматики), потрібно реалізувати можливість вказувати гнучку кількість змінних

при декларації. Ґрунтуючись на зазначеному, маємо наступний синтаксис: <тип створюваної змінної (змінних)> [<ім’я інстансу змінної>, …], де:

- тип змінної, може бути будь-який built-in type, або ім’я вже визначеної структури чи перерахування;
- ім’я інстансу може складатись із будь-яких латинських літер, чи цифр, символів тире, нижніх підкреслювань, тощо (але повинно починатись із літери);
- всі частини розділяються символом пробілу;
- кутові душки не враховуються, а лише вказують на місце для вставки коду.

Окрім типів даних є модифікатори, що додають вище зазначеним built-in types, а також створеним користувачем – типам даних: нових можливостей. Як було зазначено у попередньому розділі, такими модифікаторами є: ref – утворює посилання із звичайної змінної (тобто не утворює нову змінну, а лише надає додаткове ім’я, щоб працювати із нею), const – додає константність змінній, та [] – створює колекцію замість однієї змінної.

Їх можна комбінувати, але за деякими обмеженнями. Розглянемо всі можливі комбінації цих модифікаторів:

- const та ref, чи ref та [], чи const та ref та []: якщо посилання створюється на константну змінну та (чи) колекцію, воно повинно бути також – константне та (чи) мати ознаки колекції відповідно;
- const та []: створення константної колекції, обов’язково потребує ініціалізації (на даний час, передбачено синтаксисом, але неможливе на етапі POC).

Виходячи з того, що константність змінній обмежує зміну її значення після ініціалізації, а створення посилання вимагає змінну на котру посилатись, то при застосуванні цих модифікаторів типів, необхідно одразу ініціалізувати задекларовані змінну змінні. Ґрунтуючись на цьому, маємо наступну модифікацію синтаксису створення змінних: <модифікатор типу> <тип створюваної змінної (змінних)> <модифікатор колекції> [<ім’я інстансу змінної> = <вираз>, …], де

- модифікатор типу: може бути const чи (та) ref;
- модифікатор колекції: може бути [];
- вираз: зазначена далі у пунктах – комбінація літералів, операторів та змінних.

Приклад створення змінної, можна побачити на Рисунку 3.1, нижче.

```
1 const     int ivar1 = 1        // Constant varible
2 const ref int ivar2 = ivar1    // Reference varible
3 Data[] dvar1, dvar2            // Array variables
```

Рисунок 3.1 – Приклад декларації змінних

До змінних можна зазначити наступні вимоги, щодо компіляції:

– ім’я кожної наступної змінної, повинно бути – унікальним щодо поточного простору імен;
– значення змінної, повинно бути сумісним із її типом (включно із модифікаторами типу змінної).

3.1.3 Вирази та операції призначення

Як було зазначено вище, змінні можуть ініціалізуватись виразами, які у свою чергу є комбінацією: літералів, змінних, та операторів. Також, вже задекларовані змінні, можуть змінювати своє значення, завдяки оператору призначення, що був продемонстрований вище, та котрий має означення символом «=». Цей оператор задає деяке значення (що справа від нього) змінній (що стоїть зліва від нього). Саме цим значенням і є деякий «вираз».

Слід розуміти, що неможливо призначити змінній будь-який вираз. Вираз має повертати деяке значення, що буде відповідати типу змінної і тоді, та тільки тоді може бути призначеним їй. Тип – повернений виразом визначається операторами використаними у ньому, а точніш – останнім, застосованим.

У мові програмування SBASH, було імплементовано базові оператори, для обчислення математичних та логічних виразів, також деякі з них були використані для маніпуляціями над built-in type – string, а також колекціями. На наступній Таблиці 3.1 – наведено всі оператори, типи даних, що можливо використовувати з ними, та їх призначення. Надалі змінними a та b у контексті операторів – називають змінні зліва та справа від оператора, відповідно.

Таблиця 3.1 – Оператори мови програмування SBASH.

Оператор	Типи даних із якими використовується	Повертає тип даних	Кількість аргументів	Призначення
+	int, double	int або double	2	Додає число a до числа b
	string	string	2	Конкатенує рядок a до рядка b
-	int, double	int або double	2	Віднімає число b від числа a
*	int, double	int або double	2	Перемножує число a на число b
/	int, double	int або double	2	Ділить число a на число b
%	int, double	int або double	2	Знаходить залишок від ділення числа a на число b
^	int, double	int або double	2	Зводить число a до степеня b
++	int, double	int або double	1	Інкрементує число a (чи b)
--	int, double	int або double	1	Декрементує число a (чи b)
>	int, double	bool	2	Порівнює чи, число a більше за число b
<	int, double	bool	2	Порівнює чи, число a меньше за число b
<=	int, double	bool	2	Порівнює чи, число a більше або дорівнює за число b
>=	int, double	bool	2	Порівнює чи, число a меньше або дорівнює за число b
or	bool	bool	2	Порівнює чи хоча б одне із значень є істинним
and	bool	bool	2	Порівнює чи обидва із значень є істинними
(	–	–	–	Оператори порядку виконання дій.
)				

Тип поверненого значення, може залежати від необхідних обчислень, та буде найменш необхідним, для збереження даних, наприклад, для десяткового дробу буде використано double, але не int.

Безпосередньо сам вираз, має бути представлений у компіляторі завдяки польському нотатку, що надає найбільш просте уявлення комплексних структур, що, зазвичай, базується на стековому представлені порядку операцій виразу із операторами та аргументами. Трансляція [11] виразу у польський запис може бути зроблена як під час будови AST, так і під час компіляції.

```
1 int    ivar = 7 * (5 - 2) ^ 2       // Assign mathematical value
2 bool   bvar = true and (7 > 10)    // Assign logical value
3 string svar = "text" + "text"      // Assign string value
```

Рисунок 3.2 – Приклад використання виразів для ініціалізації змінних

Приклад використання виразів, можна побачити на Рисунку 3.2, вище. Також, як альтернативу оператору «=», можна використовувати оператори «+=» (додати та призначити) чи «-=» (відняти та призначити), що виконують операцію додавання, чи віднімання від змінної a – змінної b, та призначують отримане значення змінній a.

Ґрунтуючись на вище зазначеному, можна виділити наступні правила компіляції виразів:

– аргументи операторів, повинні відповідати доступним для маніпуляції типам;
– тип результату виразу, повинен відповідати типу змінної, якій призначається значення, або бути сумісним з ним (як int та double).

3.1.4 Перерахування

Перерахування як тип даних, вже був описаний у попередньому розділі. Як було зазначено, це сукупність задекларованих користувачем літералів. Декларація перерахування задається ключовим словом «enum», та має наступний вид: enum <ім’я перерахування>, де до ім'я ставляться такі самі вимоги, як і до ім'я інстансу змінної.

У своючергу, ллітерали знаходяться у локальному під-просторі імен, тому можуть бути не унікальними серед інший перерахувань. Щоб виділити цей під-простір, використовуються фігурні дужки (тобто символи «{» та «}»), поставлені з нового рядка після декларації перерахування. На рядках із дужками, не має бути нічого іншого.

Кожен літерал із перерахувань має такі самі вимоги до іменування, як і ім’я інстансу звичайної змінної. Попре те, перерахування може мати значення літералів isbuilt-in types, набір котрих повинен бути однаковим (в іншому разі поклик до різних літералів буде повертати різний результат, що буде суперечити

ідеології SBASH), для всіх літералів перерахування. Таким чином, можна сформувати синтаксис, що буде визначати форму декларації літералу перерахування, та його значень: <ім'я літералу перерахування>, або <ім'я літералу перерахування> = [<значення літералу перерахування >, …], де: значення літералу перерахування – розділені комою built-in літерали. Кожен новий літерал задається із нового рядочку.

Ґрунтуючись на вищезазначеному, можна сформувати наступні правила компіляції, перерахувань:

– ім'я перерахування, повинно бути унікальним;
– ім'я літералу перерахування повинно бути унікальним серед набору всіх літералів перерахування;
– набір built-in літералів для всіх літералів перерахування повинен бути однаковим.

```
1 enum COLOR
2 {
3   RED   = 0, "RED"
4   GREEN = 1, "GREEN"
5   BLUE  = 2, "BLUE"
6 }
```

Рисунок 3.3 – Приклад декларації перерахування

Приклад декларації перерахування наведено на Рисунку 3.3, вище. Для перерахування без заданих значень, як було зазначено, можна просто відкинути праву частину із оператором призначення для всіх літералів перерахування.

3.1.5 Структура

Як і перерахування, структура є заданим користувачем – типом даних. Але на відміну від нього, зберігає значення змінних. У наслідок цього, та виходячи із твердження, що ідеологією мови програмування SBASH є єдиний синтаксис – можна стверджувати, що змінні у структурі будуть задаватись за тими самими правилами, що і звичайні. Але треба пам'ятати, що за POC, такий функціонал як

ініціалізація змінних, а також модифікатори типів для них, може, а також і є спрощеним.

Декларація ж структури задається за тими самими правилами, що і перерахування, за умови, що використовується ключове слово «struct», замість «enum». Структура також має свій під-простір імен для декларації так званих атрибутів структури – змінних, що декларуються у межах структури.

Як зазначено у попередніх розділах – структура може успадковувати інші структури, тобто – агрегувати їх атрибути, тим самим обмежуючи варіативність назв змінних у своєму локальному під-просторі імен. Тим паче, що і саме спадкування інших структур повинно виконуватись за правилами – збереження унікальності атрибутів, а також супротив дублювання непрямого спадкування, тобто обмеження на пряме спадкування структури вже спакованої із інших структур. Із цього маємо синтаксис декларації структури даних із спадкуванням: struct <ім'я структури> : [<ім'я структури до спадкування>, …], де ім'я структури до спадкування – вже задекларовані структури. А також маємо наступні правила компіляції:

– успадковані структури повинні бути задекларовані;
– успадковані структури даних не повинні напряму успадковувати структури, що є в наявності у ланцюжку успадкування інших батьківських структур;
– ім'я успадкованих структур повинні бути унікальними;
– ім'я атрибутів всіх успадкованих структур, а також поточної структури повинні бути унікальними;

```
1 struct Base
2 {
3     int iattr
4 }
5
6 struct Derived : Base
7 {
8     bool battr
9 }
```

Рисунок 3.4 – Приклад декларації структури, та успадкування

На Рисунку 3.4, вище, наведено приклад декларації структури. Як було зазначено у попередніх розділах – проблему ромбовидного спадкування, компілятор має вирішувати автоматично – зберігаючи унікальний інстанс успадкованої структури, а не дублюючи його.

3.1.6 Функція

Функція – у межах мови програмування SBASH, це просто іменована частка коду, доступна для виконання необмежену кількість разів за її назвою. Декларація функції зумовлює декларацію її ім'я, параметрів, значення до повернення, а також, її, так званого «тіла», тобто безпосередньо коду, що вона виконує.

Декларація параметрів, виконується по одному, так само, як декларація змінних, за виключенням того, що параметр має мати тільки один інстанс. Параметри приводяться у круглих душках, та перераховуються через кому, їх можна не декларувати зовсім, або довільну кількість.

Зворотне значення, може бути задане, також – опціонально, але його наявність примушує користувача на використання конструкції «return» у тілі функції.

Сама декларація функції починається із ключового сова «func». Надалі надається синтаксис задання функції із параметрами, та зворотнім значенням: func <ім'я функції> ([<параметр функції>,…]) : <тип зворотного значення>, де параметр функції – дозволені всі модифікатори типів, що не потребують ініціалізації.

```
1 func function(int iparam, ref string param) : Data
2 {
3     // Function body
4 }
```

Рисунок 3.5 – Приклад декларації функції із параметрами та зворотнім значенням

Приклад декларації функції наведено на Рисунку 3.5, вище. Слід зауважити, що згідно із положенням про POC, зворотнім значенням функції не може бути колекція. Ґрунтуючись на вищезазначеному, можна сформувати наступні правила компіляції:

– ім’я параметрів функції повинно бути унікальним як у глобальному просторі імен, так і у локальному;
– тип зворотного значення повинен бути задекларованим, як і тип параметрів.

Слід пам’ятати, що параметр зазначений як посилання, може впливати на значення змінної призначеної йому при виклику функції.

3.2 Проектування конструкцій тіла функції

Тіло функції, це колекція деяких інструкцій призначених до виконання у межах виконання цієї функції. У даному пункті розглядається як правила компіляції цих конструкцій, так і основні вимоги до базової структури тіла функції. Слід розуміти, що з точки зору лексичного та граматичного аналізу – немає ніяких обмежень щодо змісту тіла функції, на відміну, на приклад, від типів даних користувача, таких як структури та перерахування, де є чіткі вимоги. Окрім цього, деякі конструкції тіла функції можуть мати можливість містити такі самі конструкції, на котрий поширюються такі самі правила, як і на них самих, окрім специфічних для самого тіла функції, описаних нижче. Доступні для використання конструкції у тілі функції:

– декларація та призначення значень змінним (без виключень, відповідно до правил зазначених у пунктах вище);
– оператори умовного переходу;
– цикли.

Декларація структур даних, а також функцій – неможлива у тілі функції за будь-яких умов.

Окремою інструкцією, можливою до вказання у тілі функціє є «return», що зобов’язан бути вказаний, у випадку наявності зворотнього значення. Синтаксис цієї конструкції повинен бути наступний: return <зворотнє значення>, де зворотнім значенням повинно бути змінна чи вираз (який обмежується згідно із РОС), тип значення котрих має повністю відповідати задекларованому типу зворотнього значення функції. Таким чином маємо наступні правила компіляції:

– тіло функції повинно містити виключно допустимі конструкції;
– необхідність наявності інструкції «return» визначається наявністю зворотнього значення у декларації функції;

- у випадку наявності інструкції «return», вона повинна бути останньою у тілі функції.

3.2.1 Оператор умовного переходу «if»

Оператор умовного переходу, це конструкція, що надає можливість надавати алгоритму різних сценаріїв виконання (тобто розгалуження), у залежності від деякої умови. Умова може задаватись як змінною типу bool, або будь-яким виразом, результатом котрого є значення типу bool. Згідно із POC, вираз буде спрощено до використання змінної типу bool, але незмінною вимогою є наявність функціонального блоку для будь-якої форми цього умовного оператору.

У мові програмування SBASH реалізовано повну форму умовного оператору «if» (від англ. «якщо»), тобто вона зумовлює декілька секцій, якими є додаткові умовні блоки «elif» (від англ. «else if» – «в іншому разі, якщо») чи «else» (від англ. «в іншому разі»). У свою чергу «elif» надають таких самих можливостей як і «if», за виключенням того, що можуть існувати лише у межах продовження оригінального умовного оператора та йти за ним. Заключним оператором умовного переходу може виступати «else», що як і «elif» – не є самодостатнім на відміну від «if», та має пріоритет нижче за «elif». Таким чином можна зазначити синтаксис операторів умовного переходу: <оператор умовного переходу> (<вираз>), де:

- оператор умовного переходу – може бути, як було зазначено вище, тільки «if» чи «elif»;
- вираз – може бути будь-який вираз, результатом котрого є значення типу bool, або змінна bool.

Оператор умовного переходу «else», не потребує ніякої умови, для виконання, бо виконується у будь-якому разі, що не підлягає для батьківського умовного оператору «if», або «elif».

Із зазначеного вище, отримаємо наступні правила компіляції, умовного оператору переходу «if», із додатковими конструкціями «elif» та «else»:

- вираз, що використовується як умова оператору, повинен бути валідним (відповідати правилам компіляції виразу);
- результатом виразу, що використовується як умова оператору, повинно бути виключно значення типу bool;
- зміст коду до виконання, валідується відповідно правилам використаних там конструкціям.

На наступній сторінці, можна ознайомитись із прикладом оператору умовного переходу «if», на Рисунку 3.6.

```
1 func foo()
2 {
3     const int ival1 = 0, ival2 = 2
4     switch( ival1 )
5     {
6         case 0
7         {
8             // case code here...
9         }
10        case 1
11        case ival2
12        {
13            // case code here...
14        }
15        default
16        {
17            // default code here...
18        }
19    }
20 }
```

Рисунок 3.6 – Приклад умовного оператору переходу із використанням «if», «elif» та «else» ключових слів

3.2.2 Оператор умовного переходу «switch»

Оператор умовного переходу «switch», працює за схожим принципом, як і оператор умовного переходу, але за виключенням того, що має велику кількість функціональних блоків до виконання. Основним принципом цієї коснструкції є навігація по різним керуючим блокам (так званим «case»), базуючись на умовному значенні. Умовою може виступати будь-яка змінна, чи вираз, результатом котрої є built-in type. У свою чергу – вибір функціонального блоку до виконання є послідовне порівняння керуючого значення із значенням вказаним у «case», котрим може виступати будь-який built-in літерал, або змінна типу built-in. Згідно із POC, виконується обмеження щодо цього і керуючим значенням може виступати – виключно, змінна. Із цього маємо синтаксис оператору умовного переходу «switch»: switch (<керуюче значення>), де керуючим значенням може бути як змінна, так і вираз.

У свою чергу функціональні блоки, цього оператору умовного переходу, можуть бути згруповані, заради вказання одного функціонального блоку для

декількох значень. Однак, у фінальному вигляді, вони повинні мати хоча б один функціональний блок, інакше – вони не мають бути вказаними. Таким чином, отримаємо синтаксис «case» конструкції: case <керуюче значення>, де керуюче значення – виключно літерал, або змінна із built-in type.

Альтернативою для «case» конструкції може бути конструкція «default», яка має бути вказана виключно останньою, за зобов'язана мати функціональний блок. Ця конструкція не має жодного керуючого значення, та призначена для виконання коду для будь-яких значень, що не були специфіковани конструкціями «case».Не може бути вказана у групі із «case» конструкціями, бо це не має сенсу.

```
1 func foo()
2 {
3    const int ival1 = 0, ival2 = 2
4    switch( ival1 )
5    {
6       case 0
7       {
8          // case code here...
9       }
10      case 1
11      case ival2
12      {
13         // case code here...
14      }
15      default
16      {
17         // default code here...
18      }
19   }
20 }
```

Рисунок 3.7 – Приклад оператору умовного переходу «switch»

Приклад оператору умовного переходу «case», наведено на Рисунку 3.7, вище. Слід пам'ятати, що наявність блоку «default», не виключає необхідності у наявності хоча б одного блоку «case».

