НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування і спеціалізованих комп‘ютерних систем

«До захисту допущено»
Завідувач кафедри
_____ В.О. РОМАНКЕВИЧ
(підпис) (ініціали, прізвище)
“___” червня 2021 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Системне програмування та спеціалізовані комп’ютерні системи”

зі спеціальності **123 «Комп'ютерна інженерія»**

на тему: Засіб порівняння версій програм на мові LISP з використанням абстрактного семантичного дерева

Виконав : студент IV курсу, групи КВ-73
Єрмоленко Денис Вадимович
(підпис)

Керівник доц. каф. СП і СКС, к.т.н. Марченко О. І.
(підпис)

Консультант з нормоконтролю, доц.каф. СП і СКС, к.т.н. Клятченко Я.М.
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) (підпис)

Рецензент_____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2021 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування і спеціалізованих комп'ютерних систем

Рівень вищої освіти – перший (бакалаврський)

Спеціальність 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Системне програмування та спеціалізовані комп'ютерні системи»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ В.О. РОМАНКЕВИЧ
(підпис)
«__» червня 2021 р.

ЗАВДАННЯ
на дипломний проєкт студента
Єрмоленка Дениса Вадимовича

1. Тема проєкту «Засіб порівняння версій програм на мові LISP з використанням абстрактного семантичного дерева»,
керівник проєкту доц.каф. СП і СКС, к.т.н. Марченко Олександр Іванович,
затверджені наказом по університету від «_____»_____2021р. №_____

2. Термін подання студентом проєкту _____

3. Вихідні дані до проєкту: див. Технічне завдання.

4. Зміст пояснювальної записки:
- Аналіз існуючих засобів порівняння;
- Розробка засобу;
- Тестування засобу;
- Керівництво користувача.

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслеників, плакатів, презентацій тощо):
- Структурна схема засобу. Схема структурна;

• UML діаграма ієрархії класів об’єктної системи вузлів абстрактного семантичного дерева. Схема структурна;
• Лексичний аналізатор. Схема алгоритму;
• Алгоритм порівняння абстрактних семантичних дерев. Схема алгоритму.
• Презентація за темою роботи.

6. Консультанти розділів проєкту[1]

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Клятченко Я.М., к.т.н., доцент каф. СП і СКС		

7. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1.	Вивчення літератури за тематикою проєкту	15.11.2020	
2.	Розроблення та узгодження технічного завдання	30.11.2020	
3.	Аналіз існуючих рішень	10.01.2021	
4.	Підготовка матеріалів першого розділу дипломного проєкту	12.04.2021	
5.	Підготовка матеріалів другого розділу дипломного проєкту	20.04.2021	
6.	Підготовка матеріалів третього розділу дипломного проєкту	27.04.2021	
7.	Підготовка матеріалів четвертого розділу дипломного проєкту	10.05.2021	
8.	Підготовка графічної частини дипломного проєкту	18.05.2021	
9.	Оформлення документації дипломного проєкту	23.05.2021	
10.	Попередній огляд матеріалів диплому	26.05.2021	

Студент _____ Денис ЄРМОЛЕНКО
(підпис)

Керівник проєкту _____ Олександр МАРЧЕНКО
(підпис)

[1] Консультантом не може бути зазначено керівника дипломного проєкту.

АНОТАЦІЯ

Кваліфікаційна робота включає пояснювальну записку (62 с., 37 рис., 4 додатки).

Метою розробки є створення засобу порівняння версій програм на мові LISP із застосуванням абстрактного семантичного дерева, який допоможе підвищити ефективність та продуктивність розроблення програмного забезпечення на мові LISP.

Розроблений засіб дозволяє:

- порівнювати структурно між собою 2 вхідні файли із текстами програм на мові LISP;
- виявляти лексичні, синтаксичні чи семантичні помилки у вхідних файлах та надавати зручний графічний інтерфейс для демонстрації помилок;
- класифікувати ідентифікатори s-виразів верхнього рівня;
- виявляти s-вирази, що були додані, видалені чи переміщені до іншого s-виразу в новій версії програми на мові LISP з урахуванням їхньої семантики й виділяти їх певним кольором всередині графічного інтерфейсу;
- дивитися окремо результати порівняння вмісту s-виразів верхнього рівня, що мають ідентифікатори.

В ході розробки засобу:

- проведено аналіз існуючих рішень;
- використано мови програмування Common Lisp та C++;
- використано фреймворк Qt5 для створення графічного інтерфейсу;
- розроблено backend частину засобу;
- розроблені юніт-тести для тестування backend частини засобу;
- розроблено frontend частину засобу.

Використання цього засобу допоможе підвищити ефективність та продуктивність розробки програмного забезпечення на мові LISP.

Ключові слова: LISP, АБСТРАКТНЕ СЕМАНТИЧНЕ ДЕРЕВО, ЛЕКСИЧНИЙ АНАЛІЗ, СИНТАКСИЧНИЙ АНАЛІЗ, C++, QT, СТРУКТУРНЕ ПОРІВНЯННЯ, GUI, ДОПОМІЖНИЙ ЗАСІБ.

ABSTRACT

The qualification work includes an explanatory note (62 p., 37 pic., 4 appendices).

The purpose of the development is to create a tool for comparing versions of programs in the LISP language using an abstract semantic tree, which will help increase the efficiency and productivity of software development in the LISP language.

The developed tool allows:

- structural comparison between the 2 input files containing programs in the LISP language;
- detect lexical, syntactic, or semantic errors in the input files and provide a user-friendly graphical interface for demonstrating errors;
- detect s-expressions that have been added, deleted, or moved to another s-expression in the new version of the program in LISP, taking into account their semantics and highlight them in a certain color within the graphical interface;
- see separately the results of comparing the content of top-level s-expressions that have identifiers.

During the implementation of the tool:

- the analysis of existing decisions was carried out;
- the Common Lisp and C++ programming languages were used;
- the Qt5 framework was used to create a graphical interface;
- the backend part was developed;
- unit-tests were developed to test the backend part of the tool;
- the frontend part was developed.

Using this tool will help increase the efficiency and productivity of software development in the LISP language.

Keywords: LISP, ABSTRACT SEMANTIC TREE, LEXICAL ANALYSIS, SYNTACTIC ANALYSIS, C++, QT, STRUCTURAL COMPARISON, GUI, HELPER TOOL.

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
	А4	ІАЛЦ.045300.002 ТЗ	Засіб порівняння версій	4		
			програм на мові LISP			
			з використанням			
			абстрактного семантичного			
			дерева.			
			Техічне завдання			
	А4	ІАЛЦ.045300.003 ТП	Засіб порівняння версій	2		
			програм на мові LISP			
			з використанням			
			абстрактного семантичного			
			дерева.			
			Відомість технічного			
			проекту			
	А4	ІАЛЦ.045300.004 ПЗ	Засіб порівняння версій	62		
			програм на мові LISP			
			з використанням			
			абстрактного семантичного			
			дерева.			
			Пояснювальна записка			

Змін.	Арк.	№ докум.	Підпис	Дата	ІАЛЦ.045300.001 ОА		
Розробив		Єрмоленко Д.В.			Засіб порівняння версій програм на мові LISP з використанням абстрактного семантичного дерева Опис альбому	Літ.	Аркуш 1 / Аркушів 2
Перевірив		Марченко О.І.					
Консульт.						КПІ ім. Ігоря Сікорського, ФПМ КВ-73	
Н. контроль		Клятченко Я.М.					
Зав. каф.		Романкевич В.О					

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
	А4	ІАЛЦ.045300.005 Д1	Структурна схема засобу.	1		
			Схема структурна			
	А4	ІАЛЦ.045300.006 Д2	UML діаграма ієрархії	1		
			класів об’єктної систем			
			вузлів абстрактного			
			семантичного дерева			
			Схема структурна.			
	А4	ІАЛЦ.045300.007 Д3	Лексичний аналізатор.			
			Схема алгоритму			
	А4	ІАЛЦ.045300.008 Д4	Алгоритм порівняння	1		
			абстрактних семантичних			
			дерев.			
			Схема алгоритму			
		Диск CD-ROM	Текст пояснювальної	1		
			записки.			
			Графічний матеріал			

Змін.	Арк.	№ докум.	Підпис	Дата	ІАЛЦ.045300.001 ОА	Арк. 2

ЗМІСТ

					ІАЛЦ.045300.002 ТЗ		
Зм	*Лист*	*№ докум.*	*Підп.*	*Дата*			
Розроб.		Єрмоленко Д.В.			Засіб порівняння версій програм на мові LISP з використанням абстрактного семантичного дерева. *Технічне завдання*	*Літ.*	*Лист* 1 / *Листів* 4
Перев.		Марченко О.І.					
Н. контр.		Клятченко Я.М.				НТУУ «КПІ ім. Ігоря Сікорського», ФПМ, КВ-73	
Затв.		Романкевич В.О.					

1. НАЙМЕНУВАННЯ ТА ГАЛУЗЬ ЗАСТОСУВАННЯ

Назва розробки: «Засіб порівняння версій програм на мові LISP з використанням абстрактного семантичного дерева».

Галузь застосування: розробка програмного забезпечення на мові LISP.

2. ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на дипломне проектування на здобуття першого (бакалаврського) рівня вищої освіти, затверджене кафедрою системного програмування і спеціалізованих комп'ютерних систем Національного технічного університету України «Київський Політехнічний Інститут імені Ігоря Сікорського».

3. МЕТА І ПРИЗНАЧЕННЯ РОБОТИ

Метою цього проєкту є створення засобу порівняння версій програм на мові LISP з використанням абстрактного семантичного дерева, який є зручним та інформативним для аналізу семантичної різниці між текстами версій програм на мові LISP, що дасть змогу підвищити ефективність та продуктивність розроблення програмного забезпечення на мові LISP.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом інформації є технічна та науково-технічна література, технічна документація, публікації у періодичних виданнях та електронні статті у мережі Інтернет.

5. ТЕХНІЧНІ ВИМОГИ

5.1 Вимоги до програмного продукту, що розробляється

- наявність зручного графічного інтерфейсу користувача;
- підтримання структурного порівняння між собою 2-х файлів із текстами програм на мові програмування LISP;
- діагностування лексичних, синтаксичних та семантичних помилок у вхідних файлах;
- проведення порівняння між собою відповідних def-s-виразів;
- виявлення s-виразів, що були додані, видалені чи переміщені до іншого s-виразу в новій версії програми на мові LISP із урахуванням їхньої семантики.

5.2 Вимоги до апаратного забезпечення

- Процесор: Intel Core i5;
- Оперативна пам’ять: 4 Гб;
- Простір на диску: 512 мб.

5.3 Вимоги до програмного та апаратного забезпечення користувача

- Операційна система Windows 10.

6. ЕТАПИ РОЗРОБКИ

№ з/п	Назва етапів виконання дипломного проекту	Термін виконання етапів
1.	Вивчення літератури за тематикою проекту	15.11.2020
2.	Розроблення та узгодження технічного завдання	30.11.2020
3.	Аналіз існуючих рішень	10.01.2021
4.	Підготовка матеріалів першого розділу дипломного проекту	12.04.2021
5.	Підготовка матеріалів другого розділу дипломного проекту	20.04.2021
6.	Підготовка матеріалів третього розділу дипломного проекту	27.04.2021
7.	Підготовка матеріалів четвертого розділу дипломного проекту	10.05.2021
8.	Підготовка графічної частини дипломного проекту	18.05.2021
9.	Оформлення документації дипломного проекту	23.05.2021
10.	Попередній огляд матеріалів диплому на кафедрі	26.05.2021

ВІДОМІСТЬ ДИПЛОМНОГО ПРОЄКТУ

№ з/п	Формат	Позначення	Найменування	Кількість листів	Примітка
1	А4	ІАЛЦ.045300.000	Завдання на дипломний проєкт	2	
2	А4	ІАЛЦ.045300.001 ОА	Опис альбому	2	
3	А1	ІАЛЦ.045300.002 ТЗ	Технічне завдання	1	
4	А1	ІАЛЦ.045300.003 ТП	Відомість технічного проєкту	1	
5	А1	ІАЛЦ.045300.004 ПЗ	Пояснювальна записка	62	
6	А1	ІАЛЦ.045300.005 Д1	Структурна схема засобу. Схема структурна	1	
7	А1	ІАЛЦ.045300.006 Д2	UML діаграма ієрархії класів об’єктної системи вузлів абстрактного семантичного дерева. Схема структурна	1	
8	А1	ІАЛЦ.045300.007 Д3	Лексичний аналізатор. Схема алгоритму	1	
9	А1	ІАЛЦ.045300.008 Д4	Алгоритм порівняння абстрактних семантичних дерев. Схема алгоритму	1	

				ІАЛЦ.045300.003 ТП		
	ПІБ	Підп.	Дата			
Розробн.	Єрмоленко Д.В.			Відомість дипломного проєкту	Лист	Листів
Керівн.	Марченко О.І.				1	1
Консульт.					КПІ ім. Ігоря Сікорського Каф. СПіСКС Гр. КВ-73	
Н/контр.	Клятченко Я.М.					
Зав.каф.	Романкевич В.О.					

Пояснювальна записка
до дипломного проєкту

на тему: Засіб порівняння версій програм на мові LISP з використанням абстрактного семантичного дерева

Київ – 2021 року

ЗМІСТ

					ІАЛЦ.045300.004 ПЗ		
Зм.	Арк.	№ докум.	Підп.	Дата			
Розроб.		Єрмоленко Д.В.			Багатофункціональний засіб порівняння версій програм на мові LISP на основі абстрактного семантичного дерева Пояснювальна записка	Літ.	Аркуш 1 / Аркушів 62
Перевір.		Марченко О.І.					
Н. контр.		Клятченко Я.М.				НТУУ "КПІ ім.Ігоря Сікорського" ФПМ КВ-73	
Затв.		Романкевич В.О.					

ДОДАТКИ

Додаток 1. Копії графічного матеріалу.

ІАЛЦ.045300.005 Д1 Структурна схема засобу. Схема структурна.

ІАЛЦ.045300.006 Д2 UML діаграма ієрархії класів об’єктної систем вузлів абстрактного семантичного дерева. Схема структурна.