Грунтуючись на вище зазначеному, можна визначити наступні правила компіляції оператору умовного переходу «switch»:

- керуюче значення оператору умовного переходу, повинно бути змінною, або виразом результатом котрого є значення built-in type;
- зміст функціонально блоку оператору умовного переходу «switch» повинен містити виключно конструкції «case» у кількості від одної, або

більше, або опціональну конструкцію «default», що повинна бути оголошена після всіх конструкцій «case»;
- керуючим значенням блоків «case», повинні бути виключно літерали, або змінні із built-in type;
- груповані, або одинарні блоки «case» зобов'язані мати функціональний блок, як і блок «default»;
- зміст коду до виконання, валідується відповідно правилам використаних там конструкціям.

3.2.3 Цикли

Цикли – це такі конструкції у мові програмування SBASH, що надають можливості: виконувати код специфікованих у їх функціональному блоці довільну кількість раз у залежності від умови специфікованої у декларації цього циклу. В мові програмування SBASH існують чотири різновиди циклів із різною функціональністю, але всі вони мають декілька єдиних вимог щодо компіляції:
- всі використані вирази у циклах повинні відповідати компіляційним правилам них самих;
- зміст коду до виконання, валідується відповідно правилам використаних там конструкціям.

Також, слід зазначити, що у циклах можуть бути використана команда «break» (від англ. «перервати»), що зупинить виконання циклу незалежно від умови ітерування циклу.

3.2.3.1 Цикл «for»

Цикл «for», є циклом із можливістю декларування локальних змінних, а також інструкцій щодо їх зміни після кожної ітерації. Загальним синтаксисом такого циклу є: for (<змінні циклу> : <умова виконання> : <інструкції щодо виконання після ітерації>), де:
- змінні циклу – звичайна декларація змінних;
- умова виконання – вираз, результатом котрого повинно бути значення типу bool;
- інструкції після виконання ітерації – вираз, який виконується після кожної ітерації.

```
1 func foo()
2 {
3     for( int i = 0; i < 10; ++i )
4     {
5         // for code here...
6     }
7 }
```

Рисунок 3.8 – Приклад декларації циклу «for»

Приклад декларації циклу «for», можна побачити на Рисунку 3.8, вище. Дана конструкція циклу має велику кількість вкладених конструкцій, таких як вирази – це не дає можливості її реалізації до POC.

3.2.3.2 Цикл «for each»

Цикл «for each» є найпростішим, що надає можливість ітеруватись через колекції не використовуючи індексу. Єдиними вимогами щодо використаних у ньому значень є наявність модифікатору колекції у переданій їй змінній, що повинна містити колекцію. Згідно із цим маємо наступний синтаксис: for (<змінна із елементом колекції> : <змінна колекції>), де:

– змінна із елементом колекції – це унікальне ім’я, що буде використовуватись для зберігання значення елемента колекції у даній ітерації;
– змінна колекції – це змінна що зберігає колекцію, також може бути і виразом, результатом котрого є колекція, але це не передбачено за POC.

```
1 func foo()
2 {
3     for( entry : collection )
4     {
5         // for code here...
6     }
7 }
```

Рисунок 3.9 – Приклад декларації циклу «for each»

3.2.3.3 Цикл «while»

Цикл «while» є найбільш наближеним до операторів умовного переходу, бо все що він має, це умову виконання ітерацій циклу, а синтаксис цього циклу має наступний вигляд: while (<умова виконання>), де умовою виконання може виступати як змінна, так і вираз, але останній ігнорується у імплімментації згідно із POC. Синтаксис цього циклу, нічим, окрім ключового слова – не відрізняється від операторів умовного переходу: while (<умова виконання>), де умова виконання може бути як змінна, так і вираз, за виключенням того, що останній не оброблюється згідно із POC.

```
1 func foo()
2 {
3     while( true )
4     {
5         // while code here...
6     }
7 }
```

Рисунок 3.10 – Приклад декларації циклу «while»

Особливо у такому типі циклу (див. Рисунок 3.10), слід пам’ятати про можливість створити цикл, що не буди мати фізичного кінця, та при необхідності використати операцію «break».

3.2.3.4 Цикл «do-wile»

Цикл типу «do-while», майже не має відмінностей від циклу «while», за виключенням тої, що перевірка умови виконання циклу, виконується після ітерації (як це виконується у всіх інших типах циклів). Синтаксис цього циклу не вказується через повну схожість із циклом типу «while», але із наявністю ключового слова «do», та наявності функціонального блоку над інструкцією умови виконання.

Треба пам’ятати, що використання даного типу циклу, навіть при неможливості виконання умови його ітерації (див. Рисунок 3.11), призведе до хоча б єдиної обробки вказаних інструкцій його функціонального блоку.

```
1 func foo()
2 {
3     do
4     {
5         // do-while code here...
6     }
7     while( false )
8 }
```

Рисунок 3.11 – Приклад декларації циклу «do-while»

3.3 Базова структура програми

Базова структура програми передбачає наявність деяких правил, згідно із котрими можна валідувати наявні конструкції. Одним із головних правил є наявність функції під назвою «main» (від. англ. «головна»), що зумовлює початок алгоритму виконання програми. Без цієї функції, мова програмування SBASH, не буде мати можливості – зрозуміти, де починати виконання програми. Функція «main», зобов’язана мати зворотнє значення типу int, що буде повернено операційній системі як код виконання програми. Параметрами цієї функції є фіксовані для визначення аргументів, або їх повна відсутність. У разі їх існування, до них поставляються наступні умови:

- кількість параметрів повинна дорівнювати двом;
- перший параметр повинен мати виключно тип даних «int»;
- другий параметр повинен мати виключно тип даних «string» із модифікатором типу – колекція.

Параметри функції «main» виставляються автоматично – мовою програмування SBASH, як і її виконання. Вони зберігають аргументи із котрими було виконано скомпільований скрипт BASH.

Наступним правилом структури програми написаної мовою програмування SBASH є допустимі конструкції у глобальному просторі імен програми, якими є виключно:

- декларація змінних;
- визначення структур та перерахувань;
- визначення функцій.

Всі інші конструкції у глобальному просторі імен програми є семантичною помилкою. Приклад програми написаної мовою програмування SBASH, можна побачити на Рисунку 3.12, нижче.

```
 1 struct PREFERENCES
 2 {
 3    const int success_result = 0
 4    const int failed_result  = 1
 5 }
 6
 7 func getSuccess() : int
 8 {
 9    PREFERENCES preferences
10
11    return preferences.success_result
12 }
13
14 func main(int argc, string[] argv) : int
15 {
16    // main code here...
17
18    return getSuccess()
19 }
```

Рисунок 3.12 - Приклад програми написаної мовою програмування SBASH

4 ПРОГРАМНА РЕАЛІЗАЦІЯ КОМПІЛЯТОРА SBASH

Цей розділ описує архітектуру, та основні особливості імплементації компілятора мови програмування SBASH. Як було зазначено у попередніх розділах, мова програмування C++ була обрана для імплементації компілятора. Згідно із цим, UML діаграми [12] у даному розділі мають характерні особливості притаманні цій мові програмування, наприклад – конструктори. Окрім того, також зазначені деякі типи даних задекларовані у стандартній бібліотеці C++ 17 стандарту.

4.1 Глобальна архітектура компілятора

Компілятор SBASH, складається із ядра, що реалізує основну бізнес-логіку, а також із деяких бібліотек, що використовуються у ядрі. Такими бібліотеками є:

– sdk – зберігає головні типи даних, що використовуються у всьому проекту;
– common – імплементує додатковий функціонал, такий як створення журналів виконання, що використовуються для процесу налагодження при наявності помилок програмі, а також дає можливість їх ідентифікувати.

Зазначений функціонал, був винесений до окремих бібліотек через такі фактори, як можливість модифікації чи активації та дезактивації певного функціоналу без модифікації коду основної бізнес-логіки. А також через те, що може бути використаний у багатьох місцях проекту.

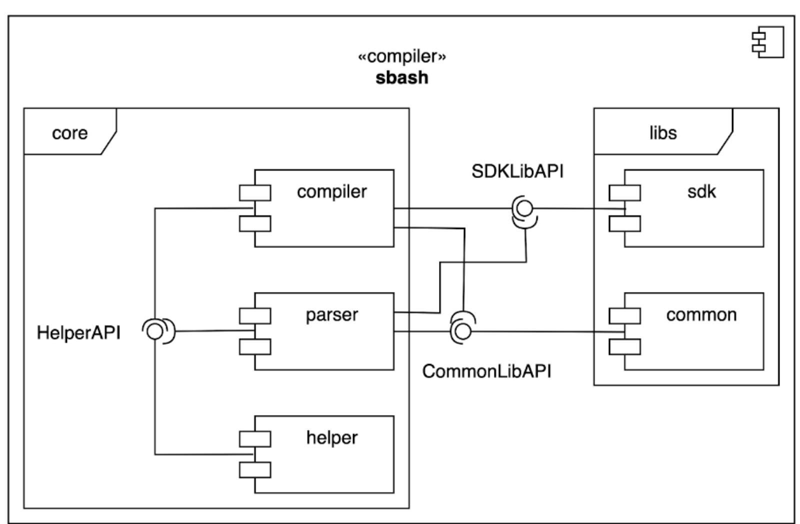

Рисунок 4.1 – Діаграма компонентів компілятору SBASH

На Рисунку 4.1, зображені головні компоненти компілятору SBASH, окрім вже зазначених бібліотек, можна побачити три основні компоненти, що використовує ядро компілятора, а саме: compiler, parser, helper. Призначення перших двох, вже відомо із попередніх розділів, вони виконують лексичний, граматичний, а також семантичний аналіз коду. Також компонент compiler – виконує побудову програми, тобто APS (див. пояснення далі), а також як результат створює фінальній BASH скрипт. У свою чергу, компонент helper = репрезентує множину класів, а також функцій, що використовуються у компонентах compiler та parser. Зазначені інтерфейси на діаграмі не мають прямої репрезентації у коді, бо є узагальненням стеку інтерфейсів кожного із компонент, спрощених для більш зрозумілої демонстрації.

4.2 SDK компілятора

Дана бібліотека уособлює у собі основні структури даних, що використовуються у компіляторі SBASH. Ці структури даних уособлюють у собі результати основних двох процесів компілятора, а саме: граматичного та семантичного аналізу.

На Рисунку 4.1, нижче, зображена одна із двох структура даних, що використовуються як результат граматичного аналізу коду на мові SBASH.

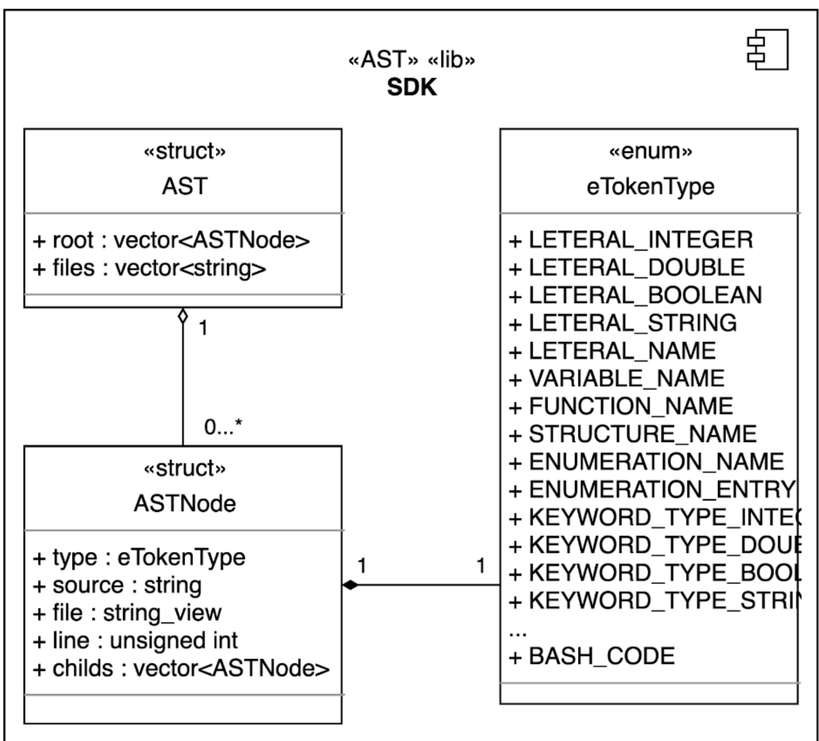

Рисунок 4.2 – Діаграма класів для AST структур у SBASH SDK

Іншими словами, ця структура описує собою граматично правильний код написаний мовою програмування SBASH, тобто репрезентує собою таку структуру даних як дерево, тому і називається AST, тобто Abstract Syntax Tree (від англ. «абстрактне синтаксичне дерево»). Згідно із цією структурою даних, її найменьшим будівельним блоком є AST Node (від англ. «вузол»), який описує наймаліший лексичний елемент коду і зберігає таку інформацію як:

– type – тип даного вузла, що задається перерахуванням eTokenType;
– source – текстове представлення значення цього вузла отримане із коду;
– file – посилання на змінну, що зберігає ім'я файла котрому належить цей вузол;
– line – номер рядка на котрому знаходиться цей вузол;
– childs – згідно із деревовидною структурою даних, може мати дітей.

У свою чергу, структура AST, є просто кореневим об'єктом що зберігає абстрактне синтаксичне дерево, а також список файлів із котрих були отримані вузли, як результат лексичного аналізу, тим самим зберігаючи місце у пам'яті.

Другою, та останньою структурою даних у SBASH SDK є APS тобто Abstract Program Structure (від англ. «абстрактна програмна структура»), зображена на Рисунку 4.3, нижче. Вона репрезентує результат семантичного аналізу компілятора. Ця структура даних має схожу із деревом семантику, але фіксовану глибину дочірніх вузлів. Кожен вузол має наступні атрибути:

– opcode – значення перерахування eOpcode, що описує всі необхідні операції доступні у BASH [13], для репрезентації SBASH коду;
– parameters – колекція аргументів, що необхідні конкретній операції BASH.

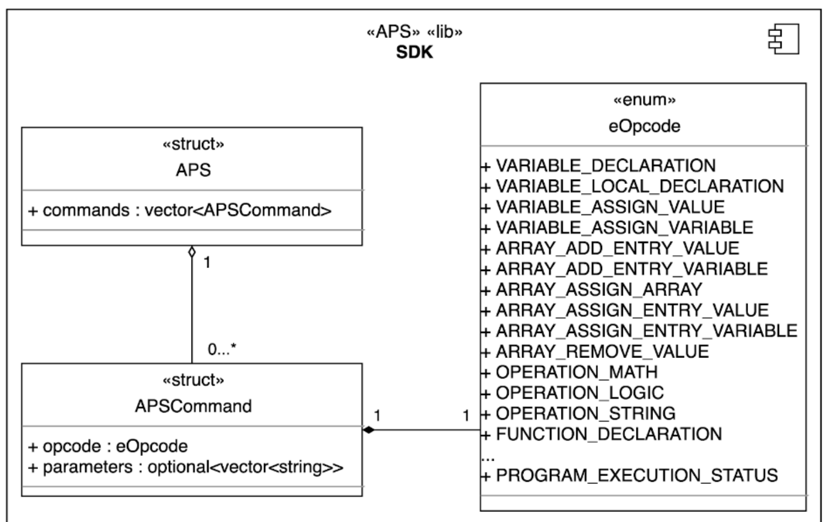

Рисунок 4.3 – Діаграма класів для AST структур у SBASH SDK

4.3 Архітектура ядра компілятора

Ядро компілятора SBASH, складається із декількох основних компонентів. Кожен компонент виконує специфічний тип операцій, та взаємодіє із іншими. У ядрі є наступні компоненти:

- drivers – компонент, що визначає основний шлях компіляції, тобто послідовність обробки даних у інших компонентах, та її акумуляція, зміну контексту;
- resources – компонент, що відповідає за зберігання контексту компіляції, та зберігання ресурсів необхідних декільком незалежним компонентам;
- rules – компонент, що відповідає за валідацію даних певного формату згідно із контекстом компіляції програми тобто відповідно до даних компонента resources;
- builders – це компонент, що компілює вже провалідований код SBASH із AST формату до код BASH у APS формат.

Всі компоненти вказані у множині, через те, що кожен із них є групою класів із єдиним інтерфейсом, що відповідає за певний функціонал, вони представлені на Рисунку 4.4, нижче.

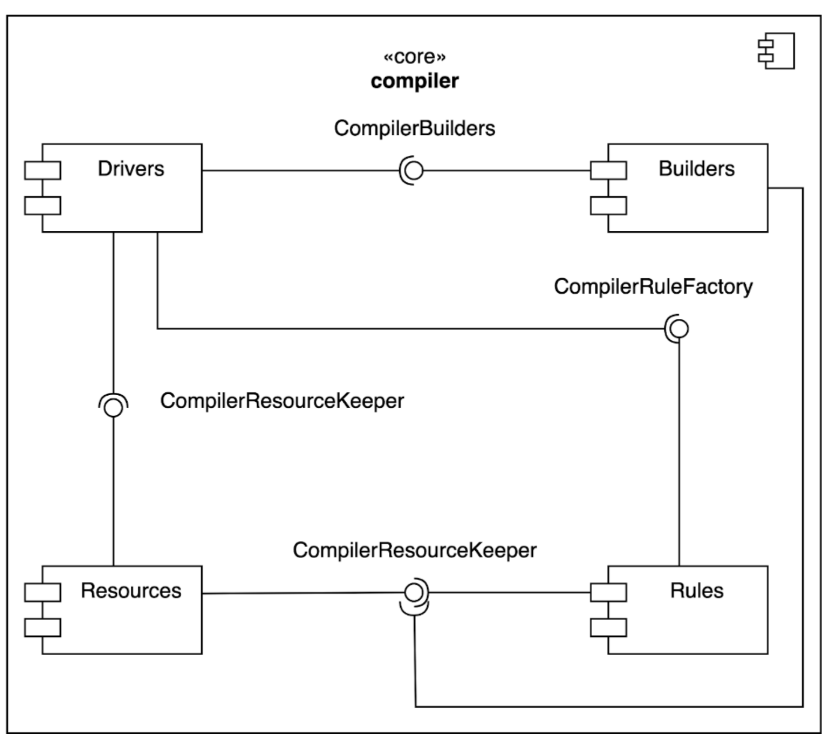

Рисунок 4.4 – Діаграма компонентів ядра компілятора SBASH

Як було зазначено вище, ядро компілятора складається із декількох компонентів. Всі ці компоненти єднаються єдиною сутністю – класом компілятора, що постає фасадом (див. Рисунок 4.5, нижче), через який можна працювати із усіма компонентами компілятора.

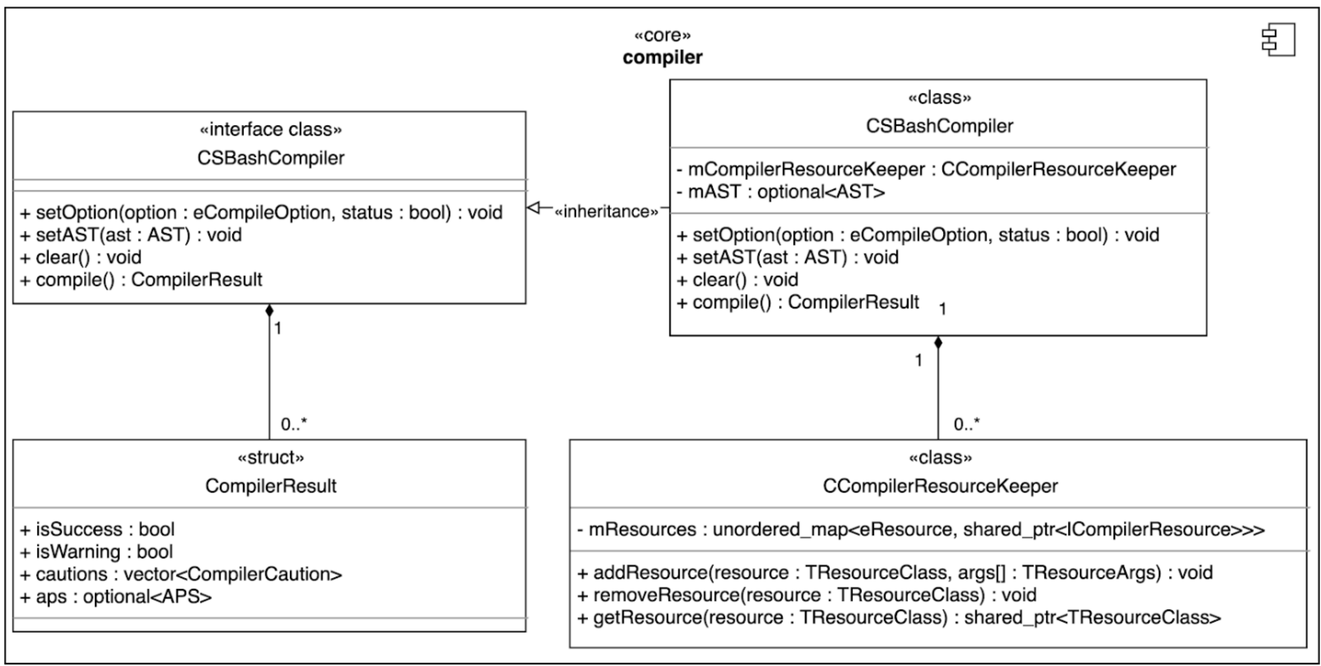

Рисунок 4.5 – Діаграма класів ядра компілятора SBASH

Слід зазначити, що ця сутність не використовується на пряму через клас із імплементацією, а має абстрактний клас-інтерфейс (на діаграмі не зазначено, заради дотримання лаконічності). Тобто у доступних до підключення файлах із кодом С++, в наявності лише цей клас-інтерфейс, а також функція, що створює його із класом імплементації, її код наведено на Рисунку 4.6, нижче. Таким чином, можна досягти інкапмсуляції імплементації за інтерфейсом не вдаючись до шаблону проектування «pimpl».

```
 1 namespace sbash
 2 {
 3     namespace compiler
 4     {
 5         std::shared_ptr<ISBashCompiler> makeCompiler()
 6         {
 7             return std::shared_ptr<ISBashCompiler>( new CSBashCompiler{} );
 8         }
 9     };
10 };
```

Рисунок 4.6 – Приклад створення класу інкапсульованого у інтерфейсі.

4.3.1 Компонент ядра компілятора: drivers

Компонент drivers (від англ. «керуючі»), як було зазначено, є сукупністю множини класів, що відповідають за керування процесом компіляції коду. Якщо роздивитись з точки зору ядра компілятора та бізнес-процесу, то цей компонент визначає, тип вхідних даних, а згідно із ним, приймає рішення, або передати виконання іншому драйверу, або продовжити виконання, а саме:

- виконати валідацію силами компоненту rules;
- занести відповідні данні за необхідністю до компоненту resources;
- при коректній валідації – виконати компіляцію коду у компоненті builders.

Собою, цей компонент реалізує шаблон проектування – команда, тобто має деякі вхідні дані, а згідно до них виконує відповідний, вже готовий, сценарій. Слід зазначити, що такими вхідними даними є AST, також цей компонент може виокремлювати частину AST, та передавати до дочірніх драйверів як вхідні дані. Звісно, вихідними даними буде APS отримане з компонента builders. На Рисунку 4.7, нижче, представлені основні класи драйверів.

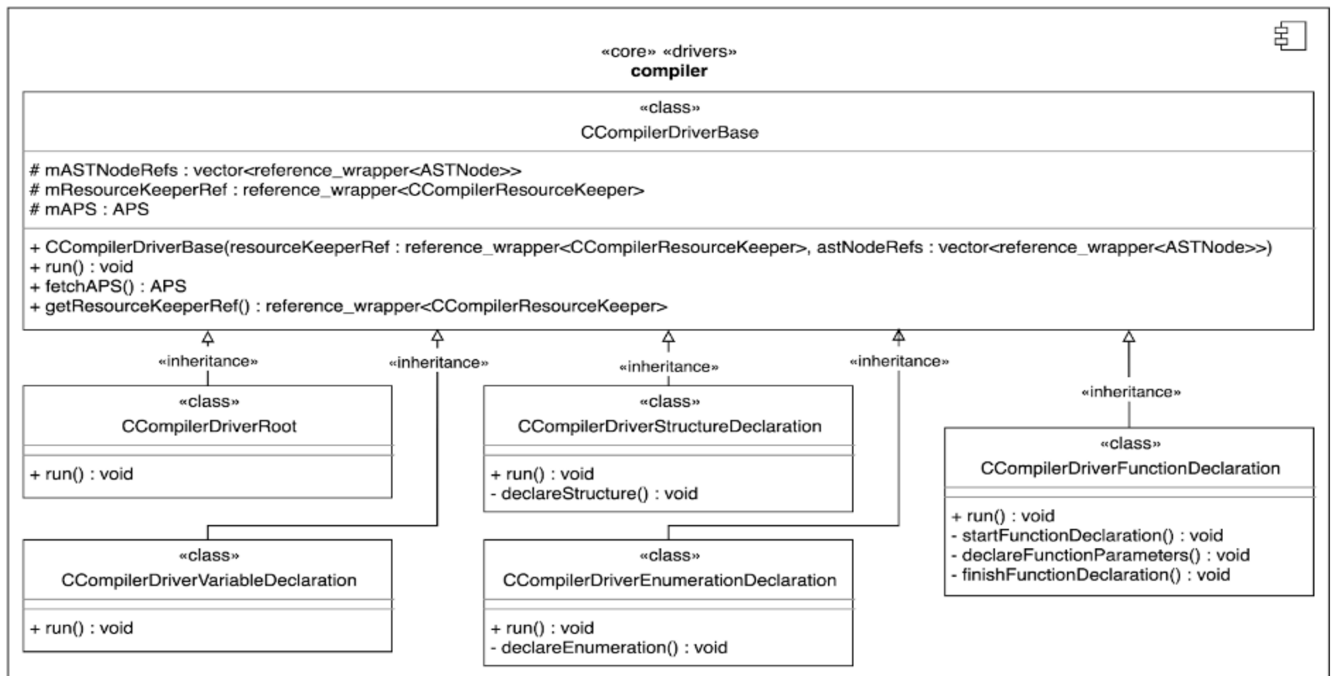

Рисунок 4.7 – Діаграма класів компонента ядра компілятора – drivers

Розглянемо головні класи драйверів (драйверів існує більше, але наведені лише першого та другого шару виконання), та за що вони відповідають.

Драйвер CCompilerDriverRoot – цей клас відповідає за обробку AST програми, що була отримана після граматичного аналізу всіх вхідних файлів програми. Головною метою цього драйвера є ініціація перевірки конструкцій допустимих у глобальному просторі імен, а також відокремлювання частин AST для передавання їх до другорядних драйверів як вхідні дані. Цей драйвер знаходиться на найвищому шару виконання, тобто є кореневим (що і відображено у його назві).

Драйвер CCompilerDriverVariableDeclaration – цей клас відповідає за обробку відокремленої частини AST дерева, що містить виключно декларацію змінної, він ініціює виконання валідації цієї змінної компонентів rules, а також побудову для неї APS можливостями компоненту buidlers. Окрім того, він додає відповідні данні до компоненту resources.

Драйвер перерахування CCompilerDriverEnumerationDeclaration та драйвер структури CCompilerDriverStructureDeclaration відповідно, виконують, той самий функціонал, що і драйвер CCompilerDriverVariableDeclaration, але специфічний для декларації відповідних структур даних, а також не використовують компонент builders, бо не створюють APS інструкцій.

Драйвер CCompilerDriverFunctionDeclaration ініціює перевірку декларації функції, а також її параметрів через компонент rules, будує її початок та завершення через компоненту builders, а також додає відповідні відомості до компоненти resources. Слід зазначити, що він також відокремлює частини AST – що є операціями тіла функції, та передає їх як вхідні дані драйверам третього рівня виконання.

4.3.2 Компонент ядра компілятора: resources

Компонент resources (від англ. «ресурси») – є сукупністю класів, що зберігають деякі дані (згідно із призначенням), та мають єдину точку доступу. Єдина точка доступу дає можливість взаємодіяти із усіма ресурсами доволі просто, використовуючи лише необхідні. Згідно із архітектурою, такою точкою виступає клас CCompilerResourceKeeper, та потребує включення лише тих заголовкових файлів, ресурси котрих – необхідні, тим самим зберігаючи час компіляції. Собою компонент resources репрезентує шаблон програмування – стратегія, тобто вибір необхідного алгоритму під час виконання програми згідно із вичерпним значенням визначаючим його. Детальніше, архітектуру цього компонента можна розглянути на Рисунку 4.8, далі.

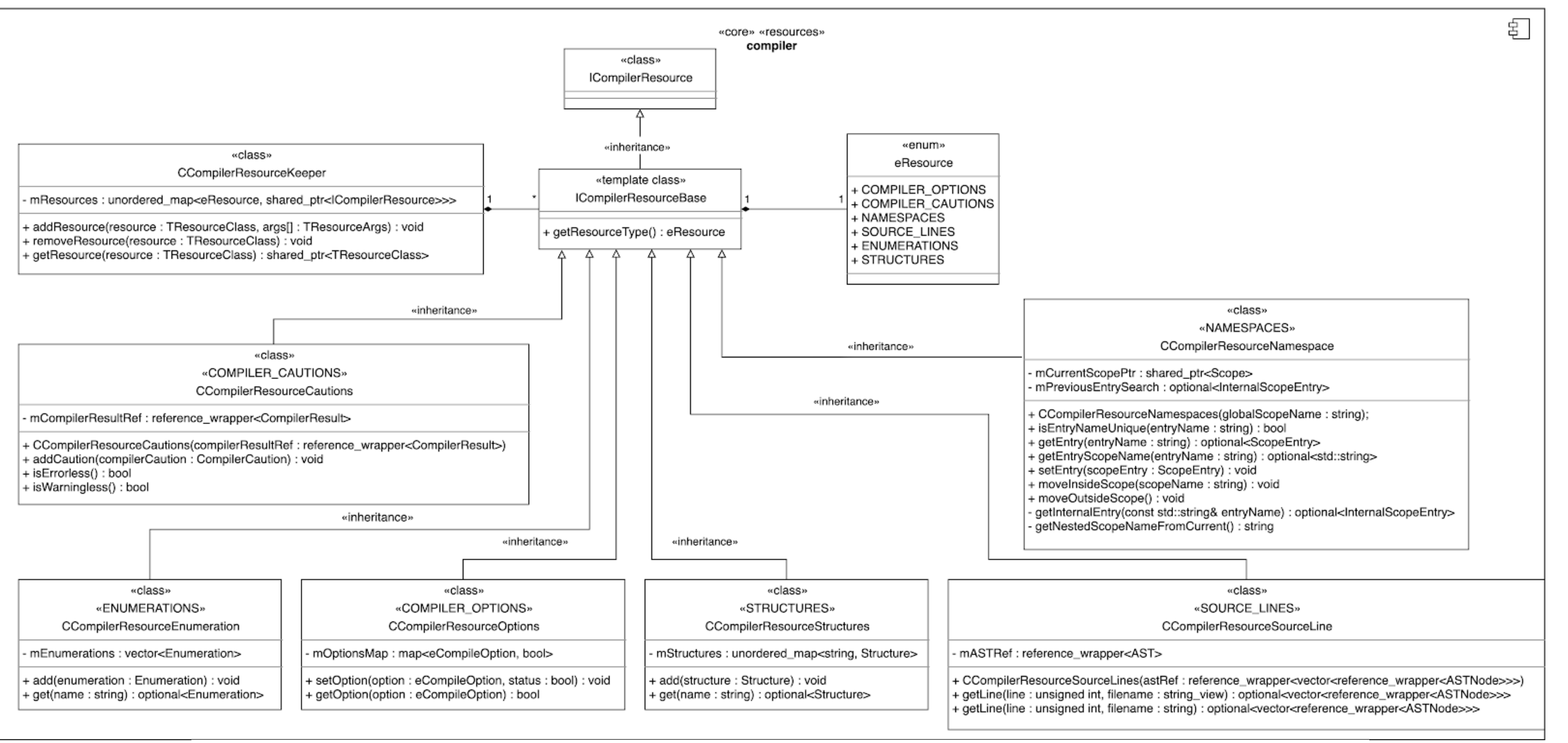

Рисунок 4.8 – Діаграма класів компоненту ядра компілятора – resources

Далі наведено короткі відомості про кожен із ресурсів компілятора доступний через описану сутність вище:

– COMPILER_CAUTIONS – це ресурс, що зберігає данні про всі помилки чи зауваження знайдені компілятором у коді;
– COMPILER_OPTIONS – це ресурс, що зберігає всі налаштування компілятора, що були виставлені користувачем;
– SOURCE_LINS – це ресурс, що дає можливість за назвою файла та номером рядка сформувати повноцінний рядок із AST;
– NAMESPACES – це ресурс, що зберігає дані про всі імена у поточному просторі імен, а також дає можливість орієнтуватись у ньому;
– ENUMERATIONS – це ресурс, що зберігає дані про всі створені у поточному просторі імен – перерахування;
– STRUCTURES – це ресурс, що зберігає дані про всі створені у поточному просторі імен – структури.

Таким чином, компонент resources спрощує інтерфейси решти компонентів ядра, а також дає можливість ним ділитися необхідними знаннями, для компіляції коду. На Рисунку 4.9, нижче, наведено приклад коду, як визначено метод класу CCompilerResourceKeeper, що дозволяє отримати ресурс із нього. Шаблонність, та вдале спадкування, дають можливість отримати ресурс за ім’ям його класу, що є найпростішим, та дає можливість уникнути незручних конструкцій. Методи, що дають можливість створювати нові ресурси у цьому класі, також є шаблонними, та мають семантику подібну функції make_shared із стандартної C++ бібліотеки.

```
1 namespace sbash
2 {
3    namespace compiler
4    {
5       class CCompilerResourceKeeper
6       {
7          public:
8             template <class TResourceClass>
9             std::shared_ptr<TResourceClass> getResource();
10      }
11
12      CCompilerResourceKeeper compilerResourceKeeper;
13
14      const auto namespacesResourcePtr = compilerResourceKeeper.get<CCompilerResourceNamespaces>();
15   };
16 };
```

Рисунок 4.9 – Приклад отримання ресурсу із класу CCompilerResourceKeeper

4.3.3 Компонент ядра компілятора: rules

Компонент rules (від англ. «правила»), відповідає за валідацію частин дерева AST. Слід зазначити, що правила мають певні вимоги до формату представлення даних (структури AST). Самі по собі класи правил мають спільний інтерфейс успадкований від базового класу правил – CCompilerRuleBase. Цей інтерфейс здебільш має структуру ускладненого функціонального об’єкту, це виражено через три основні методи валідації у цьому інтерфейсі, а саме:

- preprocessing – виконує підготовку даних для валідації;
- validate – виконує основний процес валідації даних;
- postprocessing – виконує деякі завершаючи процеси над даними після валідації.

Всі ці методи є віртуальними, але із імплементацією за замовчуванням у базовому класі, тим самим, конкретне клас-правило може імплементувати тільки необхідні йому методи, для конкретних задач. Слід зазначити, що всі ці методи повертають логічне значення, що дає можливість визначити успіх виконуємого етапу процесу валідації.

На Рисунку 4.10, нижче, можна побачити приклад, один із багатьох класів із компоненту rules. У конкретно цьому прикладі імплементовані тільки метод preprocessing та validate.

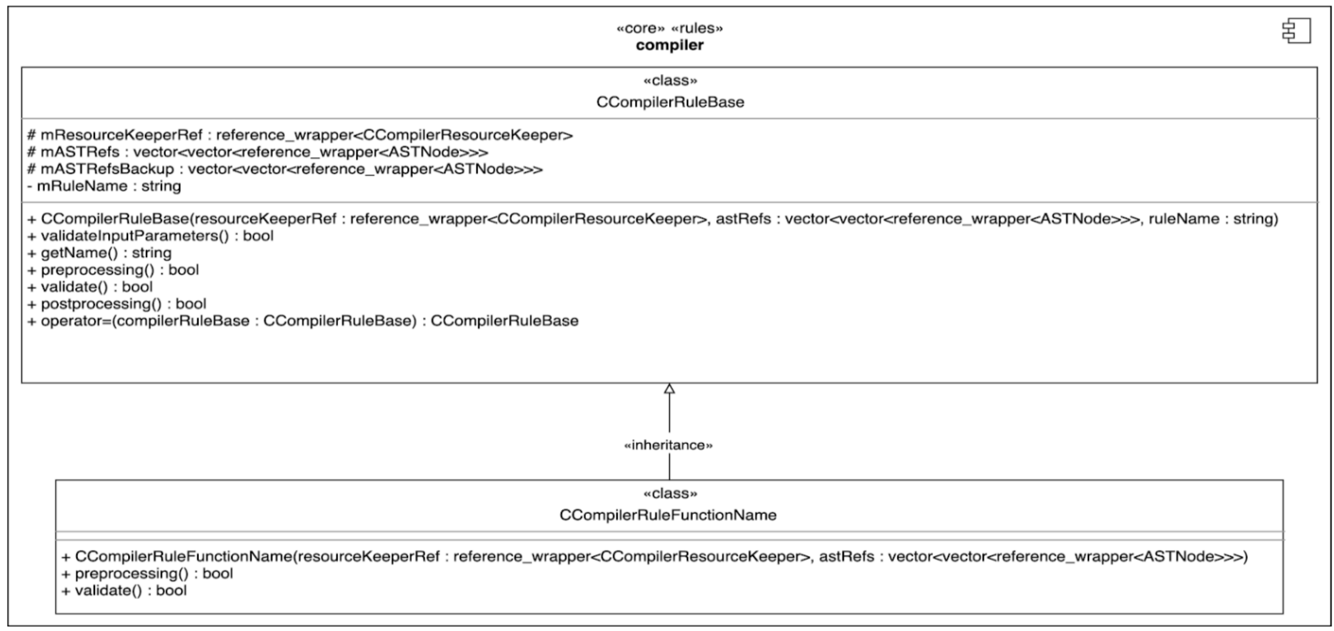

Рисунок 4.10 – Діаграма класів компоненту ядра компілятора – rules

У даному розділі не буде розглянуто діаграму класів із всіма класами компоненту rules, через те що всі вони мають схожу структуру, що була вже зазначена вище, на рисунку із діаграмою.