ІАЛЦ.045300.007 Д3 Лексичний аналізатор. Схема алгоритму.

ІАЛЦ.045300.008 Д4 Алгоритм порівняння абстрактних семантичних дерев. Схема алгоритму.

Додаток 2. Фрагменти програмного коду.

Додаток 3. Презентація.

Додаток 4. Довідка про впровадження.

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

Мова LISP – це підмножина мови програмування Common Lisp;

Терм – вузол дерева синтаксичного розбору;

ASDF (Another System Definition Facility) - це засіб, написаний на мові програмування Common Lisp, для побудови та завантаження проєктів, що написані на мові програмування Common Lisp. По суті, ASDF виконує роботу, аналогічну до тієї, яку виконують для програм на мові програмування C засоби make, dlopen, libc та autoconf;

CLOS – common lisp object system;

def-вузол – це вузол абстрактного семантичного дерева, що описує def-s-вираз;

def-s-вираз – це s-вираз, який зв`язує ідентифікатор у вигляді Lisp-символа з певним значенням. Наприклад defun зв`язує конкретний Lisp-символ із функцією, яку описує тіло defun;

EBNF – extended Backus-Naur form;

GUI – graphical user interface;

JSON – JavaScript object notation;

JSON-об’єкт – це тип даних JSON, що є послідовністю типу ключ-значення, де ключ та значення відділені між собою “:”. Ключем може бути тільки рядок, а значенням може бути будь-який тип даних JSON. Об'єкт починається з “{” і закінчується “}”;

LCS – longest common subsequence;

Lisp-символ – це лексема, яка складається із довільної кількості дозволених для Lisp-символа символів, який є ідентифікатором певного зв’язаного із ним об’єкту мови LISP;

Lisp-список – це s-вираз, запис якого починається з “(” і закінчується “)”, всередині якого можуть знаходитись 0 та більше s-виразів;

q-s-вираз – це вираз, до якого було застосовано символ ‘;

VCS – version control system.

ВСТУП

Розробники програмного забезпечення у своїй професійній діяльності при розробленні програмних засобів часто використовують різноманітні допоміжні засоби для покращення продуктивності, зменшення кількості помилок, проведення аналізу тощо. До таких допоміжних засобів можна віднести засоби підсвітки синтаксису та автодоповнення в текстових редакторах, системи контролю версій та засоби порівняння чи об’єднання текстів тощо. Серед перелічених допоміжних засобів, а саме для засобів порівняння текстів на сьогоднішній момент склалася цікава ситуація, суть якої в тому, що вони не дають для розробників багато корисної інформації при використанні їх в аналізі різних версій програм, що часто відбувається при процесі перегляду коду, написаного іншими членами команди.

Хоча й існує певна кількість засобів, які вміють проводити не просто текстове, а основане саме на аналізі програмних структур (функції, класи, методи тощо) та їхньої семантики порівняння програмних текстів, але ці засоби працюють лише для певної кількості найбільш популярних на цей час С-подібних мов програмування. Однак зараз в індустрії розроблення програмного забезпечення йде популяризація використання функціональних мов програмування, до числа яких входять мови сімейства Lisp. Взагалі мови з Lisp-подібним синтаксисом працюють не дуже гарно зі звичайними засобами порівняння тексту, але для цих мов, на жаль, не існує засобу, що проводив би порівняння на структурному рівні програмних текстів, що написані з їхнім використанням.

З огляду на вищеописаний факт розробка засобу порівняння різних версій програм на мові LISP з використанням абстрактного семантичного дерева є актуальною. Під мовою LISP у цьому проєкті розуміється підмножина мови Common Lisp, що є найбільш популярним діалектом сімейства мов Lisp.

1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

1.1 Аналіз засобів порівняння тексту

Засоби порівняння тексту – це засоби, які використовуються для порівняння та відображення різниці й подібності між вмістом текстових файлів. Ці засоби являють собою підвид засобів порівняння файлів. Вони можуть бути оформлені як самостійні програмні засоби або бути вбудованими в текстові редактори, текстові процесори чи в інтегровані середовища розробок [1].

Прикладами самостійних засобів порівняння тексту є diff, diff3, Meld, WinDiff, WinMerge, Kompare, Pretty Diff тощо [2].

Прикладами програмних засобів із вбудованим засобом порівняння тексту є vim, Emacs, Notepad++, IntelliJ IDEA, Total Commander тощо [2].

Основний метод, який здебільшого використовують засоби порівняння тексту є метод пошуку LCS (longest common subsequence) між двома текстовими файлами. Класично текстові фрагменти, які не увійшли до LCS помічаються як «видалені» або «додані». Сам алгоритм пошуку LCS може працювати на рівні окремих слів, що зазвичай реалізовано в текстових процесорах, на рівні окремих рядків, що реалізовано в орієнтованих на програмні тексти засобах, або на рівні окремих символів, що є корисним для спеціалізованих програмних засобів [1].

На практиці розробники програмного забезпечення передусім використовують саме засоби порівняння тексту, які орієнтовані на роботу з програмними текстами. Оскільки засоби порівняння тексту цього типу зазвичай мають однакові основні можливості, то розглянемо їх на прикладі Meld.

Meld – це засіб візуального порівняння та об’єднання текстів програм, що орієнтований на використання розробниками. Цей програмний засіб є

Добавлено примечание ([OM1]): Не треба використовувати «порівнювач» та «поєднувач», бо:
1.Це жаргонізми
2.А головне у Вас в темі зазначено «Засіб порівняння», тому усюди в записці повинно бути саме «засіб порівняння».

Добавлено примечание ([OM2]): Тут точно Kompare, а не Compare?

Добавлено примечание ([D3R2]): Так тут саме Kompare, бо це утиліта з проєкту KDE https://ru.wikipedia.org/wiki/Kompare

безкоштовним та має відкритий програмний код. Графічний інтерфейс Meld зображено на рис. 1.1 [3].

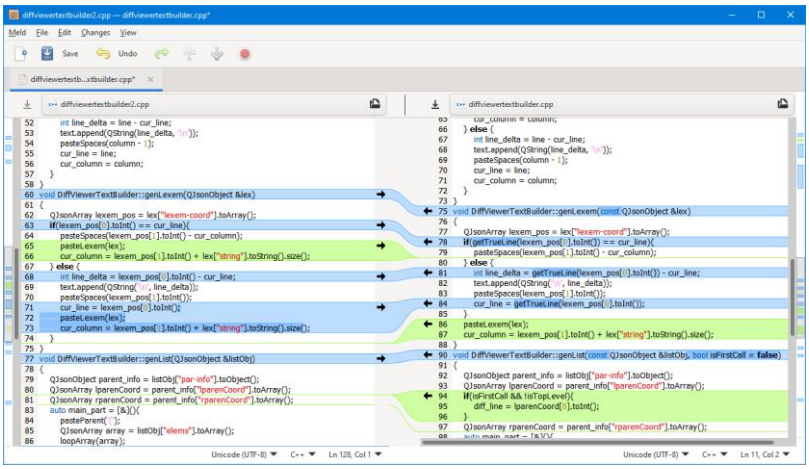

Рис 1.1 – Графічний інтерфейс Meld

Meld має такі особливості:

- підсвітка синтаксису;
- вбудоване підтримання використання регулярних виразів для фільтрації тексту, який треба ігнорувати, що корисно для виключення з порівняння коментарів;
- можливість редагувати файли з миттєвим оновленням результатів порівняння;
- можливість порівняння директорій;
- підтримання популярних VCS (version control system), як-от Git, Mercurial, Bazaar та SVN

Meld та подібні йому програмні засоби мають такі основні переваги:

- універсальність, яка означає в цьому контексті, що вони можуть порівнювати будь-які текстові файли між собою;
- зручне візуальне представлення різниці;

Добавлено примечание ([ОМ4]): При переліченні в кінці ставляться крапки з комою, а в кінці – крапка.

- демонструють абсолютно всі зміни в рядках, за винятком рядків, що ігноруються.

Але вказані переваги легко перетворюються в недоліки, коли розробник програмних засобів хоче побачити та проаналізувати семантичні зміни, що відбулися в нових версіях текстів програми, над якими іде робота.

Семантичні зміни – це такі зміни в тексті програмного засобу, що впливають саме на логіку цього програмного засобу. Сюди входять додавання, видалення чи модифікація функцій, класів, інтерфейсів тощо.

Звичайні засоби порівняння тексту, навіть з орієнтацією на тексти програмних засобів, через те, що використовують алгоритм пошуку LCS між текстами програмних засобів різних версій одного файлу, не розуміють ні синтаксис, ні семантику тієї мови програмування, на якій був написаний текст програмного засобу, а це призводить до низки таких проблем:

- якщо клас, функція або якійсь інший програмний фрагмент був переміщений у межах цього файлу та потім модифікований всередині, то є велика ймовірність того, що, наприклад, Meld буде в старій версії файлу демонструвати факт видалення цього фрагменту, а в новій версії виділяти його як новий фрагмент, що із семантичного погляду взагалі некоректно;
- якщо клас, функція або якійсь інший програмний фрагмент був переміщений до іншого файлу в проєкті, то в тому файлі, з якого було переміщено фрагмент, він буде демонструватися як видалений, а у файлі, до якого було переміщено його, як новий фрагмент;
- якщо було змінено форматування коду, то через те, що порівнюються не окремі слова тексту програм, а порівняння працює по порядку, то виходить, що в рядках, де по-суті були просто додані чи видалені пробільні символи, будуть відображатися як видозмінені, що, наприклад, для С-подібних мов програмування або для діалектів Lisp не несе нового семантичного

змісту, хоча для Python, Haskell та інших мов програмування, логіка яких залежить від пробільних відступів, це вже може нести новий семантичний зміст.

Ці проблеми створюють інформаційний шум для розробника, який, наприклад, має проаналізувати саме семантичні зміни в тексті програмного засобу, які були внесені іншим розробником у команді, і на основі аналізу схвалити зміни чи ні. Цей інформаційний шум може призвести до погіршення продуктивності, а в гіршому випадку ще й додатково до некоректних висновків щодо проаналізованих фрагментів.

Тобто виходить, що звичайні засоби порівняння тексту не підходять для аналізу семантичних змін у текстах програмних засобів.

1.2 Аналіз засобу семантичного порівняння та поєднання текстів програм Semantic Merge

Добавлено примечание ([ОМ5]): Ну а це буде засіб порівняння та поєднання

Є засіб семантичного порівняння та поєднання текстів програм Semantic Merge, що розроблений компанією Codice Software.

Semantic Merge розроблений передусім як засіб, який допомагає покращити процес, який виникає в разі спроби поєднати файли з програмними текстами різних версій одного проєкту, а це відбувається часто в сучасних умовах розроблення програмного забезпечення, коли зазвичай обов'язково іде використанням VCS. Для цього цей засіб використовує семантичне розуміння вмісту файлів різних версій проєкту, які порівнюються. Звичайно, для того, щоб у цьому засобі була можливість працювати такому методу поєднання тут реалізована окрема частина, що відповідає за семантичне порівняння. Semanitc Merge працює лише в режимі графічного інтерфейсу, приклад якого зображено на рис 1.2 [4]:

Має низку таких особливостей:

- повний семантичний режим підтримується лише для мов програмування C#, Java, C, C++ та PHP. Для інших мов

програмування цей засіб пропонує лише текстове порівняння та поєднання, але яке працює набагато краще за звичайні засоби, що були розглянуті на прикладі Meld в підрозділі 1.1, за допомогою використання технологій Xdiff та Xmerge [4];

- може розпізнавати факт переміщення функцій, класів та інших структурних елементів відповідних мов програмування не тільки всередині одного файлу, а й між файлами. Це можливо, бо порівняння відбувається на основі «method-by-mehod» [4];
- має режим візуального поєднання та порівняння. Приклад цього зображено на рис. 1.3;
- може розуміти специфічні конструкції підтримуваних мов програмування, а також у разі проблем із лексичним, синтаксичним чи семантичним аналізом показувати повний звіт із докладним описом проблем та їх місцеположенням і також у цьому разі пропонується перейти до звичайного текстового поєднання чи порівняння;
- має підтримку для плагінів.

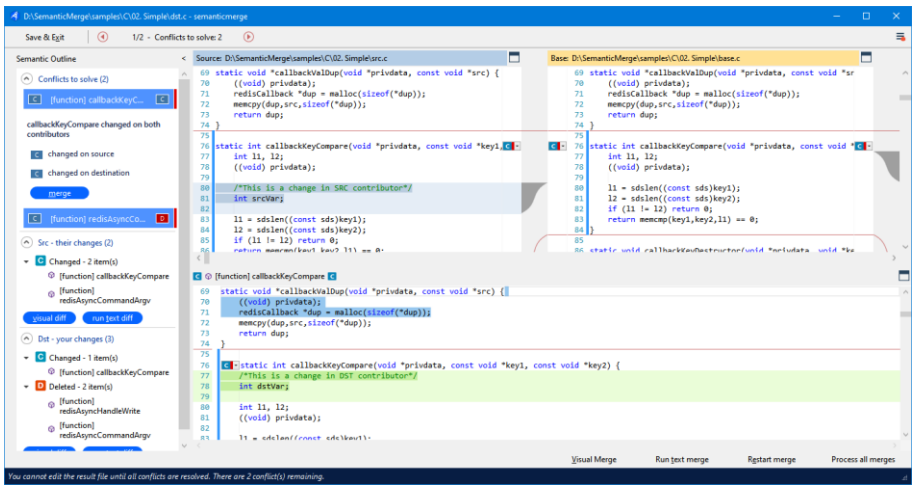

Рис 1.2 – Графічний інтерфейс Semantic Merge

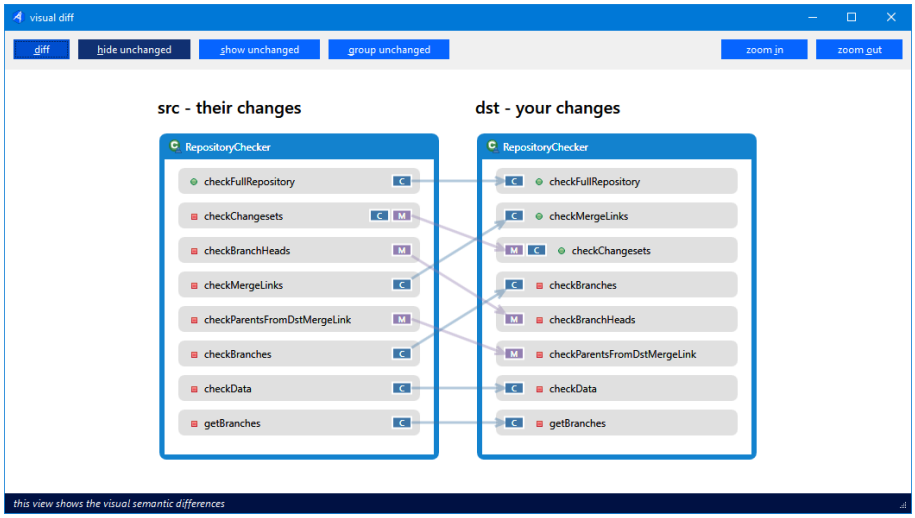

Рис 1.3 – Приклад візуального порівняння в Semantic Merge

Semantic Merge має такі недоліки:

- мала на цей час кількість повністю підтримуваних мов програмування;
- цей програмний засіб є платним. Безкоштовно доступна лише 30-денна демоверсія;
- має закритий програмний код.

1.3 Обґрунтування теми дипломного проєкту та постановка задачі

Засоби, які розглянуті у підрозділах 1.1 та 1.2, не можуть працювати в режимі структурного порівняння програмних текстів, створених з використанням мов програмування, що мають Lisp-подібний синтаксис.

З огляду на вищеописаний факт розробка засобу порівняння різних версій програм на мові LISP з використанням абстрактного семантичного дерева є актуальною. Під мовою LISP розуміється певна підмножина Common

Lisp, найбільш популярного діалекту Lisp. У підрозділах 2.2, 2.3 та 2.4 надано докладний опис підмножини Common Lisp, яка має підтримуватись у проєкті. Основна причина виділення саме певної підмножини пов'язана з фізичною неможливістю реалізувати за час, що виділений на виконання бакалаврського дипломного проєкту, підтримання всіх можливостей Common Lisp, що описані в Common Lisp HyperSpec [5].

Розроблений засіб має вирішувати таку низку задач:

- підтримувати в аналізі всі конструкції мови LISP, що описані в підрозділах 2.2, 2.3 та 2.4;
- виконувати структурне порівняння 2-х поданих до засобу файлів, що написані на мові LISP;
- вміти ігнорувати в процесі порівняння коментарі;
- вміти повідомляти користувача про лексичні чи синтаксичні помилки в програмних текстах із вказанням їхнього місцеположення;
- вміти класифікувати top-level s-вирази;
- вміти коректно підсвічувати класифіковані певним чином s-вирази всередині переглядача результатів порівняння.

2. РОЗРОБКА ЗАСОБУ ПОРІВНЯННЯ ВЕРСІЙ ПРОГРАМ НА МОВІ LISP

2.1 Загальна архітектура засобу

Засіб порівняння версій програм на мові LISP складається з двох частин:

- backend частини;
- frontend частини.

Ці частини розроблені як окремі програми, на взаємодії яких працює засіб.

Backend частина займається порівнянням текстових файлів, написаних на мові LISP та складається з таких модулів:

- модуль лексичного аналізатора;
- модуль синтаксичного аналізатора;
- модуль перетворення дерева синтаксичного розбору в абстрактне семантичне дерево;
- модуль порівняння абстрактних семантичних дерев;
- модуль генерації вихідних JSON (JavaScript Object Notation) файлів.

Backend частина розроблена з використанням мови програмування Common Lisp та знаходиться в папці DiffBackend. Кожний модуль реалізований, як один або декілька файлів, у яких визначені відповідні пакети та реалізовані відповідні функції. Пакети в Common Lisp дозволяють створювати окремі простори імен, визначати, які функції, символи тощо експортувати з пакета, та які функції, символи тощо імпортувати в пакет з інших пакетів.

Всередині папки DiffBackend є такі файли та папки:

- файл diff-backend.asd, який використовується ASDF (Another System Defintion Facitlity) для завантаження необхідних

Добавлено примечание ([ОМ6]): Знову ж таки, Ви розроблюєте не програму, а засіб, то усюди повинен бути засіб

Добавлено примечание ([ОМ7]): В дипломі треба уникати слова «опис», тому, що ви описуєте процес розробки, а не просто готовий продукт, розроблений кимось.

залежностей та правильного завантаження й побудови програмних текстів backend частини;

- файл gen-diff-backend-exe.bat, що використовується для створення виконавчого файлу backend частини diff-backend.exe;
- папка src, де знаходяться файли з сирцевим кодом backend частини;
- папка tests, де знаходяться файли з юніт тестами. Докладно про розроблення юніт тестів надано в підрозділі 3.1;

Вміст папки DiffBackend/src:

- файл main.lisp, у якому визначений пакет :diff-backend та знаходиться функція main, з якої починається робота backend частини, та реалізовані функції, які об’єднують роботу функцій, що реалізовані в інших пакетах;
- файл utils.lisp, у якому визначений пакет :diff-backend/utils та знаходяться допоміжні функції;
- файл lexer.lisp, у якому визначений пакет :diff-backend/lexer та в якому реалізовані функції лексчиного аналізатора. Докладно про його розроблення надано в підрозділі 2.2;
- файл parser.lisp, у якому визначений пакет :diff-backend/parser та в якому реалізовані функції синтаксичного аналізатора. Докладно про його розроблення надано в підрозділі 2.3;
- файл abstract-sem-tree-generator.lisp, у якому визначений пакет :diff-backend/abstract-sem-tree-generator та в якому реалізовані функції перетворення дерева синтаксичного розбору в абстрактне семантичне дерево. Докладно про його розроблення надано в підрозділі 2.4;
- файл nodes.lisp, у якому визначений пакет :diff-backend/nodes та описана система класів, яка визначає вузли, з яких будується абстрактне семантичне дерево. Докладно про його розроблення надано в підрозділі 2.4;

- файл statistics.lisp, у якому визначений пакет :diff-backend/statistics та реалізовані функції для роботи зі статистикою по формам верхнього рівня. Докладно про його розроблення надано в підрозділі 2.4;
- файл comparator.lisp, у якому визначений пакет :diff-backend/comparator та реалізовані функції для порівняння абстрактних семантичних дерев. Докладно про його розроблення надано в підрозділі 2.5;
- файл results-generator.lisp, у якому визначений пакет :diff-backend/results-generator та реалізовані функції для створення вихідних JSON файлів.

Frontend частина займається наданням користувачу GUI (graphical user interface), через який користувач має взаємодіяти із засобом та де відображаються результати роботи backend частини.

Frontend частина розроблена з використанням мови програмування C++ та з використанням кросплатформеного фреймворку Qt5. Ця частина знаходиться в папці DiffFrontend та має такі файли:

- DiffFrontend.pro
- diffviewer.cpp;
- diffviewer.h;
- diffviewertext.cpp;
- diffviewertext.h;
- diffviewertextbuilder.cpp;
- diffviewertextbuilder.h;
- global.cpp;
- global.h;
- linenumberarea.cpp;
- linenumberarea.h;
- main.cpp;
- mainwindow.cpp;

- mainwindow.h;
- mainwindow.ui;
- stat.cpp;
- stat.h.

Докладно про розроблення frontend чатстини надано в підрозділі 2.7. Загальну структура засобу зображено на рис 2.1.

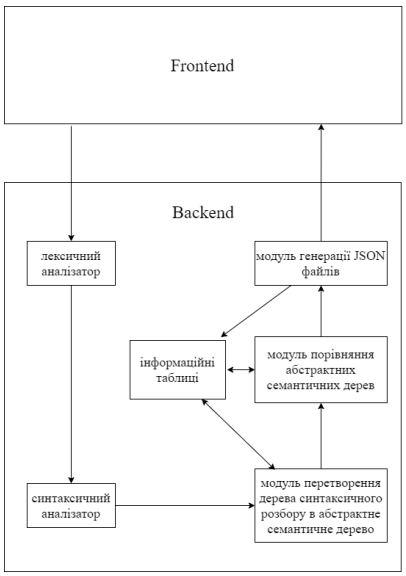

Рис. 2.1 – Загальна структура засобу

Засіб працює в разі відсутності помилок у такій послідовності:

1. Користувач у frontend частині обирає для порівняння 2 файли з різними версіями програми на мові LISP.
2. Frontend частина викликає backend частину, передаючи як аргументи командного рядка шляхи до обраних файлів на етапі 1.
3. У backend частині отримується вміст файлів як текстові рядки, що передаються на опрацювання модуля лексичного аналізатора.
4. Результати лексичного аналізатора передаються далі на опрацювання до модуля синтаксичного аналізатора.

Добавлено примечание ([D8]): Чи підійде цей рисунок для креслення А4?

Добавлено примечание ([OM9R8]): Ви мабуть, хотіли сказати формату А1.

Блоків малувато як для А1. Було б непогано продеталізувати, якщо є чим.
Але якщо кращих схем важко придумати, то що тоді поробиш.

Добавлено примечание ([D10]): Чи потрібно вказати, що така послідовність коректна для безпомилкового випадку ?

Добавлено примечание ([OM11R10]): Так, вказати буде краще.

5. Результати синтаксичного аналізатора далі передаються до модуля перетворення дерева синтаксичного розбору в абстрактне семантичне дерево.
6. Результати 5-го етапу надаються до модуля порівняння абстрактних семантичних дерев.
7. Модуль генерації вихідних JSON файлів створює відповідні JSON файли для результатів роботи backend частини.
8. Frontend частина читає JSON файли отримані на етапі 7 й виконує візуалізацію результатів backend частини.

Розроблена архітектура засобу має такі переваги:

- чітке розділення на частину, яка виконує основну логіку засобу, та частину, яка займається наданням зручного інтерефейсу взаємодії та візуалізацією результатів, що полегшує розробку;
- з’являється можливість написати будь-який інший фронтенд, наприклад, вебінтерфейс чи мобільний інтерфейс і водночас взагалі без виконання змін або із внесенням мінімальних змін до backend частини, що можливе через те, що Common Lisp може працювати на цей час практично на всіх популярних платформах;
- таку архітектуру дуже легко перетворити на клієнт-серверну архітектуру, перетворивши backend частину на сервер та перетворивши frontend частину на клієнт.

Проте така архітектура має й такі недоліки:

- збільшується загальний розмір, який буде займати готовий до використання засіб на пристрої користовача, що пов’язано з тим, що кожна частина буде навантажена своїми залежностями.
- збільшується складність відлагодження засобу.

2.2 Розробка лексичного аналізатора

Лексичний аналізатор – це етап роботи backend частини засобу, на вхід якого подається текстовий рядок, а на виході отримується список лексем, отриманий у результаті лексичного аналізу заданого текстового рядка.

Лексичний аналіз розбирає вхідний потік символів на лексеми, які є найменшою значущою одиницею в програмному тексті. Також під час аналізу виявляються лексичні помилки.

У проєкті допустимі такі лексеми:

- Роздільні символи: “(“, “)” та “””;
- Цілі числа;
- Lisp-символи;
- Символьні рядки.

Також розпізнаються та записуються в окрему таблицю рядкові коментарі. Таблиця з коментарями не бере участь в інших етапах backend частини, крім етапу генерації вихідних JSON файлів, де для цієї таблиці генерується свій JSON файл, який буде необхідний frontend частині для правильного візуального відтворення тексту файлу.

Лексичний аналіз реалізується з використанням скінченного автомату.

Позначення станів автомату лексичного аналізатора:

- START – початковий стан;
- INPUT – стан введення поточного символу;
- INTEGER – стан виділення цілого числа;
- SYMBOL – стан виділення “символу”;
- STRING – стан виділення символьного рядка;
- STRING_ESCAPE_SYM – стан розпізнавання символу, що екранується в символьному рядку;
- DELIMETER – стан виділення роздільного символа;
- COMMENT – стан виділення рядкового коментаря;
- OUT – стан виведення в рядок лексем виділеної лексеми;

Добавлено примечание ([OM12]): В дипломі відбувається НЕ опис, а **розробка**. Відповідно повинна бути витримана саме така термінологія усюди в записці

Добавлено примечание ([D13]): Як мені в цьому підрозділі та й далі не мати плутанини між символами Lisp та рядковими символами?

Добавлено примечание ([OM14R13]): Думаю, найкраще дати визначення на початку диплому при визначенні термінів.
Lisp-символ - це ….

І потім використовувати такий термін.

- OUT_COMMENT – стан виведення розпізнаного рядкового коментаря до таблиці коментарів;
- WHITESPACE – стан розпізнавання й ігнорування пробільних символів;
- ERR – стан виділення помилкової лексеми;
- ERR_OUT – виведення;
- EXIT – кінцевий стан.

Позначення вхідних символів:

- dg – [0-9];
- lt – [A-Za-z];
- sign – [+-];
- spec-sym – [+-*?!$^&@%<>~.=_];
- dm – [()’];
- nl – символ нового рядка;
- wh – пробільний символ;
- eof – символ кінця файлу.

Відповідна діаграма переходів автомату лексичного аналізатора зображена на рис. 2.2.

Модуль лексичного аналізатора реалізовано у файлі DiffBackend/src/lexer.lisp у пакеті :diffbackend/lexer. Найголовніша функція в ньому це lexer, де реалізовано алгоритм лексичного автомату. Ця функція приймає на вхід текстовий рядок, а повертає 3 значення:

- список виділених лексем;
- хеш-таблицю з коментарями, де ключ – це номер рядка, на якому був розпізнаний рядковий коментар, а значення – це спискова структура, де записаний сам текстовий рядок із коментарем та його початкова колонка;
- список лексичних помилок, який пустий, якщо немає лексичних помилок.

Інші функції в цьому пакеті є допоміжними для функції lexer.