Другою за важливістю речей у компоненті rules є допоміжний метод validate, що уособлює у собі всю бізнес-логіку, для використання колекції правил для вхідних даних. Таке рішення спрощує використання правил у іншому компоненті, а також дозволяє застосувати шаблон проектування – фабрика, що є групою фабричних функцій, що створюють колекції правил для конкретного випадка валідації. Приклад використання такого рішення можна побачити на Рисунку 4.11, нижче.

```
1 namespace sbash
2 {
3    namespace compiler
4    {
5       template <class ... TCompilerRule>
6       bool validate(tResourceKeeperRef resourceKeeperRef, tASTRefs astRefs);
7
8       bool validateBashFunctionDeclaration(tResourceKeeperRef resourceKeeperRef, tASTRefs astRefs)
9       {
10         return validate<CCompilerRuleFunctionName,
11                         CCompilerRuleFunctionBody,
12                         // All required rules...
13                         CCompilerRuleVariablesType>(resourceKeeperRef, astRefs);
14      }
15   };
16 };
```

Рисунок 4.11 – Приклад використання фабрики для валідації AST

Таким чином, далі розглянему декілька основних груп правил (фабричних функцій) створених через фабрику, не витрачаючи час на конкретні імпліментації класів цих правил:

- validateVariableDeclaration – відповідає за валідацію AST структури із оголошенням змінної;
- validateFunctionDeclaration – відповідіє за валідацію AST структури із оголошенням функції;
- validateBashFunctionDeclaration – відповідає за валідацію AST структури із оголошенням функції із вставкою BASH коду, але за виключенням коду тіла функції, що повинен бути провалідован окремо;

- validateEnumerationDeclaration – відповідає за валідацію AST структури із оголошенням перерахування, але за виключенням коду тіла функції, що не потребує валідації;
- validateStructureDeclaration – відповідає за валідацію AST структури із оголошенням структури та її ланцюжку спадкування.

Також, важливою особливістю цих функцій є наявність контексту, котрий вони не можуть змінювати відповідно до вхідних даних (за дизайном, де контекст змінений компонентом drivers), а результат валідації залежить від того ж контексту.

4.3.4 Компонент ядра компілятора: builders

Компонент ядра builders є найпростішим із всіх компонентів. Він має схожу структуру на компонент rules у сенсі спорідненості його класів до функціонального об’єкту, бо основним його інтерфейсом для обробки даних є єдина функція build.

Цей компонент можна було б створити виключно із чистих функцій, але тоді було б складно маніпулювати доволі складними даними, та залежністю на компонент resources, через дублювання коду, що створює складність модифікації та підвищує потенційні помилки. Тому, було прийнято рішення створити структуру класів, як зазначено на Рисунку 4.12, нижче:

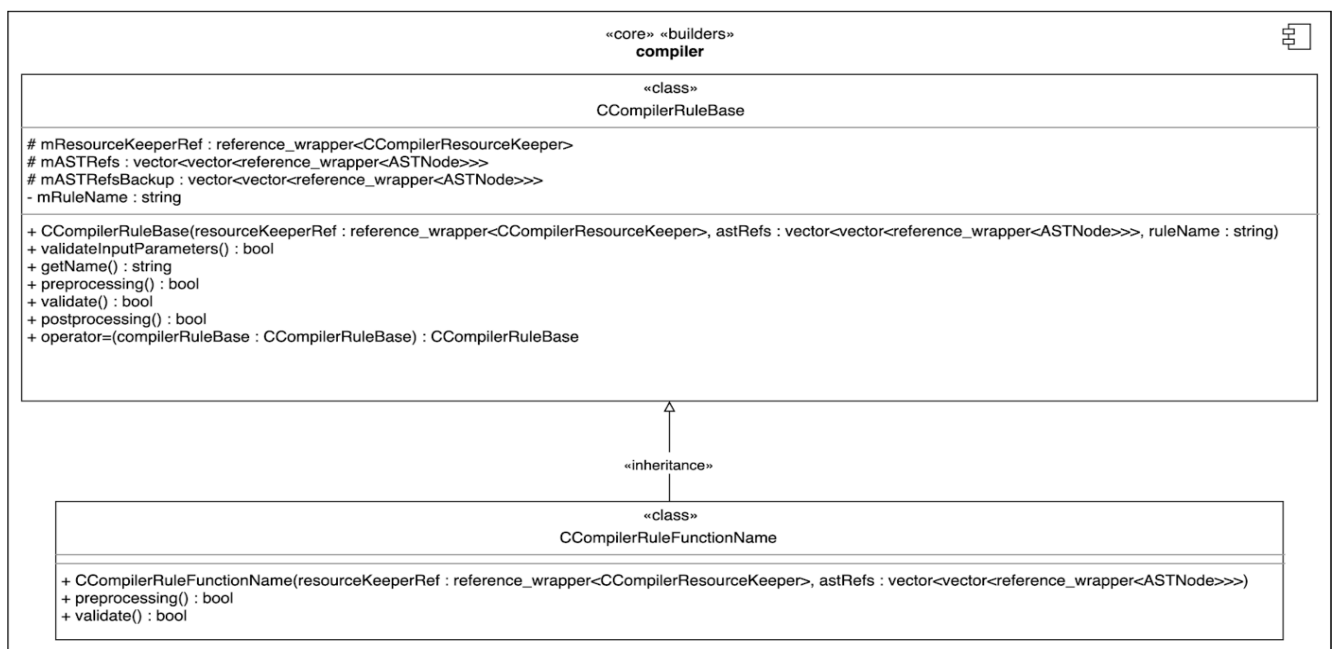

Рисунок 4.12 – Діаграма класів компоненту ядра компілятора – builders

Тобто відповідно до діаграми класів, весь загальний функціонал із обробки вхідних даних, та створення точки доступу до компоненту resources, було винесено до базового класу CCompilerBuilderBase.

Розглянемо залежність класів компоненту builders та drivers (див. Рисунок 4.13), що дає більш прозоре зрозуміння (бо це не було зазначено на глобальних діаграмах класів), які із класів створюють відповідні APS структури із вхідних AST, під час компіляції.

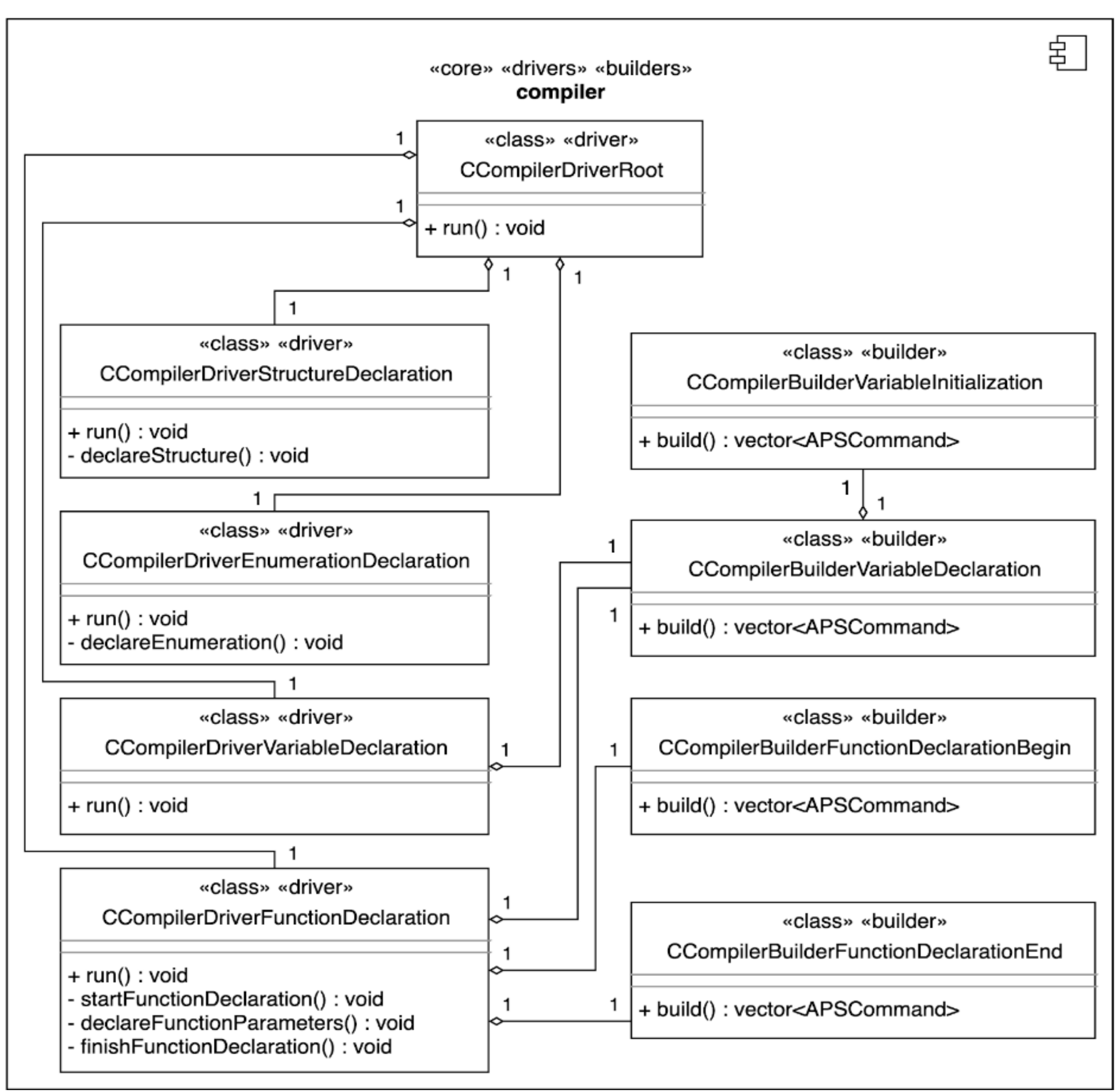

Рисунок 4.13 – Діаграма класів із відносинами класів компонентів ядра компілятора – drivers та builders

Згідно із діаграмою класів, можна зазначити, що всі класи компоненту builders, не є необхідними для існування класів компоненту drivers, але є необхідними для процесу компіляції.

4.4 Процес компіляції

Тепер, як зазначена архітектурні особливості, компонентів, та деталі реалізації, потрібно розглянути основний бізнес-процес компіляції коду, а саме компіляцію. На представленій на Рисунку 4.14, нижче – діаграмі послідовностей, зображено процес компіляції коду. Слід зазначити, що представлений процес є успішним, причиною неуспішного завершення компіляції, може бути як:

- не віно вказані вхідні параметри, тоді процес validateInputPaarmeters з компоненту main – буде останнім у ланцюжку виконання;
- незрозумілі символи для компоненти parser, або невірно вказані кодові конструкції, тоді процес tokenizeSourceFiles – буде останнім у ланцюжку виконання, або buildAST відповідно;
- помилки під час компіляції коду, тоді процес validateAST з компонента compiler буде останнім у ланцюжку виконання.

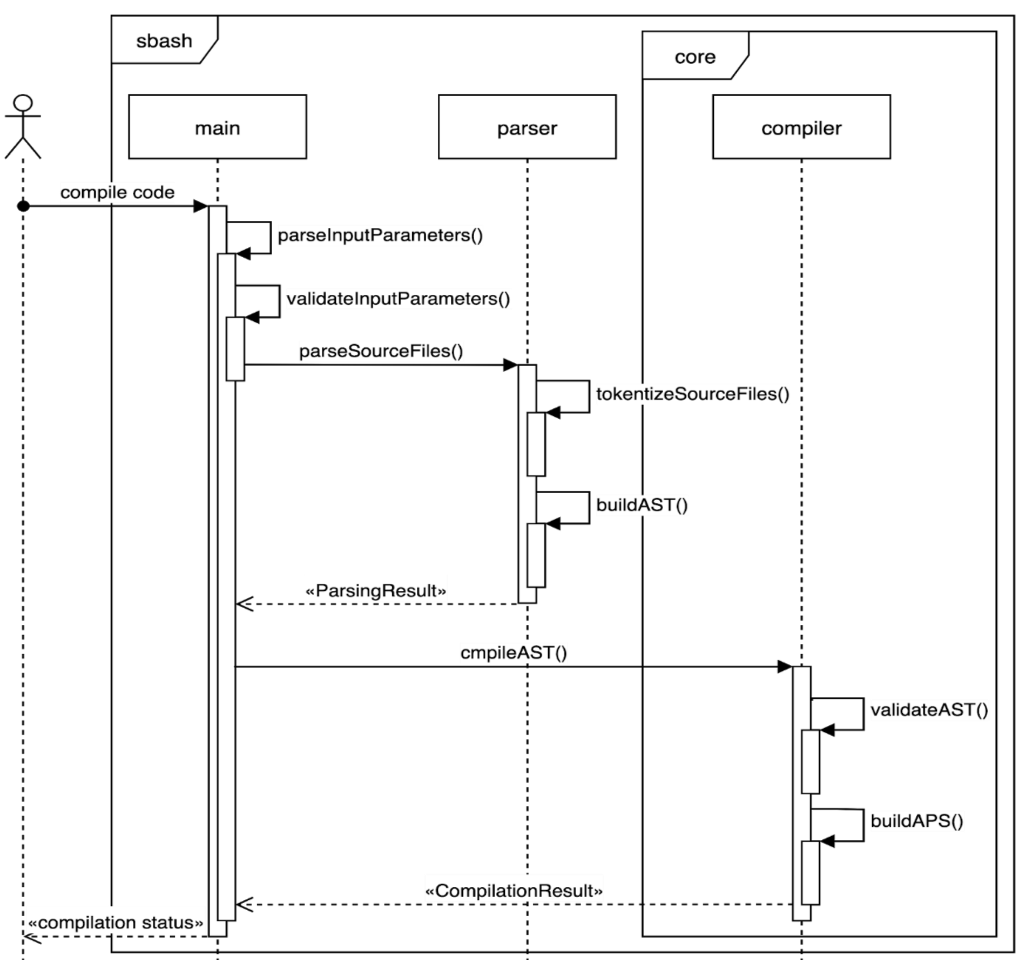

Рисунок 4.14 – Діаграма послідовностей процесу компіляції

Також слід зазначити, що назви процесів на діаграмі є уособленням суті виконаних операцій, а не фактичні назви функцій у інтерфейсу. Це зроблено заради спрощення діаграми, та репрезентації суті процесу. Окрім цього, компонент main представлений на діаграмі є уособлюванням головного процесу програми, що виконує компіляцію коду, але не має відображення на діаграмі компонентів.

4.5 Загальний огляд тестування

Розглянемо тестування створеного компілятора. Існує багато видів тестування, але згідно із специфікою проекту, та відсутністю таких компонентів як GUI (від англ. «Graphical User Interface» - «графічний інтерфейс користувача»), є основні два напрямки тестування, а саме: модульне тестування, а також інтеграційне тестування. Модульне тестування дозволяє захистити код від логічних помилок, та тестувати окремі компоненти, але не великі частини коду, у той час, як інтеграційне дозволяє тестувати всю програму в цілому. Згідно із POC, зазначеному у попередніх розділах, фокус розробки був поставлений на модульне тестування.

```
1 #define BOOST_TEST_MODULE UT_PARSER_VARIABLE
2 #include <boost/test/unit_test.hpp>
3
4 BOOST_AUTO_TEST_CASE(UT_EXAMPLE_1)
5 {
6     const bool isValid{true};
7
8     BOOST_TEST( isValid == true );
9     BOOST_TEST_INFO("SUCCESS");
10 }
11
12 BOOST_AUTO_TEST_CASE(UT_EXAMPLE_2)
13 {
14     BOOST_FAIL("FAILED");
15 }
```

Рисунок 4.15 – Приклад використання бібліотеки BOOST TEST у модульному тестуванні компілятора SBASH

Як було зазначено у попередніх розділах, для розробки компілятора потрібно було використовувати комплекс бібліотек BOOST. Також було вказано, що останній має бібліотеку для тестування BOOST TEST, саме вона і була використана для створення Unit Tests (від англ. «модульне тестування»), тобто модульних тестів.

Для прискорення розроби модульних тестів, був обраний макрос «BOOST_AUTO_TEST_CASE», що дає можливість створювати виключно функції для тестування, не витрачаючи час на створення main-функції, та інше (див. Рисунок 4.15).

5 ЕКСПЛУАТАЦІЯ ОБЛАСТЬ ЗАСТОСУВАННЯ ТА ПОДАЛЬШИЙ РОЗВИТОК

5.1 Використання компілятора SBASH

Згідно із попередніми розділами, було визначено, що компілятор не має графічного інтерфейсу, що може бути складно для використання пересічному користувачу. Але, якщо зазначити, що розробка даного проекту направлена на користувачів UNIX-подібних систем, то вони знають як працювати із терміналом. Тим не меньш, далі описано як використовувати компілятор SBASH у консолі терміналу, та основні сценарії його використання із прикладами.

5.1.1 Сценарії використання компілятора

Перше що треба зазначити, що пояснення щодо використання компілятора у різних сценаріях виконання проводиться за тих умов, що він вже скомпільований у даній операційній системі, та занесений до глобальних шляхів, щоб мати можливість використовувати його у терміналі без зазначення повного шляху до його виконуючого файлу.