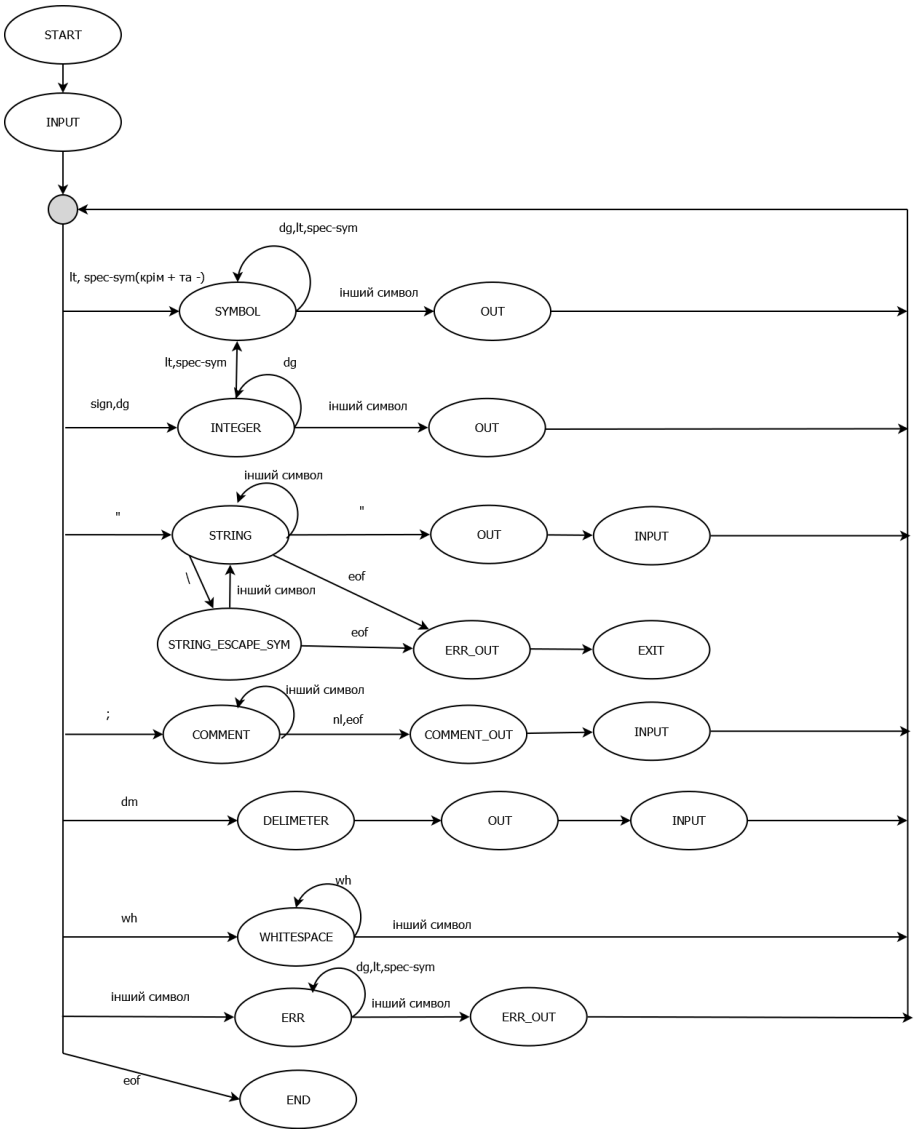

Рис. 2.2 – Діаграма переходів автомату лексичного аналізатора

Лексеми зберігаються в об’єктах класу lexem, який має такі поля:

- id – поле де зберігається унікальний ідентифікатор для лексеми. Це поле використовується для ідентифікації помилкової лексеми, що дозволяє виділяти її в frontend частині;

- line – це поле зберігає номер рядка, де знаходиться лексема. Рядки рахуються у функції lexer з 1;
- column – це поле зберігає номер колонки, на якій починається лексема. Колонки рахуються у функції lexer з 1;
- type – це поле зберігає тип лексеми як символ з пакету keyword, що дає змогу не мати проблем із перевіркою значення цього поля в інших пакетах. Може приймати такий список значень: :integer, :symbol, :left-parent, :right-parent, :quote, :string та :error-lexem;
- string – це поле зберігає текстовий рядок, де записана лексема.

У лексичному аналізаторі розпізнається 2 види лексичної помилки:

- некоректна лексема, що починається з недозволенного символу. Наприклад, :123 , #1#, |33| будуть розпізнані як некоректні лексеми;
- незакінчений символьний рядок.

Факт лексичної помилки записується в об’єкт класу lexem-error, який має такі поля:

- error-text – це поле, куди записується текстове повідомлення про лексичну помилку. У повідомленні обов’язково вказується координата помилкової лексеми. Це повідомлення відображається користувачу в frontend частині;
- error-lex-id – ідентифікатор помилкової лексеми в списку лексем. Ця інформація використовується в frontend частині, якщо користувач хоче виділити конкретну помилкову лексему.

2.3 Розробка синтаксичного аналізатора

Синтаксичний аналізатор – це етап роботи backend частини засобу, який отримує на вхід список лексем та генерує дерево синтаксичного розбору.

Граматика мови LISP дуже проста та є контекстно-вільною й описується з використанням EBNF (extended Backus-Naur form) як:

<Top> ::= <S-expr>* ;

<S-expr> ::= <Atom> | <List> | <Quote-s-expr> ;

<Atom> ::= integer | symbol | string;

<List> ::= ‘(’ <S-expr>* ‘)’ ;

<Quote-s-expr> ::= ‘’’ <S-expr>.

Нетермінальні символи граматики:

- <Top>;
- <S-expr>;
- <Atom>;
- <List>;
- <Quote-s-expr>.

Термінальні символи граматики:

- integer;
- symbol;
- string.

Модуль синтаксичного аналізатора реалізовано у файлі DiffBackend/src/parser.lisp у пакеті :diffbackend/parser.

Функція parser є початковою точкою синтаксичного аналізу, який реалізований із використанням методу рекурсивного спуску. Кожному нетермінальному символу граматики поставлені у відповідність такі рекурсивні функції:

- <Top> - top-rule;
- <S-expr> - s-expr-rule;
- <Atom> - atom-rule;
- <List> - list-rule;
- <Qutoe-s-expr> - quote-s-expr-rule.

У цьому пакеті визначені такі глобальні змінні:

- *lexems-rest* - це змінна, яка зберігає список ще неопрацьованих лексем;
- *cur-lex* - це змінна, яка зберігає лексему, що зараз опрацьовується

Розглянемо докладно код функції parser, що зображений на рис. 2.3.

```
(defun parser (lexems)
  (let ((*lexems-rest* lexems))
    (catch 'parser-start
      (top-rule))))
```

Рис. 2.3 – Код функції parser

Ця функція приймає список лексем. Далі у функції відбувається така послідовність дій:

1. Глобальній змінній *lexems-rest* присвоюється переданий до функції список лексем lexems.
2. За допомогою спеціального оператора catch створюється точка повернення `parser-start, що необхідно для роботи механізму опрацювання синтаксичних помилок, що можуть виникнути при синтаксичному аналізі.
3. Викликається функція top-rule, що робить розбір граматичного правила <Top>.

Розглянемо докладно роботу функції top-rule, що зображена на рис. 2.4.

```
(defun top-rule ()
  (let (s-expr-l)
    (loop :while *lexems-rest*
          :do (push (s-expr-rule) s-expr-l))
    `(:top () ,@(reverse s-expr-l))))
```

Рис.2.4 – Код функції top-rule

Ця функція в циклі loop збирає в список результати виклику функції s-expr-rule, що необхідно для опрацювання правила <S-expr>*. Унаслідок

роботи цієї функції повертається список, який відіграє роль початку дерева синтаксичного розбору. Терми, що відповідають нетермінальним символам граматики є Lisp-списками виду “(type add-info element*)”, де:

- type – сюди записується тип нетермінального вузла як символ Common Lisp;
- add-info – сюди у вигляді списку атрибутів, записується додаткова інформація, що має зберігатися у цьому вузлі;
- element* - це 0 або більше термінальних чи нетермінальних вузлів.

Мінімальне дерево синтаксичного розбору для цього проєкту має вигляд “(:top nil)”. Термінальні вузли є об'єктами класу lexem.

Розглянемо також роботу функції s-expr-rule, що зображена на рис. 2.5.

```
(defun s-expr-rule ()
  (ecase (lexem-type (next-lexem))
    ((:integer :symbol :string)
     (atom-rule *cur-lex*))
    ((:left-parent)
     (list-rule *cur-lex*))
    ((:quote)
     (quote-s-expr-rule *cur-lex*))
    ((:right-parent)
     (throw-error "unmatched close parenthesis" *cur-lex*))))
```

Рис. 2.5 – Код функції s-expr-rule

У цій функції відбуваються такі дії:

1. Відбувається отримання наступної лексеми через використання функції next-lexem, й отримується тип лексеми для аналізу.
2. Залежно від типу лексему далі можуть почати роботу функції atom-rule, list-rule або quote-s-expr-rule, але якщо тип лексеми :right-parent то буде викликана функція throw-error з певними аргументами для припинення роботи синтаксичного аналізатора й генерації інформації про синтаксичну помилку.

Можна побачити, що з цієї функції ми отримуємо вихідне значення, що формується в інших функціях обробки нетермінальних правил. Якщо робити

все строго, то потрібно було б як результат роботи цієї функції повертати Lisp-список виду “(:s-expr nil inner-term)” (inner-term – це терм, що інкапсульований всередину цього Lisp-спику), але цього не робиться, бо з огляду на описані правила граматики такий нетермінальний вузол буде лише обгорткою для нетермінальних вузлів типу :atom, :list та :quote, що буде надлишковим, збільшуватиме розмір вихідного дерева синтаксичного розбору та ускладнювало б подальшу роботу з ним.

Розглянемо докладно дуже коротку функцію atom-rule, що зображена на рис. 2.6.

```
(defun atom-rule (lex)
  `(:atom () ,lex))
```

Рис 2.6 – Код функції atom-rule

Ця функція формує нетермінальний вузол типу :atom й вбудовує у нього об’єкт типу lexem, що виконує роль термінального вузла.

Функції quote-s-expr-rule та list-rule подібні до інших функцій обробки нетермінальних правил граматики, а тому немає сенсу їх докладно розглядати, крім фрагмента формування вузла у функції list-rule, що зображений на рис. 2.7.

```
`(:list ((:lparen-coord ,(lexem-line left-parent-lexem)
                        ,(lexem-column left-parent-lexem))
         (:rparen-coord ,(lexem-line lex)
                        ,(lexem-column lex)))
        ,@(reverse s-expr-l))
```

Рис. 2.7. – Фрагмент функції list-rule

У цьому фрагменті можна побачити, що до нетермінального вузла типу :list записується додаткова інформація про координати лівої та правої круглої дужки. Ця інформація зазвичай не записується синтаксичними аналізаторами, але через необхідність того, що frontend частина має потім за

згенерованими backend частиною JSON файлами відтворити правильно текст файлу, то відбувається її зберігання.

Розглянемо ще докладно функцію throw-error, що зображена на рис. 2.8.

```
(defun throw-error (error-text error-lex)
  (throw 'parser-start
       (values nil
               (make-instance
                'parser-error-info
                :error-text
                (format nil "At (~S:~S) ~A"
                        (lexem-line error-lex)
                        (lexem-column error-lex)
                        error-text)
                :error-lex-id (id error-lex)))))
```

Рис. 2.8 – Код функції throw-error

Функція throw-error отримує такі вхідні аргументи:

- error-text – це текст, що описує причину синтаксичної помилки;
- error-lex – це об’єкт класу lexem, з яким пов’язана синтаксична помилка.

У функції throw-error формується об’єкт класу parser-error-info, який по своїй структурі аналогічний класу lexem-error. До полів об’єкту класу parser-error-info заноситься така інформація:

- у поле error-text формується текст, де записуються координати лексеми error-lex та опис синтаксичної помилки, що береться зі змінної error-text;
- у поле error-lex-id записується ідентифікатор лексеми error-lex.

Після цього спеціальна форма throw перериває нормальний хід роботи синтаксичного аналізатора й передає в позицію тега `parser-start, що встановлений був спеціальною формою catch у функції parser, 2 значення через використання функції values (функції Common Lisp можуть повертати багато значень), що потім із тої позиції будуть повернуті як результат роботи синтаксичного аналізатора. Всередині values прописані такі значення:

- nil, що означає факт наявності синтаксичної помилки, оскільки при безпомилковому варіанті з модуля синтаксичного аналізу повертається в цій позиції синтаксичне дерево розбору;
- об’єкт класу parser-error-info, формування якого було описано вище.

Синтаксичний аналізатор у цьому проєкті опрацьовує такі синтаксичні помилки:

- незакрита ліва кругла дужка;
- права кругла дужка без відповідної їй лівій круглій дужці;
- за знаком ` немає s-виразу.

2.4 Генерація абстрактного семантичного дерева з дерева синтаксичного розбору

Добавлено примечание ([ОМ19]): Я підозрюю, що тут у Вас буде не абстрактне синтаксичне дерево, а дерево синтаксичного розбору

Модуль генерації абстрактного семантичного дерева з дерева синтаксичного розбору – це модуль, що виконує семантичний аналіз дерева синтаксичного розбору з кореня до його термінальних вузлів, під час якого будується абстрактне семантичне дерево.

Семантичний аналіз у межах цього дипломного проєкту – це визначення семантичного призначення, або іншими словами, сенсу s-виразів, які записані в усіх нетермінальних вузлах дерева синтаксичного розбору, крім кореня цього дерева. Також відбувається виявлення семантичних помилок, які заважають створенню коректного абстрактного семантичного дерева. Оскільки цей дипломний проєкт не пов’язаний зі статичним аналізом програмного тексту на мові LISP, то такого виявлення семантичних помилок є достатнім у межах цього дипломного проєкту.

Абстрактне семантичне дерево в межах цього дипломного проєкту – це дерево, корінь якого є Lisp-списком, а всі інші вузли є CLOS об’єктами, які описують семантику відповідних s-виразів та утворюють так звану об’єктну систему.

У пакеті :diff-backend/nodes, що знаходиться у файлі DiffBackend/src/nodes.lisp розроблена ієрархія CLOS класів об’єктної системи вузлів абстрактного семантичного дерева. UML діаграма ієрархії класів цієї системи зображена на рис. 2.9.

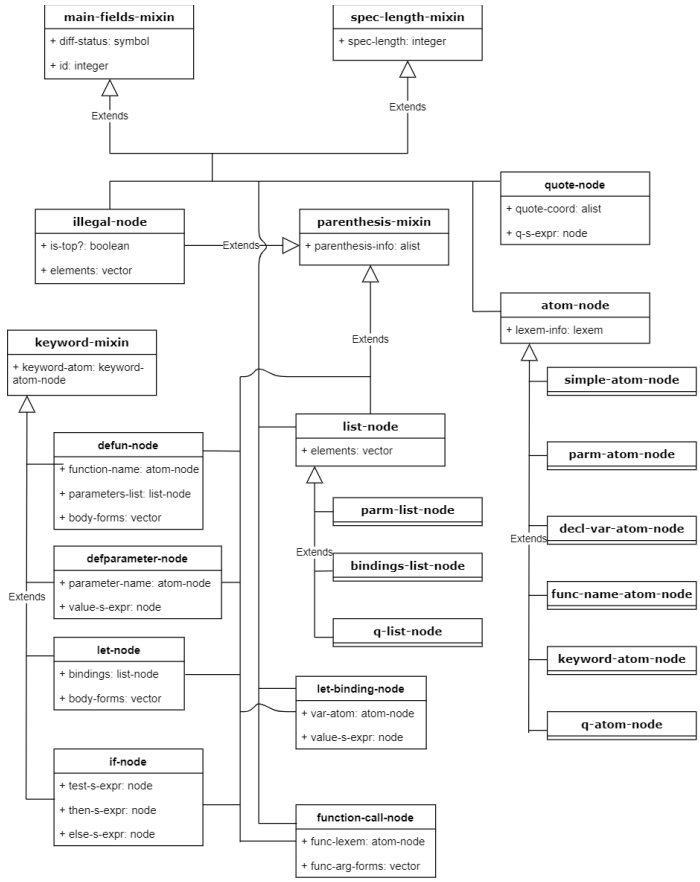

Рис. 2.9 – UML діаграма ієрархії класів об’єктної системи вузлів абстрактного семантичного дерева

Далі розглянемо докладно про суть кожного класу цієї системи разом із семантичною множиною мови LISP.

У системі є так звані класи “міксини”, суть яких у тому, що вони реалізують поля, які мають бути у 2-х або більше класів і до яких вони додаються через наслідування їхнього класу. CLOS дає змогу класам мати множинне успадкування, що дає можливість застосовувати багато класів “міксин” для формування полів конкретного класу.

У системі розроблені такі класи “міксини”:

- main-fields-mixin – реалізує поля, який мають бути в кожному класі, який реалізує конкретний вузол:
  - diff-status – має зберігати інформацію про певний статус вузла на етапі порівняння. За замовчуванням має значення :same;
  - id – має зберігати число, що є унікальним ідентифікатором вузла;
- spec-length-mixin - реалізує лише одне поле spec-length, що зберігає довжину, яку б займав би рядок s-виразу, який описується вулзом, якби там була лише необхідна кількість пробільних символів (наприклад, якщо у файлі s-вираз записаний як “(fun (fun2 arg1) arg2)”, то до spec-length запишеться довжина, яку б займав би рядок “(fun (fun2 arg1) arg2)”). Ця довжина необхідна для реалізації оптимізації алгоритму порівняння абстрактних семантичних дерев. Для поля spec-length розроблено автоматичний розрахунок його значення під час створення об`єкта конкретного вузла;
- parenthesis-mixin – реалізує лише одне поле parenthesis-info, що зберігає інформацію про круглі дужки. Ця міксина додається до класів вузлів, де s-вираз є Lisp-списком;
- keyword-mixin – реалізує лише одне поле keyword-atom, що зберігає об’єкт класу keyword-atom-node. Ця міксина додається до класів вузлів, де всередині s-виразу, який є Lisp-списком, перший атом є ключовим словом.

У системі розроблені 2 суперкласи, що реалізують загальний набір полів для певної кількості синтаксично однакових, але семантично різних класів вузлів. Це такі суперкласи як:

- atom-node – має своє поле lexems-info, куди має записуватись об’єкт класу Lexem. Суперклас atom-node описує синтаксично s-вираз, який є атомом. Від нього успадковуються такі класи:
  - simple-atom-node – реалізує семантично звичайний атом, який обчислюється в значення самого себе;
  - parm-atom-node – реалізує семантично атом, що є формальним параметром функції тощо;
  - decl-var-atom-node – реалізує семантично атом, що є Lisp-символом, до якого присвоюється певне значення;
  - func-name-atom-node – реалізує семантично атом, що є Lisp-символом, з яким пов’язана певна функція;
  - keyword-atom-node – реалізує семантично атом, що є Lisp-символом, який є ключовим словом;
  - q-atom-node – реалізує семантично атом, який є q-s-виразом чи знаходиться всередині q-s-виразу;
- list-node – має своє поле elements, куди має записуватися вектор вузлів. Синтаксично цей суперклас реалізує Lisp-список. Від нього успадковуються такі класи:
  - parm-list-node – реалізує семантично Lisp-список, що описує список формальних параметрів функції тощо;
  - bindings-list-node – реалізує семантично Lisp-список, що описує список зв’язувань Lisp-символів із певним значенням;
  - q-list-node – реалізує семантично Lisp-список, що є q-s-виразом, чи знаходиться всередині q-s-виразу.

Класи, які реалізують вузли, що мають ключове слово, такі:

- defun-node – реалізує семантично def-s-вираз, де ключове слово defun. Цей s-вираз визначає функцію. Його структура описується таким Lisp-списком “(defun func-name parameters-list body-form+)”, де:
    - defun – це ключове слово, що має бути реалізований відповідним Lisp-символом. Для нього створюється вузол класу keyword-atom-node, який записується в поле keyword-atom;
    - func-name – це Lisp-символ, що визначає назву функції. Для нього створюється вузол класу func-name-atom-node, який записується в поле function-name;
    - parameters-list – це список формальних параметрів функції. Для нього створюється вузол класу parm-list-node, який записується в поле parameters-list;
    - body-form+ - це 1 або більше s-виразів, що реалізують тіло функції. Ці s-вирази збираються у вектор та записуються в поле body-forms;
- defparameter-node – реалізує семантично def-s-вираз, де ключове слово defparameter. Цей s-вираз, визначає глобальну змінну. Його структура описується таким Lisp-списокм “(defparameter var-name value-s-expr)”, де:
    - defparameter – це ключове слово, що має бути реалізований відповідним Lisp-символом. Для нього створюється вузол класу keyword-atom-node, який записується в поле keyword-atom;
    - var-name – це Lisp-символ, що визначає назву глобального параметру. Для нього створюється вузол класу decl-var-atom-node, який записується в поле parameter-name;

- value-s-expr – це s-вираз, значення якого присвоюється параметру. Для нього створюється відповідний вузол, що записується в поле value-s-expr;

- let-node – реалізує семантично s-вираз, де ключове слово let. Цей s-вираз реалізує створення локальних змінних та виконання серії s-виразів всередині. Його структура описується таким Lisp-списком “(let bindings body-form+)”, де:
  - let – це ключове слово, з яким виконується тіж самі дії, що в попередніх класах;
  - bindings – це список зв'язувань. Для нього створюється вузол класу bindings-list-node;
  - body-form+ - має той самий сенс, що і для defun;
- if-node – реалізує семантично s-вираз, де ключове слово. Цей s-вираз реалізує класичний if. Його структура описується таким Lisp-списком “(if test-s-expr then-s-expr else-s-expr)”, де:
  - if – це ключове слово, з яким виконуються ті ж самі дії, що в попередніх класах;
  - test-s-expr – це s-вираз, що відіграє роль тесту. Для нього створюється відповідний вузол, що записується в поле test-s-expr;
  - then-s-expr – це s-вираз, що виконується, якщо test-s-expr повернув true. Для нього створюється відповідний вузол, що записується в поле then-s-expr;
  - else-s-expr – це s-вираз, що виконується, якщо test-s-expr повернув false. Для нього створюється відповідний вузол, що записується в поле else-s-expr.

Інші класи вузлів такі:

- function-call-node – реалізує семантично s-вираз, що є викликом функції. Його структура описується таким Lisp-списком “(fun-name arg-form*)”, де:

- fun-name – це Lisp-символ, що означає ім’я викликаємої функції. Для нього створюється вузол класу func-name-node та записується в поле func-lexem;
- arg-form* - це 0 або більше s-виразів, що семантично є аргументами функції, що викликається. Ці s-вирази збираються у вектор та записуються в поле func-arg-forms;

- let-binding-node – реалізує семантично Lisp-список, що реалізує зв’язування Lisp-символа з певним значенням. Його структура описується таким Lisp-списком “(var-name value-s-expr)”, де:
  - var-name – це Lisp-символ, до якого буде присвоєне певне значення. Для нього створюється вузол класу decl-var--atom-node, який записується в поле var-atom;
  - value-s-expr – це s-вираз, значення якого присвоюється var-name. Для нього створюється вузол, який записується до поля value-s-expr;
- quote-node – реалізує по суті обгортку для q-s-виразу, має такі поля:
  - quote-coord – записується координата символа ‘;
  - q-s-expr - записується вузол s-виразу;
- illegal-node – реалізує вузол для s-виразу, що семантично некоректний у відповідному контексті. Якщо створюється вузол цього класу – це означає, що було виявлено семантичну помилку. Цей вузол необхідний для коректного виділення семантичної помилки в frontend частині. Має такі поля:
  - is-top? – якщо в це поле записано false, то цей вузол знаходиться всередині іншого вузла класу illegal-node;
  - elements – записується вектор s-виразів, що були всередині.

Функції для генерації абстрактного семантичного дерева знаходяться в пакеті diff-backend/abstract-sem-tree-generator, що знаходиться у файлі DiffBackend/src/abstract-sem-tree-generator.lisp.

Всередині цього пакету визначені такі глобальні змінні:

- *current-file-ver* - може мати значення 1 та 2 та демонструє для якого вхідного файлу генерується абстрактне семантичне дерево;
- *current-id* - зберігає поточне максимальне значення ідентифікатора;
- *semantic-errors-list* - зберігає список семантичних помилок, що записуються в об’єкти класу semantic-error-info.

В пакеті реалізований клас semantic-error-info, що має такі поля:

- error-text – записується інформація про семантичну помилку;
- error-node – ідентифікатор вузла, що є об’єктом класу illegal-node.

Головна функція цього пакету, яка є початковою точкою для генерації абстрактного семантичного дерева -це abstract-sem-tree-gen, що приймає такі параметри:

- syn-tree – синтаксичне дерево розбору;
- cur-file – для якої версії вхідного файлу генерується абстрактне семантичне дерево.

Ця функція зображена на рис. 2.10.

```
(defun abstract-sem-tree-gen (syn-tree &key (curr-file 1))
  (let* ((*current-file-ver* curr-file)
         (*current-id* 0)
         (*semanitc-errors-list*)
         (res-ast (match-top syn-tree)))
    (values res-ast
            (nreverse *semanitc-errors-list*))))
```

Рис. 2.10 - Код функції abstract-sem-tree-gen

Генерація абстрактного семантичного дерева відбувається по суті відображенням вузлів дерева синтаксичного розбору у відповідні вузли абстрактного семантичного дерева.

Функції, що займаються розпізнаванням термів та генеруванням відповідних вузлів абстрактного семантичного дерева такі:

- match-top – розбирає корінь дерева синтаксичного розбору та формує Lisp-список, що відіграє роль кореня абстрактного семантичного дерева;
- match-s-expr – визначає тип терму, і якщо тип терму:
  - :atom, то викликається функція make-atom-node;
  - :quote, то викликається функція match-quote;
  - :list, то відбувається спроба визначення чи є 1 елемент списку ключовим словом і якщо так, то викликається відповідна функція (match-defun, match-let тощо), а якщо ні то викликається match-function-call;
- match-defun – зіставляє терм із патерном Lisp-списку defun та в разі успіху формує об’єкт класу defun-node;
- match-let – зіставляє терм із патерном Lisp-списку let та в разі успіху формує об’єкт класу let-node;
- match-function-call – зіставляє терм із патерном для виявлення виклику функції та в разі успіху формує об’єки класу function-call-node;
- тощо.

У разі неуспіху зіставлення формується об’єкт класу illegal-node та формується об’єкт класу semantic-errors-list з докладним повідомленням про семантичну помилку, що додається до списку *semantic-errors-list*.

Для розбору термів використовується зручний макрос Common Lisp destructuring-bind, що дає змогу розбирати Lisp-список за певним патерном.

Також при створенні вузлів для def-s-виразів, виконується функція add-to-stats з пакету :diff-backend/statistics, що знаходиться у файлі DiffBackend/src/statistics.lisp. Функції цього пакету працюють із хеш-таблицями *file-ver-1-stats* та *file-ver-2-stats*, що зберігають всередині себе інформацію про def-s-вирази. Інтерфейс функції add-to-stats зображений на рис. 2.11.

```
(defun add-to-stats (name obj &key stat-name file-ver)
```

Рис. 2.11 – Інтерфейс функції add-to-stats

Функція add-to-stats приймає такі параметри:

- name – це ім'я ідентифікатора def-s-вирази;
- obj – це посилання на об’єкт вузла, що описує def-s-вираз;
- stat-name – Lisp-символ, що вказує на тип def-s-виразу в множині (наприклад, :defuns);
- file-ver – вказує чи для 1-ї чи для 2-ї хеш-таблиці додавати.

Функція add-to-stats відповідно додає інформацію про певний виявлений def-s-вираз разом із obj в хеш-таблицю за ключем stat-name в *file-ver-1-stats* чи в *file-ver-2-stats*.

2.5 Розробка етапу порівняння абстрактних семантичних дерев

Модуль порівняння абстрактних семантичних дерев знаходиться в пакеті :diff-backend/comparator, що знаходиться у файлі DiffBackend/stc/comparator.lisp.