5.1.1.1 Версія компілятора

Одною із популярних та найпоширеніших функцій консольних програм у середовищі UNIX-подібних систем є виведення версії програми. Це, не аби як важливо – для компілятора, бо ті можливості, та вже відомі помилки (можливі) відслідковуються за версією компілятора. Демонстрація версії компілятора виводиться переданням йому для виконання аргументів – «-v» або «--version» (див. Рисунок 5.1).

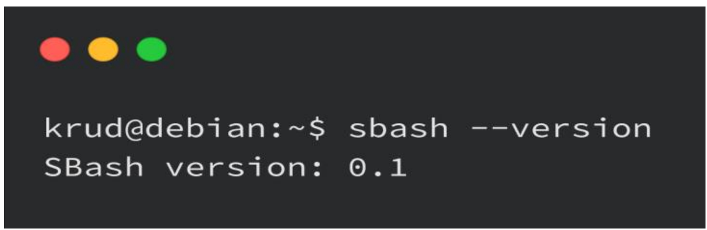

Рисунок 5.1 – Приклад виведення версії компілятора SBASH

5.1.1.2 Довідка компілятора

Як і більшість консольних програм, написаних для UNIX-подібних систем, – компілятор SBASH має вбудовану довідку, що можна отримати прямо із терміналу.

Для того, щоб її передевитись, використовують відповідний для цього аргумент, що використовується у всіх подібних випадках – «-h» або «--help», приклад використання цієї команди наведено на Рисунку 5.2, нижче.

```
krud@debian:~$ sbash --help
SBash [parameters...]

 -o              Set output file name.
 -d              Set path to output directory.
    --debug      Run program in debug mode.

 -v --version    Show program version.
 -h --help       This menu.
```

Рисунок 5.2 – Приклад виведення довідки компілятора SBASH

5.1.1.3 Компіляція

Саме головне призначення компілятора і є компіляція коду, це є функцією за замовчуванням при виконанні компілятора SBASH без наявності інших аргументів виконання. Тим не менш, компілятор потребує знати із яким ім’ям створити вихідний файл скрипту, а також звісно потребує специфікувати вхідні файли. Всі файли (шляхи до файлів), що специфікуються – компілятор сприймає як вхідний файл, у разі, якщо перед назвою одного із файлів, вказати аргумент «-o» (від англ. «output» - «вихідний»), то наступне ім’я файлу буде розпізнане як ім’я вихідного файлу (котрий не повинен існувати у файловій системі), приклад наведено на Рисунку 5.3, нижче.

```
krud@debian:~$ sbash -o script.sh script.sb
```

Рисунок 5.3 – Приклад компіляції коду компілятором SBASH

5.1.1.4 Налагодження роботи компілятора

Останнім, але не менш важливим є аргумент, що використовується для налагодження роботи компілятора у скомпільованому компіляторі. А саме він, просто виводить додаткову інформацію, що може допомогти знайти, або локалізувати помилку. Таким аргументом є «--debug» (від англ. «налагодження»).

5.1.2 Приклад скомпільованого коду

Фінальною демонстрацією роботи компілятора мови програмування SBASH, є приведення коду ним скомпільованому. Даний код не має виробничого значення, але гарний для демонстрації, щоб зрозуміти, як працює весь концепт цього проекту.

На Рисунку 5.4, нижче, можна побачити приклад вхідного коду написаного мовою програмування SBASH, та скомпільованого у мову програмування BASH (див. Рисунок 5.5).

```
 1 struct Base
 2 {
 3     double d
 4 }
 5
 6 struct Data : Base
 7 {
 8     string s
 9 }
10
11 func main() : int
12 {
13     Data d1, d2
14
15     d1.d = 1.0
16     d1.s = “string data”
17
18     d2 = d1
19
20     return 0
21 }
```

Рисунок 5.4 – Приклад програми написаної мовою програмування SBASH

```
 1 function main()
 2 {
 3     local d1_d
 4     local d1_s
 5
 6     d1_d="1.0"
 7     d1_s="string data"
 8
 9     local d2_d
10     local d2_s
11
12     d2_d="$d1_d"
13     d2_s="$d1_s"
14
15     return 0
16 }
17
18 main
19 exit $?
```

Рисунок 5.5 – Приклад програми скомпільованої компілятором SBASH

Таким чином, можна наочно побачити, як мова програмування SBASH транслюється її компілятором із складних конструкцій комплексних структур до більш простих команд, що будуть працювати на будь-якому інтерпретатору BASH.

5.2 Поточні ліміти компілятора

Згідно із твердженнями із попередніх розділів, компілятор мови програмування SBASH розроблюється у рамках імплементації POC. Що означає наявність такої імплементації, що доводить працездатність проекту, але не зумовлює імплементацію всіх, визначених на етапі проектування, можливостей. Детальніше це вже було описано вище, далі буде наведені деякі обмеження, що виникли через описані причини, а також не були зазначені у відповідних до них пунктах та підпунктах, або є важливими для згадки тут. Також вказані додаткові причини відсутності імплементації тих або інших можливостей компілятора.

5.2.1 Обмеження зворотнього значення функції

У поточній імплементації неможливо вказати зворотнім значенням функції – колекцію. Перша причина цьому є відсутність можливості додавати модифікатори типів в цілому для зворотного значення функції. Тобто як біло вже зазначено – неможливо повернути константну змінну або змінну, що є посиланням. У той час, як зворотнім значенням – можна буде зробити масив даних, інші модифікатори типів додані не будуть через те, що це ускладнить поведінку коду, що не буде відповідати ідеології мови програмування SBASH, на поточний час. Іншими словами, зворотне значення функції буде повертатись завжди за значенням та буде виконуватись його копіювання.

5.2.2 Обмеження на використання виразу

Вираз, як було вказано у відповідному пункті – це деяка конструкція, із операторів, літералів, та змінних. Відповідно до даної архітектури компілятора – обробкою такої структури даних повинен займатися компонент drivers, але через його (виразу) комплексність, не має сенсу намагатись імплементувати таке рішення. Тому використання виразу обмежується тільки простими конструкціями де є один чи два аргументи (відповідно до використаного оператора), а також один оператор, для операції призначення змінній нового значення, а у таких випадках як:

- ініціалізації змінних;
- оператори умовного переходу: if, elif, else, switch, ітд.;
- виклик функції.

Та у інших подібних конструкціях використовується вираз що має тільки одну змінну, та не має операторів. Імплементація цього буде виконана у подальшій розробці.

5.2.3 Стандартна бібліотеки

Відповідно до архітектури мови програмування SBASH, є можливість використовувати функції із вставками BASH коду. Тому імплементація стандартної бібліотеки функцій, що буде розширювати функціонал компілятору – доволі не складний у реалізації, але згідно до POC – не був реалізований. Також у рамках цієї задачі, можна реалізувати автоматичний пошук файлів для включення стандартної

бібліотеки та інших (наприклад відповідно до деякого конфігураційного файлу, або до деякої структури папок).

5.2.4 Формат виведення повідомлень компілятора із кодом SBASH

В деяких ситуаціях, таких як: лексичний, граматичний або семантичний аналіз, а також інших, є необхідність виведення коду SBASH у термінал. Цей функціонал реалізується через зворотній аналіз структури AST, за деякими параметрами, наприклад такими як:

- ім’я файлу із котрого були отримані структури AST Node;
- номер рядка із відповідними структурами;
- необхідні для виділення токенами.

Поточна реалізація структури AST Node не дозволяє визначити які із токенів повинні розділятись пробілами, а також відсутність можливості ідентифікації необхідних для виділення токенів (на поточну реалізацією, це робиться за передбаченням їх типом, а також вже відомими значенням).

Така реалізація призводить до не вдало формованого виводу помилки, попередження, тощо (хоч його і достатньо для розуміння проблеми виникнення повідомлення від компілятора). А також не дає можливість максимально точно сепарувати виділення токенів у рядку.

Вирішенням описаних проблем може бути присвоєння унікального ідентифікатора кожному токену, а також додавання деякої змінної, що буде визначати наявність пробілів довкола токенів.

5.2.5 Попереднє визначення структур даних, функцій та декларація змінних

У поточній реалізації компілятора, пошук та обробка визначення структур даних, функцій, та декларація змінних виконується лінійно із AST. Це доволі простий для виконання алгоритм, але він не оброблює всі можливі ситуація, а саме циклічне визначення.

Прикладом такого питання є декларація структури «A» та структури «B» із атрибутами протилежних структур. Таким чином, неможливо, скомпілювати жодну із структур даних, бо жодна з них не має повноти типів доступних для декларації.

Рішенням цього може бути як введення додаткової семантичної конструкції, що буде давати компілятору назву та тип структури даних, що буде задекларована, але не декларує її атрибутів. Або більш елегантний варіант, що спростить використання мови програмування SBASH, та не буде вимагати додаткових конструкцій, а саме можливість компілятору по-перше визначити назви та ціпки спадкування існуючих структур даних, до того як буде обробляти їх атрибути.

5.3 Подальший розвиток, та його методологія

У межах цієї роботи було розроблено повний стандарт мови програмування SBASH, а також вироблена максимально поста та ефективна архітектура її компілятора. Тим не менше, є багато можливих дрібних змін, що покращать роботу як із мовою програмування SBASH із точки зору стандарту, так і у компіляторі, з точки зору зручності його використання.

5.3.1 Повідомлення компілятора із попередженнями

Одним із таких подальших змін є поліпшення компілятора з точки зору відображення підказок про код, що компілюється. Також такі повідомлення називають попередженнями компілятора, бо вони дають знати користувачу, коли він потенційно не вірно, або не безпечно використовується код.

У поточній імплементації компілятора вже створена інфраструктура для виводу таких повідомлень з точки зору помилок компіляції. Її використання дозволить доволі просто імплементувати цю додаткову можливість.

5.3.2 Відокремлення SBASH SDK

У поточній організації проекту, всі бібліотеки в ньому використані – зберігаються у межах одного репозиторію GIT. Однією із таких бібліотек є SDK, що вже була описана у попередніх розділах. Ця бібліотека зберігає всі необхідні дані для репрезентації програми написаної мовою програмування SBASH у структурах даних AST а також у APS.

З точки зору подальшої розробки, буде доволі зручно додати туди методи, що дозволять робити лексичний та граматичний аналізи. Що можуть бути використані наприклад у сторонніх проектах для побудови AST та його використання. Також це дозволить модифікувати процес лексичного та граматичного аналізу без зміни

омновного компілятора. Іншими словами, компоненти ядра компілятора будуть відокремлені фізично від компонентів, що служать для попередньої обробки вхідних даних.

5.3.3 Функції із змінною кількістю аргументів

Однією із потужних можливостей такої мови програмування як C++ у новітніх стандартах є шаблони. Ця технологія дозволяє писати багато узагальнених алгоритмів, та структур даних. Але слід пам'ятати, що по-перше вона доволі не проста у реалізації компілятора, а по-друге на даний час не має відображення у ідеології мови програмування SBASH.

Тим не менш, один із її аспектів, а саме шаблоні функції із зміною кількістю аргументів можуть бути використані і у мови програмування SBASH, через свою універсальність, та необхідність у скриптах, та просто незамінність, наприклад у таких функціях як, наприклад: print – що виводить текст до консолі.

Тому у подальшому розвитку, доволі вірогідно застосування цієї технології у мові програмування SBASH, що значно спростить написання функцій.

5.3.4 Перевантаження функцій

Більшість сучасних мов програмування підтримують таку можливість як перезавантаження функцій. Основна ідея цієї особливості компілятора полягає у тому, що можливо визначити декілька функцій із однаковим ім’ям, але із різним набором параметрів (з точки зору типів цих параметрів та їх кількості, але не порядку їх визначення).

Така функція відсутня у поточному дизайні мови програмування SBASH, що призводить до складності іменування функцій. Ця функція реалізовується зі сторони компілятора, тому може бути додана у нього під час подальшої розробки компілятора.

ВИСНОВКИ ТА РЕКОМЕНДАЦІЇ

У магістерській роботі проведені дослідження проблеми розробки на мові програмування BASH у embedded системах, а також розроблено нову мову програмування SBASH із однойменним компілятором до неї.

Розроблена мова програмування SBASH, вирішує наступні проблеми, що існували у мові програмування BASH:

- вирішує неузгодженість синтаксису у мові програмування BASH;
- вирішує проблему bashism-у;
- вирішує невизначеність параметрів функцій;
- вирішує проблему повторного оголошення змінних та функцій.

А також мова програмування SBASH додає нові можливості у розробку BASH скриптів, такі як:

- новий тип даних – композитна структура;
- новий тип даниз – перерахування;
- модифікатори типів;
- росширений перелік built-in типів;
- строга типізація;
- дозволяє компілювати скрипт.

Згідно із проведеними дослідженнями, можна стверджувати, що мова програмування BASH займає важливе місце у галузі розробки програмного забезпечення, бо посягає третє місце по популярності серед всіх скриптових мов програмування (див. Рисунок 1.7). Тим більш важливою вона є у embedded розробці через відомі особливості даній галузі. Такі особливості цієї мови програмування, як динамічна типізація, відсутність семантичної валідації, та проблема bashim-у, ускладнюють роботу з нею.

Рекомендованною до застосування галуззю є не тільки embedded системи, а і будь-яка система під котру розроблються скрипти мовою програмування BASH. Бо скрипт написаний мовою програмування SBASH має більш сувору валідацію, та позбавлений проблеми bashsim-у, через що є більш універсальним та надійним (у сенсі – стійкості до помилок), а ніж пересічний сприпт написаний мовою програмування BASH.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1 Programming embedded systems: with C and GNU development tools. (2006) [Електронний ресурс] / Michael Barr; Anthony J. Massa. // С. 1–2. – Режим доступу: https://books.google.com/books?id=nPZaPJrw_L0C&pg=PA1

2 IEEE Transactions on Nuclear Science. (1996) [Електронний ресурс] / H. Kleines, K. Zwoll // С. 1. – Режим доступу: http://temporeal.lesc.ufc.br/downloads/Artigos/Real%20time%20UNIX%20in%20embedded%20control-a%20case%20study%20within%20the%20context%20of%20LynxOS.pdf

3 Wiktionary: Bashism [Електронний ресурс] / Режим доступу: https://en.wiktionary.org/wiki/bashism

4 ShellCheck [Електронний ресурс] / Режим доступу: https://github.com/koalaman/shellcheck

5 Most Popular Technologies [Електронний ресурс] / Режим доступу: https://insights.stackoverflow.com/survey/2019#most-popular-technologies

6 FOLDOC – Free Online dictionary of computing – builtin [Електронний ресурс] / Режим доступу: http://foldoc.org/builtin

7 FOLDOC – Free Online dictionary of computing – primitive [Електронний ресурс] / Режим доступу: http://foldoc.org/primitive

8 Advanced compiler design and implementation. (1997) [Електронний ресурс] / S. Muchnick // С. 1. – Режим доступу: https://books.google.com.ua/books?id=Pq7pHwG1_OkC&lpg=PR31&ots=4ZcWGrh4pP&dq=compiler&lr&pg=PP1#v=onepage&q=compiler&f=false

9 Boost Background Information [Електронний ресурс] / Режим доступу: https://www.boost.org/users/index.html

10 Lex and yacc. (1997) [Електронний ресурс] / John R. Levine, John Mason, John R Levine, B.A., Ph.D., Tony Mason, Doug Brown, John R.. Levine, Paul Levine // С. xi. – Режим доступу: https://books.google.com.ua/books?id=fMPxfWfe67EC&lpg=PP2&ots=RdSSik-KAU&dq=LEX%20and%20YACC&lr&pg=PR11#v=onepage&q=LEX%20and%20YACC&f=false

11 Translation to and from Polish Notation. (1962) [Електронний ресурс] / C. L. Hamblin // С. 210–213. – Режим доступу: https://academic.oup.com/comjnl/article/5/3/210/424386

12 Learning UML 2.0. (2006) [Електронний ресурс] / Russ Miles, Kim Hamilton // Режим доступу: https://books.google.com.ua/books?id=BUKbAgAAQBAJ&lpg=PP1&hl=uk&pg=PP1#v=onepage&q&f=false

13 The Linux Document Project: Advanced Bash-Scripting Guide [Електронний ресурс] // Режим доступу: https://www.tldp.org/LDP/abs/html/index.html

Додаток А
ПЕРЕЛІК КОПІЙ ДЕМОНСТРАЦІЙНОГО МАТЕРІАЛУ

Випускна кваліфікаційна робота
з теми:
«Розробка компілятора строго-типізованої мови програмування для командної оболонки BASH у середовищі UNIX-подібних систем»

Рисунок 1 – Назва проекту

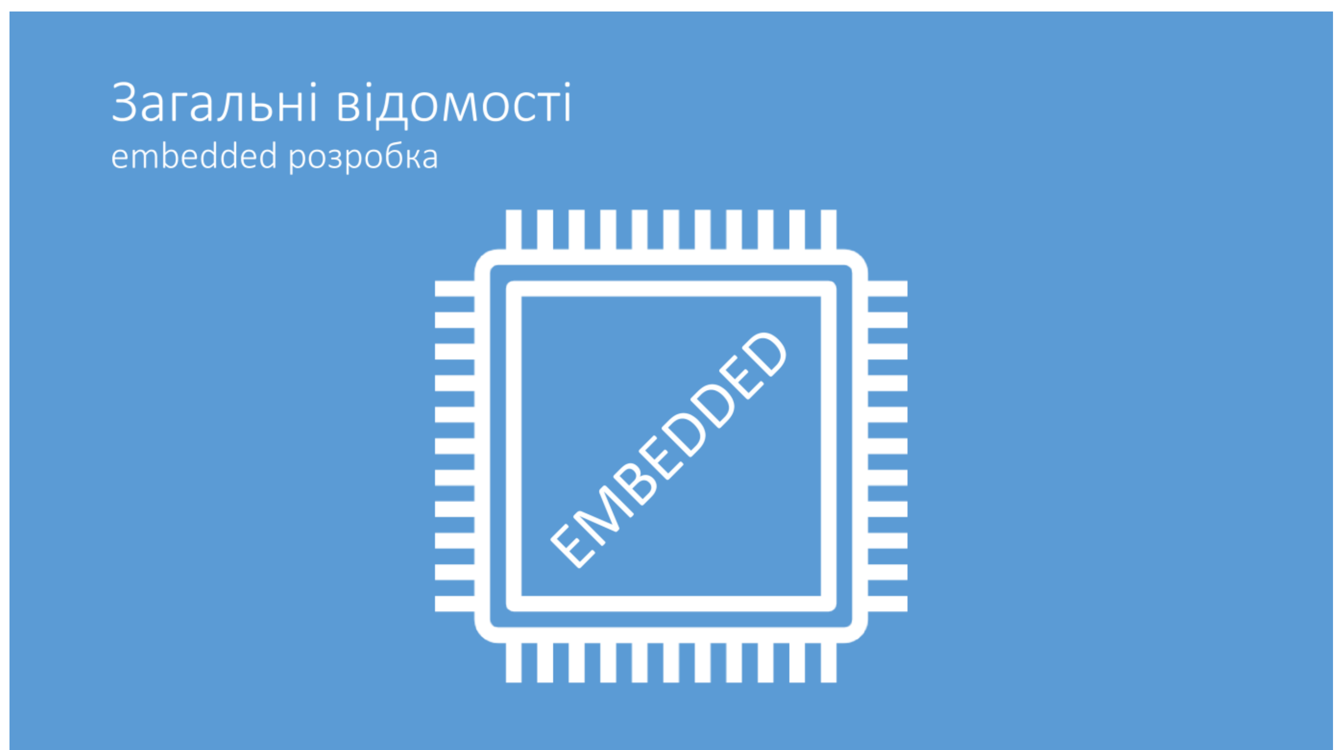

Рисунок 2 – Загільні відомості – embedded розробка, вступ

Загальні відомості
embedded розробка

Embedded розробка у середовищі UNIX-подібних систем, її особливості:

- обмеження обладнання;
- обмеження доступного програмного забезпечення;
- обмеження ресурсів системи.