Головна функція цього пакету, яка є початковою точкою для алгоритму порівняння абстрактних семантичних дерев – це start-asts-comparing, що приймає такі параметри:

- ast-1 – абстрактне семантичне дерево першого вхідного файлу;
- ast-2 – абстрактне семантичне дерево другого вхідного файлу.

Схема алгоритму порівняння зображена на рис. 2.12.

Головна суть цього алгоритму – це “розфарбовування” вхідних абстрактних семантичних дерев через встановлення у вузлів певного значення поля :diff-status та заповнення даними таблиць *file-ver-1-stats* (на схемі позначено як def-ht-1) та *file-ver-2-stats* (на схемі позначено як def-ht-1) з пакету :diff-backend/statistics.

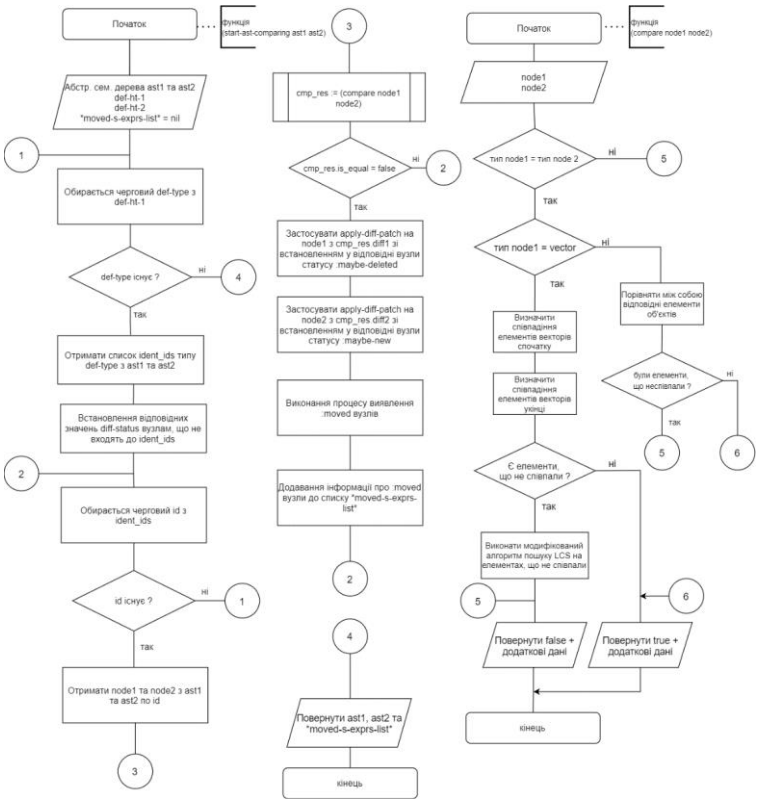

Рис. 2.12 – Алгоритм порівняння абстрактних семантичних дерев

В алгоритмі порівняння відбувається лише між вузлами піддерев, чиї корені є вузлами , які описують def-s-вирази одного типу (тобто є def-вузлами). Про це вказано в технічному завданні й це має сенс, бо вузли, що не є частинами def-вузлів, не можна струтурно порівняти між собою, бо саме def-s-вирази утворюють програмні структури.

Оскільки порівняння має відбуватися між def-вузлами одного типу, то обирається черговий def-type, що є ключем def-ht-1. З використанням def-type отримуємо з def-ht-1 та def-ht-2 списки ident1 та ident2 ідентифікаторів відповідних def-вузлів типу def-type. Із цих списків далі отримуємо з використанням функції get-identical-id-names список ident_ids, що складається з елементів їхнього пересікання. Def-вузли, чиї ідентифікатори з ident1 не

увійшли в ident_ids, отримають значення :diff-status :deleted, а до def-ht-1 буде записана інформація, що відповідний йому def-s-вираз був видалений у новій версії програми. Так само def-вузли, чиї ідентифікатори з ident2 не увійшли в ident_ids, отримають значення :diff-status :new, а до def-ht-2 буде записана інформація, що відповідний йому def-s-вираз був доданий у новій версії програми.

Далі починається порівняння def-вузлів з однаковими ідентифікаторами між собою. Для цього виконується виклкик до функції compare.

Взагалі compare є сімейством функцій, що визначене за допомогою defgeneric. Конкретні реалізації compare визначаються за допомогою defmethod. У Common Lisp виклик конкретної реалізації вибирається за допомогою техніки динамічного диспатчингу через аналіз аргументів, що передаються у функцію, яка визначена через defgeneric.

Визначення сімейства функцій compare зображено на рис. 2.13.

```
(defgeneric compare (obj1 obj2)
  (:method (obj1 obj2)
    (values nil 0 (when obj1 (id obj1)) (when obj2 (id obj2)))))
```

Рис 2.13 – Визначення сімейства функцій compare

Функції compare приймають 2 аргументи й залежно від їхнього типу буде викликана конкретна його реалізація, якщо вона є, в іншому ж разі викликається дефолтна реалізація, що прописана в defgeneric.

Усі реалізації compare повертають такі вихідні значення:

1. is-fully-equal – чи були об'єкти, що порівнювалися, повністю рівними між собою.
2. equal-leaf-counts – кількість рівних термінальних вузлів між двома об’єктами. Це необхідно для модифікованого алгоритму пошуку LCS, який буде розглянутий далі.

3. diff-patch1 – деревоподібна структура, що зберігає вузли, які треба позначити як :maybe-deleted та додати до списку *maybe-deleted-nodes*.
4. diff-patch2 – деревоподібна структура, що зберігає вузли, які треба позначити як :maybe-new та додати до списку *maybe-new-nodes*.

Функції compare спеціалізовані на порівнянні вузлів однакових типів та порівнянні векторів вузлів. Це дає змогу автоматично не вважати однаковими вузли, чиї s-вирази синтаксично однакові, але мають різний семантичний сенс.

Порівнюючи вузли одного типу відбувається порівняння їхніх відповідних елементів між собою. Розписувати реалізацію цього для всіх типів вузлів докладно не має сенсу.

Більш цікавим було реалізовано порівняння векторів вузлів між собою. Спочатку відбувається визначення однакових вузлів на початку та наприкінці векторів. Якщо ж було виявлено неоднаковість, то для цих “середніх вузлів” обох векторів, які позначимо як M1 та M2, викликається модифікований алгоритм пошуку LCS.

Реалізований модифікований алгоритм пошуку LCS подібний до алгоритма пошуку LCS між двома рядками. Цей алгоритм, як і його оригінал, складається з 2-х етапів:

1. Заповнення матриці LCS. Його псевдокод зображено на рис. 2.14.
2. Читання матриці LCS для отримання шуканої підпослідовності елементів. Його псевдокод зображено на рис. 2.15.

За допомогою застосування цього алгоритму отримуємо підпослідовність вузлів, що має найбільше сумарне значення однакових термінальних вузлів. Вузли, що не увійшли у цю підпослідовність додаються до відповідних diff-patch.

Після того як було отримано результат порівняння між відповідними def-вузлами та якщо було виявлено неповний збіг, то виконуються такі дії:

1. Виконується функція apply-diff-patch на цих вузлах.
2. Ці вузли помічаються у таблицях def-ht-1 та def-ht-2 як модифіковані

```
function getLCSarr(M1[1..m], M2[1..n])
    LCSarr = array(0..m, 0..n)
    for i := 0..m
        LCSarr[i,0] = 0
    for j := 0..n
        LCSarr[0,j] = 0
    for i := 1..m
        for j := 1..n
            cmp_res := compare(M1[i], M2[j])
            if cmp_res.equal-leaf-counts > 0
                LCSarr[i,j] := LCSarr[i-1,j-1] + cmp_res.equal-leaf-counts
            else
                LCSarr[i,j] := max(LCSarr[i,j-1], LCSarr[i-1,j])
    return LCSarr
```

Рис. 2.14 – Псевдокод функції getLCSarr

```
function backtrackLCSarr(LCSarr[0..m,0..n], i, j)
    if i = 0 or j = 0
        return;
    if (LCSarr[i-1,j] = LCSarr[i,j-1]) and (LCSarr[i,j] != LCSarr[i-1,j-1])
        addToResList(LCSarr[i,j])
        backtrackLCSarr(LCSarr,i-1, j-1)
    R := {}
    if LCSarr[i,j-1] >= LCSarr[i-1,j]
        backtrackLCSarr(LCSarr,i, j-1)
    if LCSarr[i-1,j] >= LCSarr[i,j-1]
        backtrackLCSarr(LCSarr, i-1, j)
    return;
```

Рис. 2.15 – Пседокод функції backtrackLCSarr

У іншому разі відбувається перехід на наступний етап.

Далі, якщо списки *maybe-deleted-nodes* та *maybe-new-nodes* є непусті, то є ймовірність того, що всередині цих вузлів є s-вирази, що були переміщені на нове місце в новій версії програми.

Вирішенням цієї проблеми займається функція maybe-issue-resolver, яка у своїй роботі використовує сімейства функцій traverse-and-compare та collapse-node.

Сімейство функцій traverse-and-compare виконують прямий обхід підвузлів всередині вузла та пробують порівняти із ними вузол, що є :maybe-deleted (чи є частиною такого вузла). Суть у тому, що переміщений s-вираз може бути всередині нового s-виразу й функції traverse-and-compare перевіряють це. При знаходженні s-виразу, що був переміщений, відповідні йому вузли отримують як :diff-status Lisp-список виду “(:moved to-id)”, де to-

id це унікальний числовий ідентифікатор вузла для цього s-виразу в протилежній версії програми. Також інформація про це додається в список *moved-s-exprs-list* як об’єкт класу moved-s-exprs-info, що має такі поля:

- s-expr-id1 та s-expr-id2 – зберігають ідентифікатори відповідних вузлів переміщеного s-виразу;
- start-coord-of-id1 та end-coord-of-id1 – координати початку та кінця s-виразу в старій версії програми;
- start-coord-of-id2 та end-coord-of-id2 – координати початку та кінця s-виразу в новій версії програми.

Координати записуються для спрощення виділення курсором переміщеного s-виразу всередині frontend частини.

Сімейство функцій collapse-node просто розкладають вузли на підвузли.

Тепер можна надати псевдокод функції maybe-issue-resolver, що зображений на рис. 2.16.

```
function maybe-issue-resolver(*maybe-deleted-nodes*, *maybe-new-nodes*)
   repeat
       for кожний елемент m-del-n в *maybe-deleted-nodes*
           for кожний елемент m-new-n в *maybe-new-nodes*
               if compare(m-del-n, m-new-n) = true then
                  видалити m-del-n з *maybe-deleted-nodes*;
                  видалити m-new-n з *maybe-new-nodes*;
                  додати інформацію до *moved-s-exprs-list*;
               else
                  if traverse-and-compare(m-del-n, m-new-n) = true then
                     вадалити m-del-n з *maybe-deleted-nodes*;
                  endif
               endif
           endfor
       endfor
      застосувати до кожного елементу *maybe-deleted-nodes* функцію collapse-node
      та отримати з підвузлів новий список *maybe-deleted-nodes*;
  while  довжина *maybe-deleted-nodes* > 0 та довжина *maybe-new-nodes* > 0
end
```

Рис. 2.16 – Псевдокод функції maybe-issue-resolver

Після виконання функції maybe-issue-resolver обробка поточної пари def-вузлів закінчується. Якщо є наступна пара def-вузлів для обробки, то для них знову виконуються вищеописані дії. Якщо вже нема для поточного def-

type пари def-вузлів для опрацювання, то при наявності наступного def-type, він береться та весь цикл повторюється.

Як результат модуль порівняння абстрактних семантичних дерев повертає повертає абстрактні семантичні дерева із модифікованими вузлами та список *moved-s-exprs-list*.

2.6 Опис JSON файлів, що генеруються backend частиною

Результати роботи backend частини генеруються як сукупність JSON файлів. Було обрано використання JSON файлів для передачі інформації від backend частини до frontend частини, бо цей формат є структурованим, зручним та його підтримання є практично для всіх популярних мов програмування.

Генерація JSON файлів відбувається у функціях пакету :diff-backend/results-generator, що знаходиться всередині файлу DiffBackend/src/results-generator.lisp. Використовується для створення JSON файлів допоміжна бібліотека cl-json.

Залежно від конкретних умов можуть бути сгенеровані такі JSON файли:

- lexems1.json та lexems2.json;
- comments1.json та comments2.json;
- lexer-errors-msgs1.json та lexer-errors-msgs2.json;
- parser-error-msg1.json та parser-error-msg2.json;
- semantic-errors-msgs1.json та semantic-errors-msgs2;
- s-exprs-tree1.json та s-exprs-tree2.json;
- top-level-stats.json;
- moved-s-exprs-info.json.

Файли lexems1.json та lexems2.json генеруються, якщо були виявлені лексичні помилки або синтаксична помилка у відповідних вхідних файлах. Приклад внутрішньої структури цих файлів зображено на рис. 2.17.

Добавлено примечание ([OM20]): Якщо тут буде описано процес генерації, а не просто текст згенерованих json-файлів, то треба й називати «Генерація JSON файлів backend частиною»

Добавлено примечание ([D21R20]): Процес генерації не є взагалі цікавим, тому в цьому розділі буде описаєий зміст файлів

Добавлено примечание ([OM22R20]): ок

Рис. 2.17 – Приклад внутрішньої стуктури файлів lexems1.json та lexems2.json

Структура lexems1.json або lexems2.json проста – це масив JSON-об’єктів, що по своїй структурі ідентичні до об’єктів класу lexem, який розроблений у підрозділі 2.2. По суті ці файли зберігають масив лексем, що необхідно для відтворення тексту й виділення помилкових лексем у frontend частині.

Файли comments1.json та comments2.json генеруються, якщо у відповідних вхідних файлах було виявлено на лексичному аналізі коментарі. Приклад внутрішньої структури цих файлів зображено на рис 2.18.

Структура comments1.json або comments2.json проста – це JSON-об’єкт, ключі якого є номерами рядків, а значення є об’єктами, що описують коментар цього рядку. Вони потрібні для коректного відтворення змісту вхідних файлів у frontend частині.

Файли lexer-errors-msgs1.json та lexer-errors-msgs2.json генеруються, якщо у відповідних вхідних файлах було виявлено лексичні помилки. Приклад внутрішньої структури цих файлів зображено на рис. 2.19.

Рис 2.18 – Приклад внутрішньої структури файлів comments1.json та comments2.json

Структура lexer-errors-msgs1.json та lexer-errors-msgs2.json складається з масиву об’єктів, які по своїй структурі ідентичні до об’єктів класу lexem-error, що розроблений у підрозділі 2.2. Ці файли зберігають масив повідомлень про лексичні помилки разом з інформацією про відповідний їм ідентифікатор лексеми, що записаний відповідно або в lexems1.json, або в lexems2.json.

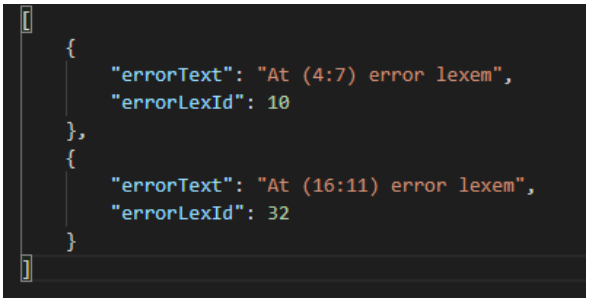

Рис. 2.19 – Приклад внутрішньої стуктури файлів lexer-errors-msgs1.json та lexer-errors-msgs2json

Файли parser-error-msg1.json та parser-error-msg2.json генеруються, якщо було виявлено синтаксичну помилку у відповідних вхідних файлах. Приклад внутрішньої структури цих файлів зображено на рис. 2.20.

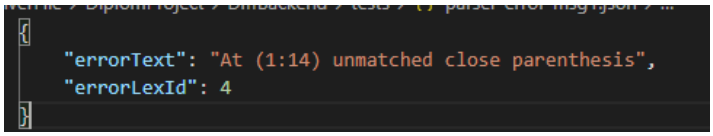

Рис. 2.20 - Приклад внутрішньої структури файлів parser-error-msg1.json та parser-error-msg2.json

Структура parser-error-msg1.json та parser-error2.json складається з JSON-об'єкту, шо по своїй структурі ідентичний до об'єкту класу parser-error-info, що розроблений у підрозділі 2.3. Ці файли зберігають повідомлення про синтаксичну помилку разом з інформацією про відповідний йому ідентифікатор лексеми, що записаний відповідно або в lexems1.json, або в lexems2.json.

Файли semantic-errors-msgs1.json та semantic-errors2.json генеруються, якщо були виявлені семантичні помилки у відповідних вхідних файлах. Приклад внутрішньої структури цих файлів зображено на рис. 2.21.

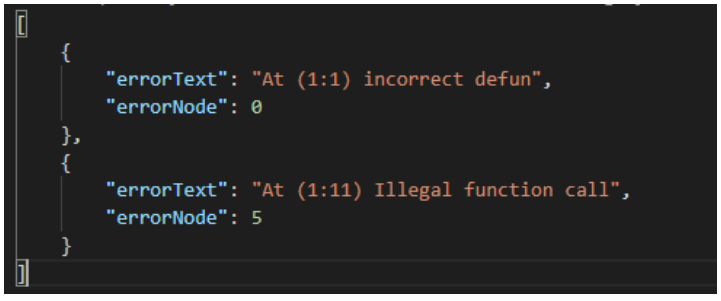

Рис 2.21 - Приклад внутрішньої стуктури файлів semantic-errors-msgs1.json та semantic-errors-msgs2.json

Структура semantic-errors-msgs1.json та semantic-errors-msgs2.json складається з масиву об’єктів, які по своїй структурі ідентичні до об’єктів класу semantic-error-info, що розроблений у підрозділі 2.4. Ці файли зберігають масив повідомлень про семантичні помилки разом з інформацією про відповідний їм ідентифікатор вузла, що знаходиться всередині відповідно або s-exprs-tree1.json, або s-exprs-tree2.json.

Файли s-exprs-tree1.json та s-exprs-tree2.json генеруються, якщо були згенеровані абстрактні семантичні дерева з текстів відповідних вхідних файлів. Приклад внутрішньої структури цих файлів зображено на рис. 2.22.

Рис 2.22 – Приклад внутрішньої структури s-exprs-tree1.json та s-exprs-tree2.json

Структури s-exprs-tree1.json та s-exprs-tree2.json цікаві, бо мають подібність до дерева синтаксичного розбору, що був розроблений у підрозділі 2.3. Всередині структура складається з масиву на верхньому рівні, який по суті відіграє роль кореня дерева, всередині якого знаходяться піддерева, вузли

яких будуються з JSON-об’єктів, які описують s-вирази. Це дерево будується практично прямим відображенням вузлів абстрактного семантичного дерева в JSON формат, але зі втратою всіх зайвих семантичних даних, які були необхідні для етапу порівняння, що розроблений у підрозділі 2.5.

Усі JSON-об’єкти, що описують s-вирази мають такі ключі:

- type – у нього записується інформація про тип s-виразу. Може мати значення “list”, “atom”, “quote”;
- id – записується унікальний ідентифікатор вузла;
- diff-st – записується інформація про статус вузла, що був визначений на етапі порівняння абстрактних семантичних дерев. Може мати значення “same”, “deleted”, “new”, або масиву “[“moved” node-id]”, де node-id – це ідентифікатор вузла в іншому s-exprs-tree, який вказує на логічну ідентичність цих вузлів та те, що цей вузол був переміщений в іншу позицію в новій версії;

JSON-об’єкти, що мають тип s-виразу “list” мають додатково такі ключі:

- par-info – записується інформація про координати круглих дужок списку;
- elems – записується масив з JSON-об’єктами, що описують s-вирази, які знаходяться всередині списку.

Також JSON-об’єкт, що має тип s-виразу “list”, може мати ключ props, куди записується додаткова інформація про цей вузол (наприклад, інформація про статус top-level форми цього s-виразу).

JSON-об’єкти, що мають тип s-виразу “atom” мають додатково такі ключі:

- lexem-coord – записується інформація про координати лексеми, що є атомом;
- lexem-type – записується тип лексеми атома;
- string – записується текстова форма лексеми атома.

JSON-об’єкти, що мають тип s-виразу “quote” мають додатково такі ключі:

- quote-coord – записується інформація про координати символа “”;
- q-s-expr – записується JSON-об’єкт, що описує s-вираз, до якого було застосовано символ “”.

Інформація з файлів s-exprs-tree1.json та s-exprs-tree2.json використовується у frontend частині для коректного відтворення текстів вхідних файлів, виділення певним кольором s-виразів тощо.

Файл top-level-stats.json генерується, якщо було виконано етап порівняння абстрактних семантичних дерев. Приклад внутрішньої структури цього файлу зображено на рис. 2.23.

Структура top-level-stats.json складається з JSON-об’єкту, де є ключі:

- old-ver – записується інформація по s-виразам верхнього рівня, які є def-s-виразами, у старій версії програми на мові LISP;
- new-ver – записується відповідно інформація вже по новій версії програми на мові LISP.

Значенням ключів old-ver та new-ver, є JSON-об’єкти, ключі якого мають назву конкретного типу def-s-виразу в множинній формі. Ці ключі зберігають об’єкт, ключі якого є назвою класу ідентифікаторів, а значення список список відповідних ідентифікаторів.

Рис 2.23. – Приклад внутрішньої структури top-level-stats.json

Файл moved-s-exprs-info.json генерується, якщо було виконано етап порівняння абстрактних семантичних дерев та були виявлені s-вирази, які були переміщені на нову позицію в новій версії LISP програми. Приклад внутрішньої структури цього файлу зображено на рис. 2.24.

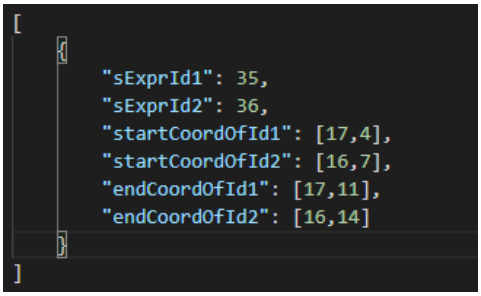

Рис. 2.24. – Приклад внутрішньої структури moved-s-exprs-info.json

Структура moved-s-exprs-info.json складається з масиву JSON-об'єктів, які по своїй структурі ідентичні об'єктам класу moved-s-expr-info, що був розроблений у підрозділі 2.5. Інформація з цього файлу використовується у frontend частині для ідентифікації виділення конкретного "переміщеного" s-виразу всередині переглядачів тексту.

2.7 Розробка frontend частини

Добавлено примечание ([ОМ23]): Аналогічно

Frontend частина – це частина, яка розроблена як окрема програма та займається наданням користувачу GUI та займається візуалізацією результатів роботи backend частини.

Frontend частина розроблена з використанням мови програмування C++ та з використанням кросплатформеного фреймворку Qt5, який використовується для створення GUI. Файли цієї частини знаходяться в папці DiffFrontend.

Frontend частина, як і будь-яка програма на C++, починає свою роботу з функції main, що знаходиться у файлі main.cpp. Функція main створює об'єкт класу MainWindow та запускає цикл нескінченний цикл реагування на дії користувача (на термінології Qt - це event loop), який може бути перерваний закриттям програми.

Клас MainWindow, що наслідується від класу QMainWindow, розроблений для реалізації роботи методів, які викликаються при взаємодії з елементами графічного інтерфейсу засобу. Найбільш цікавою функцією цього класу є on_startCompareButton_clicked, яка й викликає всередині себе backend частину засобу, надаючи як аргументи командного рядку шляхи до обраних користувачем файлів із різними версіями програм на мові LISP, та відбувається подальший аналіз створених backend частиною JSON файлів, залежно від яких вміст графічного інтерфейсу засобу буде видозмінено певним чином.

Також у класі MainWindow є такі методи:

- getJsonDocument – отримує об’єкт класу QJsonDocument із вмісту JSON файлів;
- cleanOldJsonFiles – очищує всередині папки засобу старі JSON файли;
- getStringFromFile – отримує об’єкт класу QString із вмісту файлу;
- тощо.

Також тут визначено enum ViewerMode, що має такі значення:

- NormalMode – означає звичайний режим роботи, коли порівняння у backend частині було виконано без помилок;
- ErrorsMode – означає режим демонстрації помилок, що були виявлені у backend частині.

Інформацію про початковий вигляд графічного інтерфейсу записано до файлу mainwindow.ui. Цей файл було розроблено з використанням засобу Qt Designer. Опис інтерфейсу надано в розділі 4.

Клас Global – це клас, що розроблений із використанням шаблону проєктування “Одинак”, що означає, що він може мати лише один екземпляр. Цей клас зберігає дані, що мають бути глобальними.

Клас DiffViewer – це клас, що наслідується від класу QPlainTextEdit та визначає віджет переглядача текстів.

Клас LineNumberArea – це клас, що наслідується від класу QWidget та визначає віджет області з номерами рядків.

Клас Stats – це клас, що відповідає за зберігання та обробку даних із файлу top-level-stats.json.

Клас DiffViewerText – це клас, що зберігає тестовий рядок, що буде відображатись у об’єкті класу DiffViewer.

Клас DiffViewerTextBuilder – це клас, що займається побудовою рядка для об’єкту класу DiffViewerText. Рядок будується як текст із тегами HTML, що дає змогу змінювати шрифт та колір тексту, а також робтити виділення певних фрагментів тексту певним кольором.

3. ТЕСТУВАННЯ ЗАСОБУ

3.1 Розробка юніт-тестів backend частини

Юніт-тести – це тести, які тестують роботу певних модулів. Вони надають змогу швидко тестувати роботу модуля під час процесу розробки та дають певні гарантії їхньої працездатності.

У backend частині для кожного модулю було розроблено свої юніт-тести, окрім модулю генерації JSON файлів. Для розробки та їх запуску було використано бібліотеку для юніт-тестів коду на мові програмування Common Lisp Rove [6].

Файли з юніт-тестами знаходяться в папці DiffBackend/tests.

Головним файлом у цій папці є файл test-engines.lisp, у якому знаходиться пакет :diff-backend/tests/test-engines.

У цьому пакеті розроблені такі допоміжні функції для визначення юніт-тестів:

- def-lexer-test – розроблений для визначення юніт-тестів лексичного аналізатора;
- def-parser-test – розроблений для визначення юніт-тестів синтаксичного аналізатора;
- def-ast-test – розроблений для визначення юніт-тестів модуля генерації абстрактного семантичного дерева;
- def-stats-test – розроблений для юніт-тестування роботи функцій пакету: diff-backend/statistics;
- def-comparator-test – розроблений для визначення юніт-тестів модуля порівняння абстрактних семантичних дерев.

Також було розроблено допоміжний пакет :diff-backend/tests/test-utils, що знаходиться у файлі test-utils.lisp.

У цьому пакеті розроблені такі функції:

- deep-equal – функція, яка проводить повне порівняння 2-х переданих їй об’єктів. У разі неспівпадіння друкує в консоль зручний для аналізу лог процесу порівняння;
- сімейство функцій conv-for-cmp-test, яке перетворює абстрактне семантичне дерево у форму, яка зручна для використання всередині def-comparator-test.

Усього було розроблено 90 юніт-тестів, із яких:

- 26 юніт-тестів лексичного аналізатора, що знаходяться в пакеті :diff-backend/tests/lexer;
- 11 юніт-тестів синтаксичного аналізатора, що знаходяться в пакеті :diff-backend/tests/parser;
- 15 юніт-тестів модуля генерації абстрактного семантичного дерева, що знаходяться в пакеті :diff-backend/tests/abstract-sem-tree-generator;
- 3 юніт-тести, що знаходяться в пакеті :diff-backend/tests/statistics;
- 35 юніт-тестів модуля порівняння абстрактних семантичних дерев, що знаходяться в пакеті :diff-backend/tests/comparator.

Для виконання юніт-тестів використовується виклик всередині REPL (read-evel-print loop), що зображений на рис. 3.1.

```
CL-USER> (asdf:test-system :diff-backend)
```

Рис. 3.1 – Команда запуску юніт-тестів backend частини

Унаслідок будемо мати такий вивід результатів юніт-тестування, що зображений на рис. 3.2.