Рисунок 3 – Загільні відомості – embedded розробка

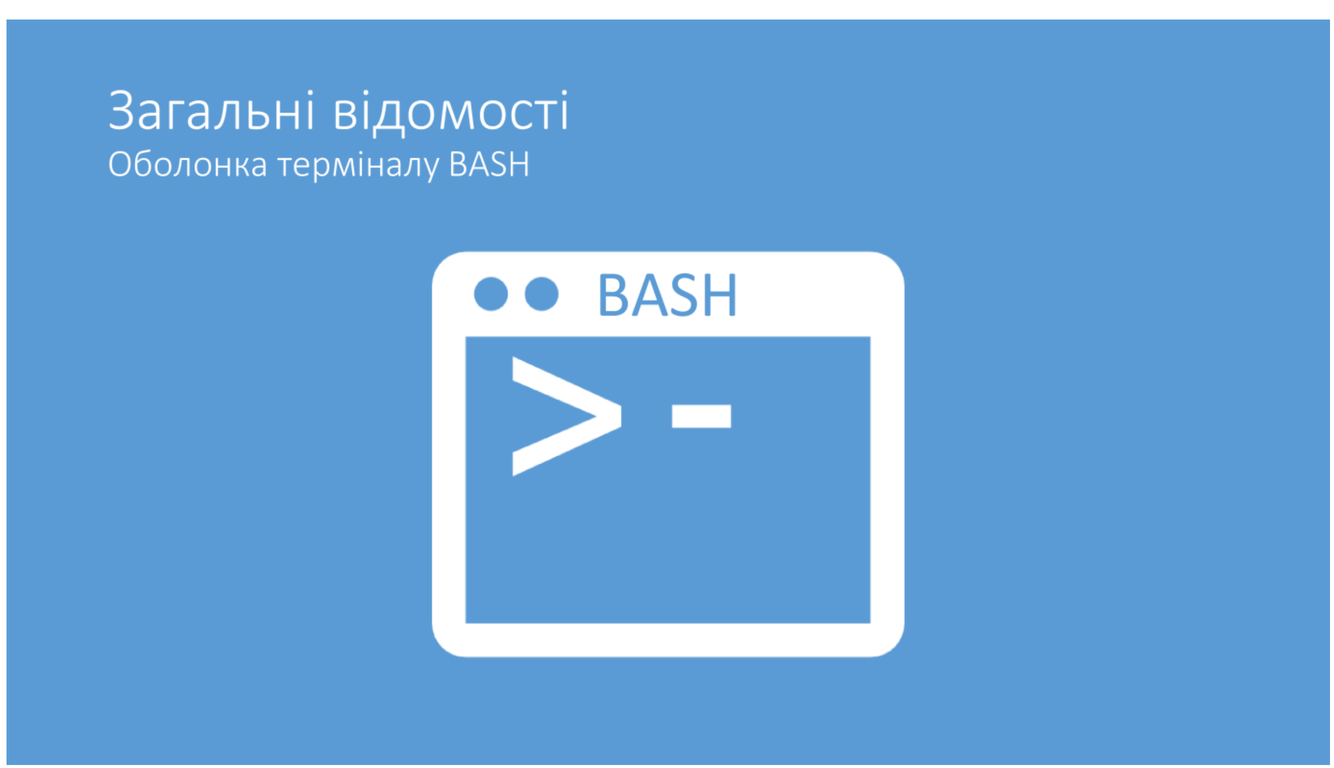

Рисунок 4 – Загільні відомості – оболонка терміналу BASH, вступ

Загальні відомості

Оболонка терміналу BASH

Проблеми розробки скриптів мовою програмування BASH, що унеможливлюють її застосування:

- проблема bashism-у (різниця між програмами різних дистрибутивів);
- неоднозначность синтаксису, та неузгодженність його конструкцій;
- відсутність валідації коду;
- відсутність попередження користувача при неоднозначних ситуаціях.

Рисунок 5 – Загільні відомості – оболонка терміналу BASH

Рисунок 6 – Мова програмування SBASH – загальний огляд, вступ

Мова програмування SBASH

Загальний огляд

Головними характеристиками мови програмування SBASH є наступні:

- строго типізована система типів;
- процедурна семантика;
- компільований тип використання;
- C-like синтаксис.

Також можна зазначити що на дизайн SBASH мали вплив такі мови програмування як C та Python, а аналогів – не існує.

Рисунок 7 – Мова програмування SBASH – загальний огляд

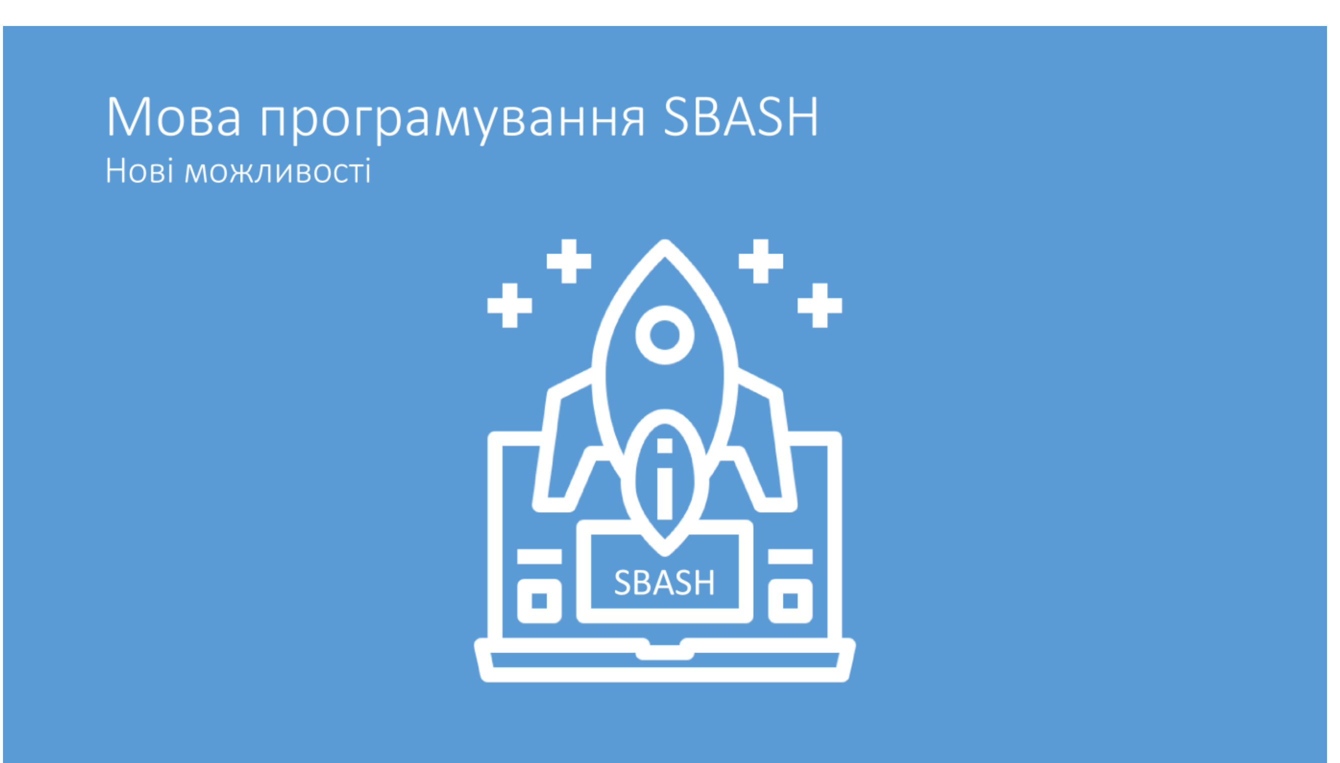

Рисунок 8 – Мова програмування SBASH – нові можливості, вступ

Мова програмування SBASH
Нові можливості

Категоризація нових можливостей має наступну структуру:

- структури даних;
  - Композитні – structure;
  - не композитні – enumeration;
- модифікатори типів даних;
  - constant;
  - reference;
- синтаксис;
  - визначені параметри функцій;
  - визначені оператори умовного переходу, та цикли.

Рисунок 9 – Мова програмування SBASH – нові можливості

Рисунок 10 – SBASH компілятор, вступ

SBASH Компілятор

Базові компоненти компілятора мови програмування SBASH, відповідальні за процес обробки коду:

- lexer;
- parser;
- compiler
  - drivers;
  - resources;
  - rules;
  - builders.

SBASH SOURCES ➤ TOKENS ➤ AST ➤ APS ➤ BASH SCRIPT

Рисунок 11 – SBASH компілятор

Рисунок 12 – Мова програмування SBASH – подальший розвиток, вступ

Рисунок 13 – Мова програмування SBASH – подальший розвиток


```
struct Base
{
   double d
}
struct Data : Base
{
   string s
}
func main() : int
{
   Data d1, d2
   d1.d = 1.0
   d1.s = “string data”
   d2 = d1
   return 0
}
```

```
function main()
{
   local d1_d
   local d1_s
   d1_d=“1.0”
   d1_s=“string data”
   local d2_d
   local d2_s
   d2_d=“$d1_d”
   d2_s=“$d1_s”
   return 0
}
main
exit $?
```

Рисунок 14 – Приклад скомпільованого коду – SBASH у BASH

Рисунок 15 – Подяка за увагу

Додаток Б
ВИТЯГИ КОДУ РЕАЛІЗАЦІЇ КОМПІЛЯТОРА SBASH

Файл: SBashMain.cpp

```
/*****************************************************************
***********
 * Project       SBash (Solid Bash)
 * (c) copyright 2020
 * Creator       Kirill Rud
 *               All rights reserved

*************************************************************************
***/

/**
 * @file:    SBashMain.cpp
 *
 * @author:  Kirill Rud
 *
 * @created: Jan 2, 2020
 *
 * @brief    This file contains main function of the program and
general logic.
 */

#include "common/api/tracelog/TraceLogDefinition.hpp"
#include "imp/helper/api/CompileMessageHelper.hpp"
#include "imp/helper/api/ToStringHelper.hpp"
#include "imp/helper/api/InputParametersHelper.hpp"
#include "imp/helper/api/UtilityHelper.hpp"
#include "imp/helper/api/FileSystemHelper.hpp"
#include "imp/parser/api/ISBashParser.hpp"
#include "imp/compiler/api/ISBashCompiler.hpp"
#include "imp/ProgramInfo.hpp"

DEFINE_SCOPE("SBashMain")

enum class eExecutionStatus
{
```

```
          OK                      = 0
        , WRONG_INPUT_PARAMETERS = 1
        , PARSING_ERROR          = 2
        , COMPILING_ERROR        = 3
      };

      int main(const int argc, const char** argv)
      {
         eExecutionStatus executionStatus{eExecutionStatus::OK};

         auto& traceManager = common::tracelog::getTraceManagerInstance();

         traceManager.setThreadAlias( std::this_thread::get_id(), "main" );

traceManager.setProcessingStrategy(common::tracelog::eProcessingStrategy::P
ROCESS_MESSAGES_IN_SEPARATE_THREAD);



         #if RELEASE_BUILD

            traceManager.setLogMessageFilter(
static_cast<int>(common::tracelog::eLogMessageType::CATEGORY_RELEASE) );
            traceManager.setLoggingType(
static_cast<int>(common::tracelog::eLogginType::PRINT_TO_CONSOLE) );

         #endif

         #if DEVELOPMENT_BUILD

            traceManager.setLogMessageFilter(
static_cast<int>(common::tracelog::eLogMessageType::CATEGORY_DEVELOPMENT)
);
            traceManager.setLoggingType(
static_cast<int>(common::tracelog::eLogginType::SAVE_TO_FILE_AND_PRINT_TO_C
ONSOLE) );
            traceManager.setOutputFile("sbash.log");

         #endif
```

```
        sbash::helper::inputparameters::ProgramOptions programOptions;

        auto inputParameters       =
sbash::helper::inputparameters::toInputParameters(argc, argv);
        auto parsedInputParameters =
sbash::helper::inputparameters::parseInputParameters(
std::move(inputParameters) );

        const auto validationResult =
sbash::helper::inputparameters::validateParsedParameters(parsedInputParamet
ers);

        if( validationResult.reason !=
sbash::helper::inputparameters::eValidationReason::OK )
        {
           LOG_ERROR( "Wrong input parameters! Reason: %s, because of:
%s", sbash::helper::inputparameters::toString(validationResult.reason),
validationResult.source );

           executionStatus = eExecutionStatus::WRONG_INPUT_PARAMETERS;
        }
        else
        {
           programOptions = std::move(
sbash::helper::inputparameters::convertToProgramOptions(
std::move(parsedInputParameters) ) );
        }



        if( executionStatus == eExecutionStatus::OK )
        {
           if( programOptions.isDebugActive == true )
           {
              traceManager.setLogMessageFilter(
static_cast<int>(common::tracelog::eLogMessageType::CATEGORY_DEVELOPMENT)
);
           }
```

```
            const auto programAction{
sbash::helper::utility::deductActionType(programOptions) };

            switch( programAction )
            {
               case
sbash::helper::utility::eActionType::COMPILE_SOURCE_FILES:
               {
                  bool isAllResourcesAvailable{true};

                  const auto& sourceFilesNames =
programOptions.sourceFiles;
                  for( const auto& inputFile : sourceFilesNames )
                  {
                     if(
sbash::helper::filesystem::isFileAvailable(inputFile) == false )
                     {
                        isAllResourcesAvailable = false;

                        LOG_ERROR("Couldn't find file: %s", inputFile)
                        LOG_INFO("Aborting compilation.")

                        break;
                     }
                  }

                  auto& outputDirectory = programOptions.outputDirectory;

                  if( outputDirectory.empty() )
                  {
                     outputDirectory = programOptions.executablePath;
                  }

                  if( isAllResourcesAvailable == true &&
sbash::helper::filesystem::isDirectoryAvailable(outputDirectory) == false )
                  {
                     isAllResourcesAvailable = false;
```

```
                    if(
sbash::helper::filesystem::isFileSystemObjectAvailable(outputDirectory) ==
false )
                    {
                       LOG_ERROR("Couldn't use output directory, it
doesn't exist: %s", outputDirectory)
                    }
                    else
                    {
                       LOG_ERROR("Couldn't use output directory,
permission denied: %s", outputDirectory)
                    }

                    LOG_INFO("Aborting compilation.")
                 }

                 if( isAllResourcesAvailable )
                 {
                    auto parserPtr = sbash::parser::makeParser();

                    DBG_INFO("Reading files...")

                    std::vector<std::string> fileLines;
                    for( const auto pathToSourceFile : sourceFilesNames )
                    {
                       DBG_INFO("Reading file: %s", pathToSourceFile)

                       fileLines = std::move(
sbash::helper::filesystem::readFileLines(pathToSourceFile) );

                       parserPtr->addSourceFile(
sbash::parser::SourceFile{ std::move(pathToSourceFile),
std::move(fileLines) } );
                    }

                    DBG_INFO("Parsing code...")

                    auto parsingResult = parserPtr->parse();

                    parserPtr->clear();
```

```
                    if( parsingResult.isSuccess )
                    {
                       DBG_INFO("Code is successfully parsed.")

                       auto compilerPtr = sbash::compiler::makeCompiler();

                       compilerPtr->setAST( std::move(
parsingResult.ast.value() ) );

                       auto compilingResult = compilerPtr->compile();

                       compilerPtr->clear();

                       if( compilingResult.isSuccess )
                       {
                          if( compilingResult.isWarning )
                          {
                             DBG_INFO("Code is successfully compiled
(warnings exist).")

                             executionStatus =
eExecutionStatus::COMPILING_ERROR;

                             const auto& compilingWarningCollection =
compilingResult.cautions;

                             for( const auto& compilingCaution :
compilingWarningCollection )
                             {

sbash::helper::compilemessage::printCaution(compilingCaution);
                             }
                          }
                          else
                          {
                             DBG_INFO("Code is successfully compiled.")
                          }
                       }
                       else
                       {
                          DBG_ERROR("Compiling failed!")
```

```
                            executionStatus =
eExecutionStatus::COMPILING_ERROR;

                            const auto& compilingErrorCollection =
compilingResult.cautions;

                            for( const auto& compilingCaution :
compilingErrorCollection )
                            {

sbash::helper::compilemessage::printCaution(compilingCaution);
                            }
                        }
                    }
                    else
                    {
                        DBG_ERROR("Code parsing failed!")

                        executionStatus = eExecutionStatus::PARSING_ERROR;

                        const auto& parsingErrorCollection =
parsingResult.cautions;

                        for( const auto& parsingCaution :
parsingErrorCollection )
                        {

sbash::helper::compilemessage::printCaution(parsingCaution);

                            DBG_INFO("")
                        }
                    }
                }

                break;
            }
            case
sbash::helper::utility::eActionType::SHOW_PROGRAM_VERSION:
            {
```

```
                LOG_INFO("%s version: %s", sbash::programinfo::name,
sbash::programinfo::version)

                break;
            }
            case sbash::helper::utility::eActionType::SHOW_PROGRAM_HELP:
            {
                LOG_INFO("%s [parameters...]", sbash::programinfo::name)
                LOG_INFO("")
                LOG_INFO(" -o             Set output file name.")
                LOG_INFO(" -d             Set path to output directory.")
                LOG_INFO("    --debug     Run program in debug mode.")
                LOG_INFO("")
                LOG_INFO(" -v --version   Show program version.")
                LOG_INFO(" -h --help      This menu.")
                LOG_INFO("")

                break;
            }
            default:
            {
                LOG_ERROR("Unknown input parameters!")
            }
        }
    }



    return static_cast<int>(executionStatus);
}
```

Файл: CSBashParser.hpp

```
/*********************************************************************
***********
 * Project       SBash (Solid Bash)
 * (c) copyright 2020
 * Creator       Kirill Rud
 *               All rights reserved
```

```
*****************************************************************************
***/

      /**
       * @file:    CBashParser.hpp
       *
       * @author:  Kirill Rud
       *
       * @created: Jan 11, 2020
       *
       * @brief    This file contains the CBashParser class declaration.
This class
       * provides ability to parse source files written in SBash language
(usually
       * they have the *.sb extension) into the AST tree.
       */

      #pragma once

      #include "imp/parser/api/ISBashParser.hpp"
      #include "imp/parser/src/InternalASTNode.hpp"

      namespace sbash
      {
         namespace parser
         {
            class CSBashParser final : public ISBashParser
            {
               public:

                  virtual void addSourceFile(SourceFile&& sourceFile)
override final;
                  virtual void clear() override final;
                  virtual ParsingResult parse() override final;

               private:

                  struct LexedFile
                  {
                     std::string                 filename;
```

```
                    std::vector<InternalASTNode> nodes;
                };

                struct ParsedFile
                {
                    std::string     filename;
                    sbash::sdk::AST ast;
                };

                using tSourceFiles = std::vector<SourceFile>;
                using tLexedFiles  = std::vector<LexedFile>;
                using tParsedFiles = std::vector<ParsedFile>;
                using tDataVariant = std::variant<tSourceFiles,
tLexedFiles, tParsedFiles, sbash::sdk::AST>;

                struct StageStatus
                {
                    tDataVariant                             data;
                    std::vector<sbash::sdk::CompilerCaution> errors;
                };

            private:

                StageStatus lexSourceFiles(tDataVariant&& sourceData)
const;
                StageStatus parseSourceFiles(tDataVariant&& lexedData)
const;
                StageStatus mergeASTRoots(tDataVariant&& parsedData)
const;

            private:

                tSourceFiles mFileSources;
        };
    };
};
```

Файл: CSBashParser.cpp

```
/*********************************************************************
***********
```

```
 * Project       SBash (Solid Bash)
 * (c) copyright 2020
 * Creator       Kirill Rud
 *               All rights reserved

*************************************************************************
***/

/**
 * @file:    CBashParser.cpp
 *
 * @author:  Kirill Rud
 *
 * @created: Jan 11, 2020
 *
 * @brief    Implementation of the CBashParser.hpp
 */

#include "imp/parser/src/CSBashParser.hpp"

#include <boost/range/algorithm/for_each.hpp>
#include <boost/range/adaptor/sliced.hpp>
#include "common/api/tracelog/TraceLogDefinition.hpp"

#include "imp/helper/api/ToStringHelper.hpp"
#include "imp/parser/src/lexerwrapper/CSBashLexerWrapper.hpp"
#include "imp/parser/src/parserwrapper/CSBashParserWrapper.hpp"

DEFINE_SCOPE("CSBashParser")

using namespace ::sbash::sdk;

namespace sbash { namespace parser {

void CSBashParser::addSourceFile(SourceFile&& sourceFile)
{
   mFileSources.push_back( std::move(sourceFile) );
}

void CSBashParser::clear()
{
```

```
        mFileSources.clear();
    }

    static std::vector<std::string> extractFileNames(const
std::vector<SourceFile>& fileSources)
    {
        std::vector<std::string> fileNames;

        fileNames.reserve( fileSources.size() );

        for( const auto& sourceFile : fileSources )
        {
            fileNames.push_back(sourceFile.name);
        }

        return fileNames;
    }

    ParsingResult CSBashParser::parse()
    {
        ParsingResult parsingStatus{ true, std::move(
extractFileNames(mFileSources) ), { }, std::nullopt };

        using tParsginFunction =
std::function<StageStatus(tDataVariant&&)>;

        std::vector<tParsginFunction> parsingStagesFunctions
        {
              [this](tDataVariant&& data) -> StageStatus { return
lexSourceFiles( std::move(data) );    }
            , [this](tDataVariant&& data) -> StageStatus { return
parseSourceFiles( std::move(data) );  }
            , [this](tDataVariant&& data) -> StageStatus { return
mergeASTRoots( std::move(data) );     }
        };

        tDataVariant parsingStageInputData = std::move(mFileSources);

        for( const auto& function : parsingStagesFunctions )
        {
```

```
            auto stageStatus = function( std::move(parsingStageInputData)
);

            if( stageStatus.errors.empty() )
            {
               parsingStageInputData = std::move(stageStatus.data);
            }
            else
            {
               parsingStatus.isSuccess = false;
               parsingStatus.cautions  = std::move(stageStatus.errors);

               break;
            }
         }

         if( parsingStatus.isSuccess )
         {
            parsingStatus.ast = std::move(
std::get<AST>(parsingStageInputData) );
         }

         return parsingStatus;
      }

      CSBashParser::StageStatus CSBashParser::lexSourceFiles(tDataVariant&&
sourceData) const
      {
         tLexedFiles                  lexedFiles;
         std::vector<CompilerCaution> lexingErrors;

         auto& sourceFilesLines = std::get<tSourceFiles>(sourceData);

         CSBashLexerWrapper lexer;

         unsigned int currentFileNameIndex{0};

         lexer.setOnErrorCallback(
            [&lexingErrors, &sourceFilesLines,
&currentFileNameIndex](std::vector<tErrorLine>&& errorLines)
            {
```

```
            for( auto&& line : errorLines )
            {
               lexingErrors.push_back(
                  CompilerCaution
                  {
                       eCautionType::ERROR
                     , "The source file contain unrecognized symbol!"
                     ,
std::string{sourceFilesLines[currentFileNameIndex].name}
                     , std::move(line)
                  }
               );
            }
         }
      );

      for( auto&& [fileName, fileSource] : sourceFilesLines )
      {
         DBG_INFO("Lexing %s file...", fileName)

         lexer.setSource( std::move(fileSource) );

         auto tokens = lexer.tokenize();

         if( tokens.empty() == false )
         {
            lexedFiles.push_back( { std::move(fileName),
std::move(tokens) } );
         }

         ++currentFileNameIndex;
      }

      return StageStatus{ std::move(lexedFiles), std::move(lexingErrors)
};
   }

   CSBashParser::StageStatus
CSBashParser::parseSourceFiles(tDataVariant&& lexedData) const
   {
      tParsedFiles                 parsedFiles;
```

```
        std::vector<CompilerCaution> parsingErrors;

        auto& lexedFilesLines = std::get<tLexedFiles>(lexedData);

        CSBashParserWrapper parser;

        unsigned int currentFileNameIndex{0};

        parser.setOnErrorCallback(
           [&parsingErrors, &lexedFilesLines,
&currentFileNameIndex](std::vector<tErrorLine>&& errorLines)
           {
              for( auto&& line : errorLines )
              {
                 parsingErrors.push_back(
                    CompilerCaution
                    {
                         eCautionType::ERROR
                       , "The source file contain unexpected
construction!"
                       ,
std::string{lexedFilesLines[currentFileNameIndex].filename}
                       , std::move(line)
                    }
                 );
              }
           }
        );

        for( auto&& [fileName, nodes] : lexedFilesLines )
        {
           DBG_INFO("Parsing %s file...", fileName)

           parser.setSource( std::move(nodes) );

           auto ast = parser.parse();

           parsedFiles.push_back( { std::move(fileName), std::move(ast) }
);

           ++currentFileNameIndex;
```

```
        }

        return StageStatus{ std::move(parsedFiles), parsingErrors };
    }

    static void updateFileLinks(const std::string& fileName,
boost::adaptors::sliced_range<std::vector<ASTNode>>& astNodes)
    {
        for( auto& astNode : astNodes )
        {
            astNode.file = fileName;

            auto& childNodes = astNode.childs;

            if( childNodes.empty() == false )
            {
                auto iteratorsWrapper = childNodes |
boost::adaptors::sliced( 0, childNodes.size() );

                updateFileLinks(fileName, iteratorsWrapper);
            }
        }
    }

    CSBashParser::StageStatus CSBashParser::mergeASTRoots(tDataVariant&&
parsedData) const
    {
        StageStatus stageStatus{ AST{ }, { } };

        auto& rootAST = std::get<AST>(stageStatus.data);

        auto& parsedFiles = std::get<tParsedFiles>(parsedData);

        auto& [root, files] = rootAST;

        for( auto&& parsedFileData : parsedFiles )
        {
            auto&& [parsedFileName, parsedAST]   = parsedFileData;
            auto&& [parsedRoot, parsedFileNames] = parsedAST;

            #if DEVELOPMENT_BUILD
```

```
                DBG_INFO( "File: [%s], AST Tree: [%s]", parsedFileName,
helper::tostring::toString(parsedAST) )

            #endif

            const auto astNodesWithBrokenLinksBeginIndex = root.size();

            files.push_back( std::move(parsedFileName) );
            root.insert( root.begin(), parsedRoot.begin(), parsedRoot.end()
);

            const auto astNodesWithBrokenLinksEndIndex = root.size();

            const auto& currentParsedTreeFileName = files.back();
            auto        astNodesWithBrokenLinks   = root |
boost::adaptors::sliced(astNodesWithBrokenLinksBeginIndex,
astNodesWithBrokenLinksEndIndex);

            updateFileLinks(currentParsedTreeFileName,
astNodesWithBrokenLinks);
        }

        return stageStatus;
    }

    }; };
```

Файл: CSBashCompiler.hpp

```
    /*****************************************************************
***********
     * Project       SBash (Solid Bash)
     * (c) copyright 2020
     * Creator       Kirill Rud
     *               All rights reserved

***************************************************************************
***/

    /**
```

```
 * @file:    CSBashCompiler.hpp
 *
 * @author:  Kirill Rud
 *
 * @created: May 6, 2020
 *
 * @brief    This file contains the CSBashCompiler class declaration.
This
 * class provides ability to compile AST built from parsed source
files written
 * in SBash language.
 */

#pragma once

#include "imp/compiler/api/ISBashCompiler.hpp"
#include "imp/compiler/src/resources/CCompilerResourceKeeper.hpp"

namespace sbash
{
   namespace compiler
   {
      class CSBashCompiler final : public ISBashCompiler
      {
         public:

            CSBashCompiler();

            virtual void setOption(const eCompileOption option, const
bool status) override final;
            virtual void setAST(sbash::sdk::AST&& ast) override
final;
            virtual void clear() override final;
            virtual CompilerResult compile() override final;

         private:

            CCompilerResourceKeeper        mCompilerResourceKeeper;
            std::optional<sbash::sdk::AST> mAST;
      };
   };
```

```
};
```

Файл: CSBashCompiler.cpp

```
/**********************************************************************
***********
 * Project       SBash (Solid Bash)
 * (c) copyright 2020
 * Creator       Kirill Rud
 *               All rights reserved

***************************************************************************
***/

/**
 * @file:    CSBashCompiler.cpp
 *
 * @author:  Kirill Rud
 *
 * @created: May 6, 2020
 *
 * @brief    Implementation of the CSBashCompiler.hpp
 */

#include "imp/compiler/src/CSBashCompiler.hpp"

#include "common/api/tracelog/TraceLogDefinition.hpp"

#include "imp/compiler/src/resources/CCompilerResourceCautions.hpp"
#include
"imp/compiler/src/resources/CCompilerResourceEnumerations.hpp"
#include "imp/compiler/src/resources/CCompilerResourceNamespaces.hpp"
#include "imp/compiler/src/resources/CCompilerResourceOptions.hpp"
#include
"imp/compiler/src/resources/CCompilerResourceSourceLines.hpp"
#include "imp/compiler/src/resources/CCompilerResourceStructures.hpp"
#include "imp/compiler/src/drivers/CCompilerDriverRoot.hpp"
#include "imp/helper/api/ToStringHelper.hpp"

DEFINE_SCOPE("CSBashCompiler")
```

```
namespace sbash { namespace compiler {

CSBashCompiler::CSBashCompiler()
: mCompilerResourceKeeper{}
, mAST{}
{
   mCompilerResourceKeeper.addResource<CCompilerResourceOptions>();
}

void CSBashCompiler::setOption(const eCompileOption option, const
bool status)
{
   mCompilerResourceKeeper.getResource<CCompilerResourceOptions>()-
>setOption(option, status);
}

void CSBashCompiler::setAST(sbash::sdk::AST&& ast)
{
   mAST = std::move(ast);
}

void CSBashCompiler::clear()
{
   mAST = std::nullopt;
}

CompilerResult CSBashCompiler::compile()
{
   CompilerResult compilerResult{ true, false, {}, {} };

   if( mAST.has_value() == false )
   {
      DBG_ERROR("AST is not set!")
   }
   else
   {
      DBG_INFO("Code compilation is started.")

      mCompilerResourceKeeper.addResource<CCompilerResourceCautions>(
std::ref(compilerResult) );
```

```
mCompilerResourceKeeper.addResource<CCompilerResourceSourceLines>(
std::ref( mAST.value() ) );

mCompilerResourceKeeper.addResource<CCompilerResourceNamespaces>("global");

mCompilerResourceKeeper.addResource<CCompilerResourceEnumerations>();

mCompilerResourceKeeper.addResource<CCompilerResourceStructures>();

           tASTNodeRefs astNodeRefs;
           for( auto& astNode : mAST->root )
           {
              astNodeRefs.push_back( std::ref(astNode) );
           }

           CCompilerDriverRoot driverRoot{
std::ref(mCompilerResourceKeeper), astNodeRefs };

           driverRoot.run();

           DBG_INFO("Code compilation is complited.")

           if( compilerResult.isSuccess )
           {
              compilerResult.aps = std::move( driverRoot.fetchAPS() );

              DBG_INFO("Program: %s", sbash::helper::tostring::toString(
compilerResult.aps.value() ) )
           }


mCompilerResourceKeeper.removeResource<CCompilerResourceCautions>();

mCompilerResourceKeeper.removeResource<CCompilerResourceSourceLines>();

mCompilerResourceKeeper.removeResource<CCompilerResourceNamespaces>();

mCompilerResourceKeeper.removeResource<CCompilerResourceEnumerations>();

mCompilerResourceKeeper.removeResource<CCompilerResourceStructures>();
```

```
        }

        return compilerResult;
    }

    }; };
```

Файл: CCompilerDriverRoot.hpp

```
    /*******************************************************************
***********
     * Project       SBash (Solid Bash)
     * (c) copyright 2020
     * Creator       Kirill Rud
     *               All rights reserved

***************************************************************************
***/

    /**
     * @file:    CCompilerDriverRoot.hpp
     *
     * @author:  Kirill Rud
     *
     * @created: May 6, 2020
     *
     * @brief    This file contains the CCompilerDriverRoot class
declaration. This
     * class is from drivers family, it handles top program structure.
     */

    #pragma once

    #include "imp/compiler/src/drivers/CCompilerDriverBase.hpp"

    namespace sbash
    {
       namespace compiler
       {
          class CCompilerDriverRoot final : public CCompilerDriverBase
          {
```

```
            public:

                CCompilerDriverRoot(tResourceKeeperRef resourceKeeperRef,
tASTNodeRefs astNodeRefs);
                virtual ~CCompilerDriverRoot() = default;

                virtual void run() override final;
            };
        };
    };
```

Файл: CCompilerDriverRoot.cpp

```
    /*********************************************************************
***********
     * Project       SBash (Solid Bash)
     * (c) copyright 2020
     * Creator       Kirill Rud
     *               All rights reserved

***********************************************************************
***/

    /**
     * @file:    CCompilerDriverRoot.cpp
     *
     * @author:  Kirill Rud
     *
     * @created: May 6, 2020
     *
     * @brief    Implementation of the CCompilerDriverRoot.hpp
     */

    #include "imp/compiler/src/drivers/CCompilerDriverRoot.hpp"

    #include "common/api/tracelog/TraceLogDefinition.hpp"

    #include "imp/helper/api/ToStringHelper.hpp"
    #include "imp/compiler/src/rules/CompilerRuleFactory.hpp"
    #include "imp/compiler/src/rules/CCompilerRuleGlobalNamespace.hpp"
    #include "imp/compiler/src/drivers/DriversHelper.hpp"
```

```
#include
"imp/compiler/src/drivers/CCompilerDriverEnumerationDeclaration.hpp"
#include
"imp/compiler/src/drivers/CCompilerDriverFunctionDeclaration.hpp"
#include
"imp/compiler/src/drivers/CCompilerDriverStructureDeclaration.hpp"
#include
"imp/compiler/src/drivers/CCompilerDriverVariableDeclaration.hpp"
#include "imp/compiler/src/ASTHelper.hpp"

DEFINE_SCOPE("CCompilerDriverRoot")

using namespace ::sbash::helper::tostring;
using namespace ::sbash::sdk;

namespace sbash { namespace compiler {

CCompilerDriverRoot::CCompilerDriverRoot(tResourceKeeperRef
resourceKeeperRef, tASTNodeRefs astNodeRefs)
: CCompilerDriverBase{resourceKeeperRef, astNodeRefs}
{

}

void CCompilerDriverRoot::run()
{
   DBG_FUNCTION()

   for( auto& astNodeRef : mASTNodeRefs )
   {
      auto astRef = convertToASTRefs(astNodeRef);

      if( validate<CCompilerRuleGlobalNamespace>(mResourceKeeperRef,
astRef) )
      {
         auto& node = astNodeRef.get();

         switch( node.type )
         {
            case eTokenType::KEYWORD_TYPE_INTEGER:
            case eTokenType::KEYWORD_TYPE_DOUBLE:
```

```
                    case eTokenType::KEYWORD_TYPE_BOOLEAN:
                    case eTokenType::KEYWORD_TYPE_STRING:
                    case eTokenType::LETERAL_NAME:
                    {

delegateExecution<CCompilerDriverVariableDeclaration>(this, astNodeRef,
mAPS);

                       break;
                    }
                    case eTokenType::KEYWORD_FUNCTION:
                    case eTokenType::KEYWORD_FUNCTION_BASH:
                    {

delegateExecution<CCompilerDriverFunctionDeclaration>(this, astNodeRef,
mAPS);

                       break;
                    }
                    case eTokenType::KEYWORD_ENUMERATION:
                    {

delegateExecution<CCompilerDriverEnumerationDeclaration>(this, astNodeRef,
mAPS);

                       break;
                    }
                    case eTokenType::KEYWORD_STRUCTURE:
                    {

delegateExecution<CCompilerDriverStructureDeclaration>(this, astNodeRef,
mAPS);

                       break;
                    }
                    default:
                    {
                       DBG_ERROR( "Impossible to handle AST node with type:
%s", toString(node.type) )
                    }
                 }
```

```
        }
    }
}

}; };
```

Файл: ICompilerResourceBase.hpp

```
/**********************************************************************
***********
 * Project       SBash (Solid Bash)
 * (c) copyright 2020
 * Creator       Kirill Rud
 *               All rights reserved

****************************************************************************
***/

/**
 * @file:    ICompilerResourceBase.hpp
 *
 * @author:  Kirill Rud
 *
 * @created: May 6, 2020
 *
 * @brief    This file contains the ICompilerResourceBase class
declaration.
 * This class provide ability to create a resource.
 */

#pragma once

namespace sbash
{
   namespace compiler
   {
      enum class eResource
      {
           COMPILER_OPTIONS
         , COMPILER_CAUTIONS
         , NAMESPACES
```

```
            , SOURCE_LINES
            , ENUMERATIONS
            , STRUCTURES
        };

        class ICompilerResource
        {
            public:

                virtual ~ICompilerResource() = default;
        };

        template <eResource TResourceType>
        class ICompilerResourceBase : public ICompilerResource
        {
            public:

                virtual ~ICompilerResourceBase() = default;

                static eResource getResourceType()
                {
                    return TResourceType;
                }
        };
    };
};
```

Файл: CCompilerResourceKeeper.hpp

```
/*******************************************************************
***********
 * Project       SBash (Solid Bash)
 * (c) copyright 2020
 * Creator       Kirill Rud
 *               All rights reserved

*********************************************************************
***/

/**
 * @file:    CCompilerResourceKeeper.hpp
```

```
 *
 * @author:  Kirill Rud
 *
 * @created: May 6, 2020
 *
 * @brief    This file contains the CCompilerResourceKeeper class
declaration.
 * This class provide ability to get compiler resources.
 */

#pragma once

#include <memory>
#include <unordered_map>

#include "imp/compiler/src/resources/ICompilerResourceBase.hpp"

namespace sbash
{
   namespace compiler
   {
      class CCompilerResourceKeeper
      {
         public:

            template <class TResourceClass, class ... TResourceArgs>
            void addResource(TResourceArgs&& ... args)
            {
               mResources[ TResourceClass::getResourceType() ] =
std::make_shared<TResourceClass>( std::forward<TResourceArgs>(args) ... );
            }

            template <class TResourceClass>
            void removeResource()
            {
               mResources.erase( TResourceClass::getResourceType() );
            }

            template <class TResourceClass>
            std::shared_ptr<TResourceClass> getResource()
            {
```

```
                    std::shared_ptr<ICompilerResource>
resourcePtr{nullptr};

                    auto resourceIter = mResources.find(
TResourceClass::getResourceType() );

                    if( resourceIter != mResources.end() )
                    {
                       resourcePtr = resourceIter->second;
                    }

                    return
std::dynamic_pointer_cast<TResourceClass>(resourcePtr);
                 }

              private:

                 std::unordered_map<eResource,
std::shared_ptr<ICompilerResource>> mResources;
           };
        };
     };
```

Файл: CCompilerResourceNamespaces.hpp

```
     /*********************************************************************
***********
      * Project       SBash (Solid Bash)
      * (c) copyright 2020
      * Creator       Kirill Rud
      *               All rights reserved

**************************************************************************
***/

     /**
      * @file:    CCompilerResourceNamespaces.hpp
      *
      * @author:  Kirill Rud
      *
      * @created: May 8, 2020
```

```
      *
      * @brief    This file contains the CCompilerResourceNamespaces class
      * declaration. This class is from resources family, it stores names
in
      * namespaces and provides access to them.
      */

     #pragma once

     #include <vector>
     #include <unordered_map>
     #include "sdk/api/TokenType.hpp"

     #include "imp/compiler/src/resources/ICompilerResourceBase.hpp"
     #include "imp/compiler/api/ISBashCompiler.hpp"

     namespace sbash
     {
        namespace compiler
        {
           enum class eParameter
           {
              TYPE
           };

           struct ScopeEntry
           {
              std::string                                  name;
              std::string                                  file;
              unsigned int                                 line;
              std::vector<sbash::sdk::eTokenType>          attributes;
              std::unordered_map<eParameter, std::string> parameters;
           };

           class CCompilerResourceNamespaces : public
ICompilerResourceBase<eResource::NAMESPACES>
           {
              public:

                 CCompilerResourceNamespaces(const std::string&
globalScopeName);
```

```
                bool isEntryNameUnique(const std::string& entryName);
                std::optional<ScopeEntry> getEntry(const std::string&
entryName);
                std::optional<std::string> getEntryScopeName(const
std::string& entryName);
                void setEntry(ScopeEntry&& scopeEntry);

                void moveInsideScope(const std::string& scopeName);
                void moveOutsideScope();

             private:

                struct InternalScopeEntry
                {
                   ScopeEntry  entry;
                   std::string scopeName;
                };

                struct Scope
                {
                   std::string                    name;
                   std::shared_ptr<Scope>         parrentScopePtr;
                   std::vector<InternalScopeEntry> entries;
                };

             private:

                std::optional<InternalScopeEntry> getInternalEntry(const
std::string& entryName);
                std::string getNestedScopeNameFromCurrent() const;

             private:

                std::shared_ptr<Scope>            mCurrentScopePtr;
                std::optional<InternalScopeEntry> mPreviousEntrySearch;
          };
       };
    };
```

Файл: CCompilerResourceNamespaces.cpp

```
/********************************************************************
***********
 * Project       SBash (Solid Bash)
 * (c) copyright 2020
 * Creator       Kirill Rud
 *               All rights reserved

*****************************************************************************
***/

/**
 * @file:    CCompilerResourceNamespaces.cpp
 *
 * @author:  Kirill Rud
 *
 * @created: May 8, 2020
 *
 * @brief    Implementation of the CCompilerResourceNamespaces.hpp
 */

#include "imp/compiler/src/resources/CCompilerResourceNamespaces.hpp"

#include <sstream>
#include <functional>
#include <list>
#include <boost/range/algorithm/find_if.hpp>
#include <boost/algorithm/string/join.hpp>
#include "common/api/tracelog/TraceLogDefinition.hpp"

DEFINE_SCOPE("CCompilerResourceNamespaces")

namespace sbash { namespace compiler {

CCompilerResourceNamespaces::CCompilerResourceNamespaces(const
std::string& globalScopeName)
: mCurrentScopePtr{ std::make_shared<Scope>() }
, mPreviousEntrySearch{ std::nullopt }
{
   mCurrentScopePtr->name = globalScopeName;
}
```

```
      bool CCompilerResourceNamespaces::isEntryNameUnique(const
std::string& entryName)
      {
         return (getInternalEntry(entryName).has_value() == false);
      }

      std::optional<ScopeEntry> CCompilerResourceNamespaces::getEntry(const
std::string& entryName)
      {
         std::optional<ScopeEntry> scopeEntry;

         auto internalScopeEntryOptional = getInternalEntry(entryName);

         if( internalScopeEntryOptional.has_value() )
         {
            scopeEntry = internalScopeEntryOptional->entry;
         }

         return scopeEntry;
      }

      std::optional<std::string>
CCompilerResourceNamespaces::getEntryScopeName(const std::string&
entryName)
      {
         std::optional<std::string> scopeName;

         auto internalScopeEntryOptional = getInternalEntry(entryName);

         if( internalScopeEntryOptional.has_value() )
         {
            scopeName = internalScopeEntryOptional->scopeName + "_" +
entryName;
         }

         return scopeName;
      }

      static std::string toString(const eParameter& parameter)
      {
```

```
        switch( parameter )
        {
            case eParameter::TYPE: return "TYPE";
        }
    }

    void CCompilerResourceNamespaces::setEntry(ScopeEntry&& scopeEntry)
    {
        mPreviousEntrySearch = std::nullopt;

        const auto& entryName = scopeEntry.name;

        if( isEntryNameUnique(entryName) )
        {
            DBG_INFO("The entry with name %s has been added, from file: %s,
at line: %s", entryName, scopeEntry.file, std::to_string(scopeEntry.line))

            std::stringstream scopeEntryParametersAndAttributesStream;

            auto currentNestedScope = getNestedScopeNameFromCurrent();

            scopeEntryParametersAndAttributesStream << "Parameter: [";
            for( const auto& parameter : scopeEntry.parameters )
            {
                scopeEntryParametersAndAttributesStream << " ";
                scopeEntryParametersAndAttributesStream <<
toString(parameter.first);
                scopeEntryParametersAndAttributesStream << "=";
                scopeEntryParametersAndAttributesStream << parameter.second;
            }
            scopeEntryParametersAndAttributesStream << " ], attributes: [";
            for( const auto& attribute : scopeEntry.attributes )
            {
                scopeEntryParametersAndAttributesStream << " ";
                scopeEntryParametersAndAttributesStream <<
sbash::sdk::toString(attribute);
            }
            scopeEntryParametersAndAttributesStream << " ], scope: ";
            scopeEntryParametersAndAttributesStream << currentNestedScope;

            DBG_INFO( "%s", scopeEntryParametersAndAttributesStream.str() )
```

```
            mCurrentScopePtr->entries.push_back( InternalScopeEntry{
std::move(scopeEntry), std::move(currentNestedScope) } );
         }
         else
         {
            DBG_INFO("The entry with name %s hasn't been added, because it
already exists.", entryName)
         }
      }

      void CCompilerResourceNamespaces::moveInsideScope(const std::string&
scopeName)
      {
         DBG_INFO("%s -> %s", mCurrentScopePtr->name, scopeName)

         auto newScopePtr = std::make_shared<Scope>();

         newScopePtr->name             = scopeName;
         newScopePtr->parrentScopePtr = std::move(mCurrentScopePtr);

         mCurrentScopePtr = std::move(newScopePtr);
      }

      void CCompilerResourceNamespaces::moveOutsideScope()
      {
         mPreviousEntrySearch = std::nullopt;

         if( mCurrentScopePtr->parrentScopePtr != nullptr )
         {
            DBG_INFO("%s <- %s", mCurrentScopePtr->parrentScopePtr->name,
mCurrentScopePtr->name)

            mCurrentScopePtr = std::move(mCurrentScopePtr-
>parrentScopePtr);
         }
         else
         {
            DBG_ERROR("The %s scope is on the top!", mCurrentScopePtr-
>name)
```

```
        }
    }

    std::optional<CCompilerResourceNamespaces::InternalScopeEntry>
CCompilerResourceNamespaces::getInternalEntry(const std::string& entryName)
    {
        std::optional<InternalScopeEntry> internalScopeEntry;

        std::function<std::optional<InternalScopeEntry>(const Scope&)>
findEntryNameUniqueInScope;
        findEntryNameUniqueInScope = [&entryName,
&findEntryNameUniqueInScope](const auto& scope)
        {
            std::optional<InternalScopeEntry> scopeEntry;

            const auto& entries = scope.entries;

            const auto searchIter = boost::range::find_if(entries,
                [&entryName](const auto& internalEntry)
                {
                    return internalEntry.entry.name == entryName;
                }
            );

            if( searchIter == entries.end() )
            {
                if( scope.parrentScopePtr != nullptr )
                {
                    findEntryNameUniqueInScope( *(scope.parrentScopePtr) );
                }
            }
            else
            {
                scopeEntry = InternalScopeEntry{ *searchIter };
            }

            return scopeEntry;
        };

        if( mPreviousEntrySearch.has_value() && mPreviousEntrySearch-
>entry.name == entryName )
```

```
         {
            DBG_INFO("Search duplication -> return previous result.")

            internalScopeEntry = mPreviousEntrySearch;
         }
         else
         {
            internalScopeEntry =
findEntryNameUniqueInScope(*mCurrentScopePtr);

            if( internalScopeEntry.has_value() )
            {
               DBG_INFO("The entry with name %s has been found.",
entryName)
            }
            else
            {
               DBG_INFO("The entry with name %s hasn't been found.",
entryName)
            }

            mPreviousEntrySearch = internalScopeEntry;
         }

         return internalScopeEntry;
      }

      std::string
CCompilerResourceNamespaces::getNestedScopeNameFromCurrent() const
      {
         std::list<std::string> nestedScopeNames;

         std::shared_ptr<Scope> extractionScopePtr = mCurrentScopePtr;

         while( extractionScopePtr != nullptr )
         {
            nestedScopeNames.push_front( extractionScopePtr->name );

            extractionScopePtr = extractionScopePtr->parrentScopePtr;
         }
```

```
   return boost::algorithm::join(nestedScopeNames, "_");
}

}; };
```

Файл: CMakeLists.txt (головний)

```
######################################################################
###########
 # Project       SBash (Solid Bash)
 # (c) copyright 2020
 # Creator       Kirill Rud
 #               All rights reserved
######################################################################
###########

###
 # @file:    CMakeLists.txt
 #
 # @author:  Kirill Rud
 #
 # @created: Jan 3, 2020
 #
 # @brief    This is the main cmake file. It configures basic project
 # preferences and build details.
###

cmake_minimum_required(VERSION 3.2)

project(SBash VERSION 0.1 LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)

set(CMAKE_CXX_FLAGS  "${CMAKE_CXX_FLAGS} -Werror")

if("${BUILD_TYPE}" STREQUAL "RELEASE")
         message(STATUS "Build type: ${BUILD_TYPE}")
         add_compile_definitions(RELEASE_BUILD)
elseif("${BUILD_TYPE}" STREQUAL "DEVELOPMENT")
         message(STATUS "Build type: ${BUILD_TYPE}")
```

```
		add_compile_definitions(DEVELOPMENT_BUILD)
	else()
		message(STATUS "Undefined build type, using DEVELOPMENT as 
default.")
		add_compile_definitions(DEVELOPMENT_BUILD)
	endif()

	set(BUILD_TYPE "DEVELOPMENT" CACHE STRING "BUILD_TYPE")

	set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/../bin)
	set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/../bin)

	find_package(Boost 1.62 COMPONENTS program_options REQUIRED)

	set(BOOST_INCLUDE_DIRECTORY "${Boost_INCLUDE_DIR}" CACHE STRING 
"BOOST_INCLUDE_DIRECTORY")

	find_program(PATH_TO_FLEXCXX_GENERATOR NAMES flexc++)
	find_program(PATH_TO_BISONCXX_GENERATOR NAMES bisonc++)

	if("${PATH_TO_FLEXCXX_GENERATOR}" STREQUAL 
"PATH_TO_FLEXCXX_GENERATOR-NOTFOUND")
	   message(FATAL_ERROR "\nThe flexc++ is not installed!\nPlease, 
install flexc++ (suggestion: sudo apt install flexc++)." )
	endif()

	if("${PATH_TO_BISONCXX_GENERATOR}" STREQUAL 
"PATH_TO_BISONCXX_GENERATOR-NOTFOUND")
	   message(FATAL_ERROR "\nThe bisonc++ is not installed!\nPlease, 
install bisonc++ (suggestion: sudo apt install bisonc++)." )
	endif()

	set(PATH_TO_FLEXCXX_GENERATOR "${PATH_TO_FLEXCXX_GENERATOR}" CACHE 
STRING "PATH_TO_FLEXCXX_GENERATOR")
	set(PATH_TO_BISONCXX_GENERATOR "${PATH_TO_BISONCXX_GENERATOR}" CACHE 
STRING "PATH_TO_BISONCXX_GENERATOR")

	if("${BUILD_TYPE}" STREQUAL "DEVELOPMENT")
	   message(STATUS "Unit Tests is activated for development build.")
	else()
```

```
        message(STATUS "Unit Tests is not activated for non-development
build.")
     endif()

     macro (unit_test UT_NAME UT_INCLUDES UT_LIBS)
        if("${BUILD_TYPE}" STREQUAL "DEVELOPMENT")
           add_executable(${UT_NAME} ${UT_NAME}.cpp)
           add_custom_command(TARGET ${UT_NAME} POST_BUILD COMMAND
${UT_NAME})

           target_compile_options(${UT_NAME} PRIVATE -
DBOOST_TEST_DYN_LINK)
           target_include_directories(${UT_NAME} PRIVATE
"${BOOST_INCLUDE_DIRECTORY}")

           foreach(UT_INCLUDE_PATH ${UT_INCLUDES})
              target_include_directories(${UT_NAME} PRIVATE
"${UT_INCLUDE_PATH}")
           endforeach(UT_INCLUDE_PATH)

           target_link_libraries(${UT_NAME} LINK_PRIVATE -
lboost_unit_test_framework ${UT_LIBS})
        endif()
     endmacro(unit_test)

     add_subdirectory(libs)
     add_subdirectory(sbash)
```

Файл: CMakeLists.txt (для виконавчого файлу компілятора)

```
###########################################################################
#####
      # Project       SBash (Solid Bash)
      # (c) copyright 2020
      # Creator       Kirill Rud
      #               All rights reserved
     #####################################################################
###########

     ###
```

```
 # @file:    CMakeLists.txt
 #
 # @author:  Kirill Rud
 #
 # @created: Jan 3, 2020
 #
 # @brief    This cmake file configure the main SBash executable.
###

set(EXECUTABLE_NAME "sbash")

add_executable(${EXECUTABLE_NAME} SBashMain.cpp)

list(APPEND LIBS compiler)
list(APPEND LIBS helper)
list(APPEND LIBS parser)

foreach(libraryName ${LIBS})
   add_subdirectory(${libraryName})
endforeach(libraryName)

list(APPEND LIBS sbash-common)
list(APPEND LIBS sbash-sdk)

list(APPEND BOOST_LIBS -lboost_system)
list(APPEND BOOST_LIBS -lboost_filesystem)

list(APPEND STD_LIBS -pthread)

target_include_directories(${EXECUTABLE_NAME} PRIVATE
"${SBASH_INCLUDE_DIRECTORY}")
target_include_directories(${EXECUTABLE_NAME} PRIVATE
"${SBASH_LIBS_INCLUDE_DIRECTORY}")
target_include_directories(${EXECUTABLE_NAME} PRIVATE
"${BOOST_INCLUDE_DIRECTORY}")

target_link_libraries(${EXECUTABLE_NAME} LINK_PRIVATE ${LIBS}
${BOOST_LIBS} ${STD_LIBS})
```