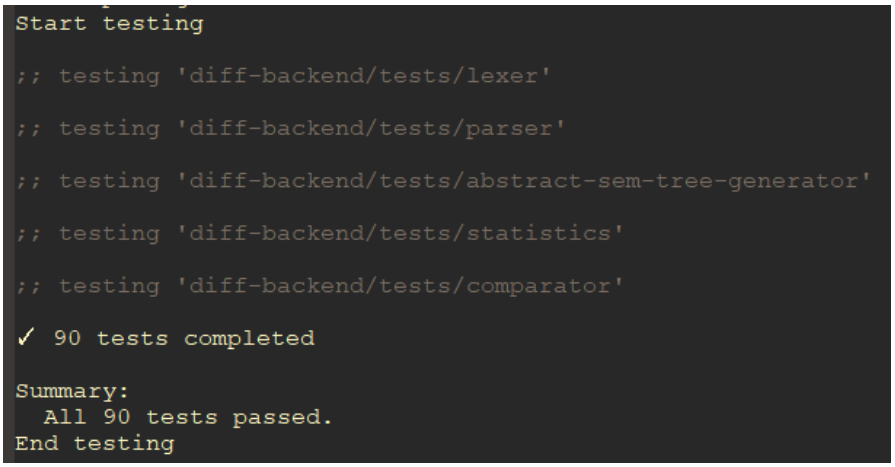

Рис. 3.2 – Результат виконання юніт-тестів backend частини

3.2 Ручне тестування роботи frontend частини

Оскільки за час, що виділений на виконання дипломного проєкту, практично неможливо розробити нормальні юніт-тести для тестування роботи графічного інтерфейсу, що реалізований із використанням мови програмування C++ та фреймворку Qt5, то frontend частина тестувалася вручну.

Підхід ручного тестування полягає у виконанні певних дій над елементами графічного інтерфейсу та перевірки того, що виконуються очікуванні дії.

Із скріншотів розділу 4 можна побачити, що графічний інтерфейс працює адекватно, а значить він був достатньо протестований цим способом.

4. КЕРІВНИЦТВО КОРИСТУВАЧА

4.1 Запуск засобу та стартове вікно засобу

Запуск засобу відбувається через запуск lisp-diff.exe всередині папки зі встановленим засобом.

Після запуску засобу з’явиться стартове вікно засобу, що зображено на рис. 4.1.

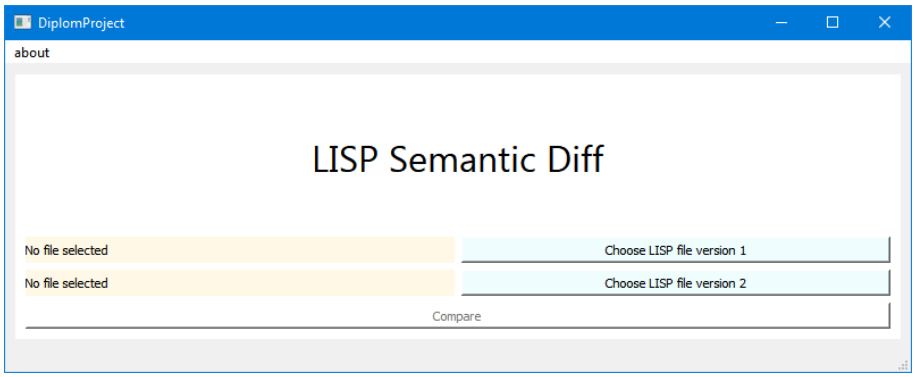

Рис. 4.1 – Стартове вікно засобу

Далі потрібно вибрати 2 версії файлу з програмою на мові LISP. Для цього потрібно відповідно натиснути на кнопку “Choose LISP file version 1” або “Choose LISP file version 2” після чого з’являться системні меню, які надаються засобами операційної системи, для вибору відповідних версій LISP файлу.

Після того як будуть обрані файли стартове вікно буде мати вигляд як на рис. 4.2.

У місцях, де був текст “No file selected”, буде зараз відображено повний шлях у файловій системі до відповідного обраного файлу. Кнопка “Compare” стане активною для використання.

Для активації порівняння файлів потрібно зараз натиснути на кнопку “Compare”.

Якщо під час порівнянні були виявлені помилки у файлах, то буде відображене вікно режиму помилок, користування яким описано в підрозділі 4.2.

Якщо помилок не було виявлено та порівняння було виконано успішно, то буде відображено вікно перегляду результатів порівняння, користування яким описано в підрозділі 4.3.

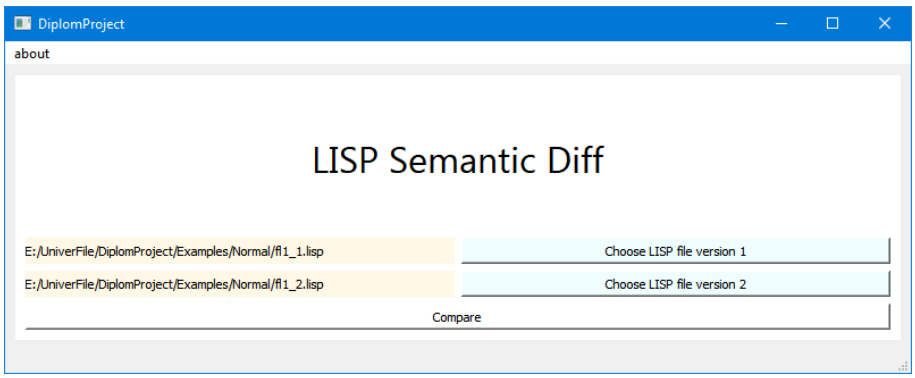

Рис. 4.2 – Стартове вікно засобу після вибору LISP файлів

4.2 Опис вікна режиму помилок

Вікно режиму помилок з’являється, якщо було виявлено лексичні, синтаксичні чи семантичні помилки хоча б в одному із файлів. При цьому також з’являється вікно повідомлення про помилки, що зображено на рис. 4.3.

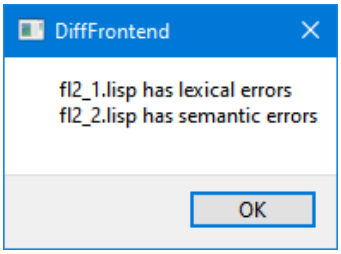

Рис. 4.3 – Вікно повідомлення про помилки

Вікно режиму помилок має вигляд, що зображений на рис. 4.4.

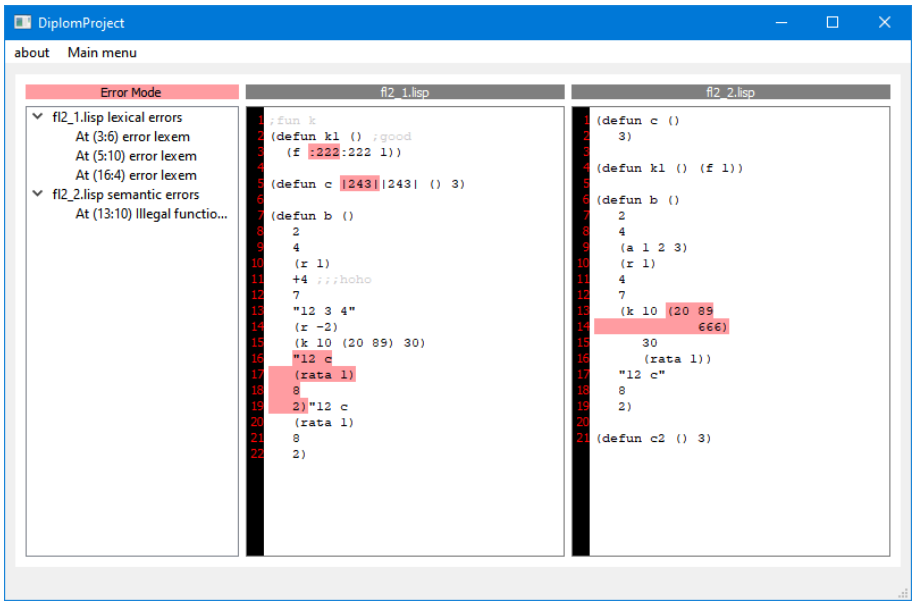

Рис. 4.4. – Вікно режиму помилок

Всередині переглядачів тексту знаходиться зміст відповідних файлів, де червоним кольором виділено частини, що є причинами відповідних помилок.

У лівій панелі знаходиться деревоподібна структура, де кореневі елементи відображають ім’я файлу й тип помилок у ньому. Дочірні елементи вказують у своєму тексті координати помилки й текст, що описує відповідну помилку. Якщо натиснути на дочірній елемент, то відповідний їй помилковий фрагмент буде виділений у переглядачі тексту відповідного файлу в жовтий колір. Приклад цього зображено на рис. 4.5.

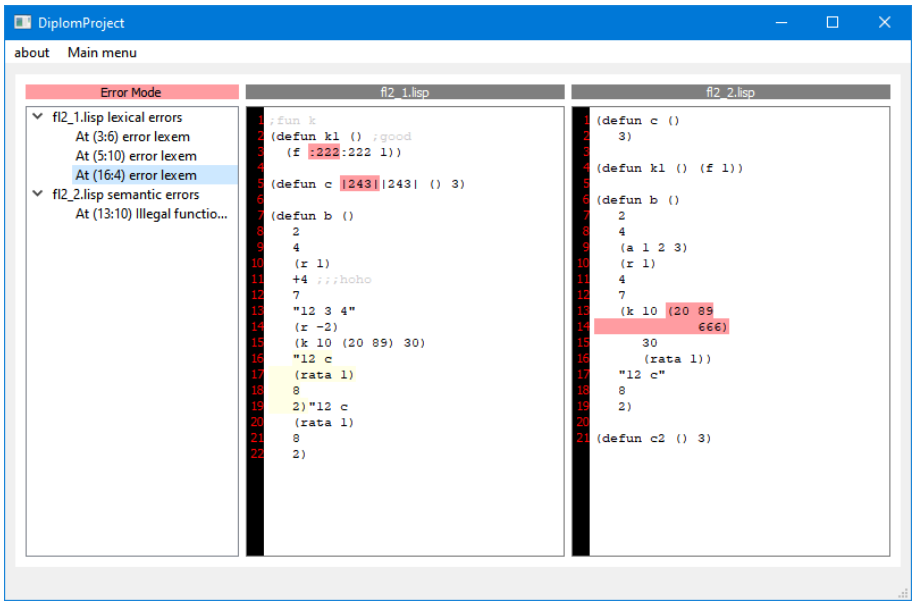

Рис. 4.5. – Виділення конкретної помилки в режимі помилок

4.3 Опис вікна перегляду результатів порівняння

За браком жодних відповідних помилок у файлах відображається вікно перегляду результатів порівняння. Його вигляд для конкретних прикладів вхідних файлів зображено на рис 4.6.

Вікно програми поділено на 3 області:

- область навігації за результатами порівняння;
- область перегляду коду 1-го вхідного файлу;
- область перегляду коду 2-го вхідного файлу.

В областях перегляду коду s-вирази можуть бути виділені за результатами порівняння певним кольором:

- зеленим виділяються s-вирази, додані в новій версії LISP програми;
- червоним виділяються s-вирази, видалені в новій версії LISP програми;

– блакитним виділяються s-вирази, що переміщенні з одного місця на інше в новій версії LISP програми.

Коментарі мають сірий колір тексту.

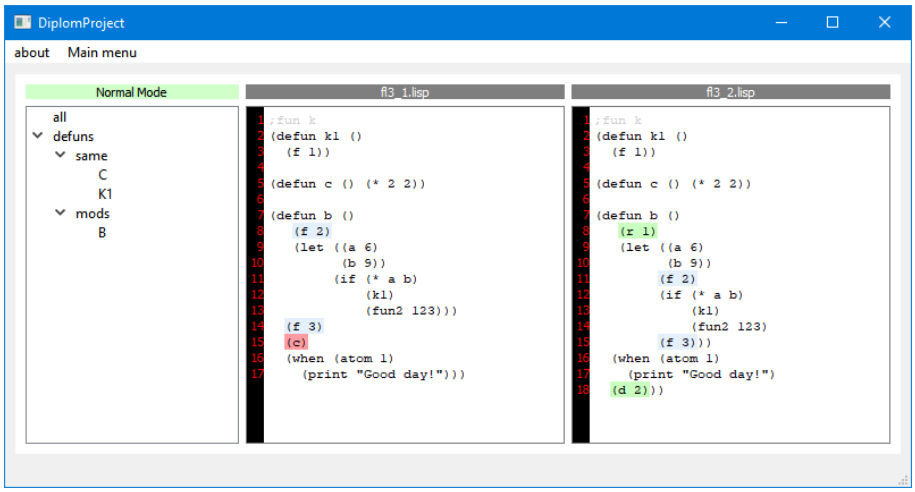

Рис. 4.6 – Вікно перегляду результатів порівняння

Якщо в областях перегляду коду натиснути на s-вираз, що був помічений як переміщений, то він буде виділений фіолетовим кольором в областях перегляду коду. Це надає можливість візуально його виділити з поміж інших переміщених s-виразів. Приклад цього зображено на рис. 4.7.

В області навігації за результатами порівняння є 2 основні елементи:

– all – при натисканні на нього відбувається демонстрація всього коду в областях перегляду коду;
– дерева навігації по ідентифікаторам def-s-виразів.

У дереві навігації по ідентифікаторам def-s-виразів на 1 рівні знаходиться назва конкретного def-s-виразу в множині. На 2 рівні знаходяться класи ідентифікаторів, що були виявлені при порівнянні. На цьому рівні можуть бути такі класи:

- same – сюди записуються ідентифікатори тих def-s-виразів, що не були модифіковані в новій версії LISP програми;
- mods – сюди записуються ідентифікатори тих def-s-виразів, що були модифіковані в новій версії LISP програми;
- dels - сюди записуються ідентифікатори тих def-s-виразів, що були видалені в новій версії LISP програми;
- news – сюди записуються ідентифікатори тих def-s-виразів, що були додані до нової версії LISP програми.

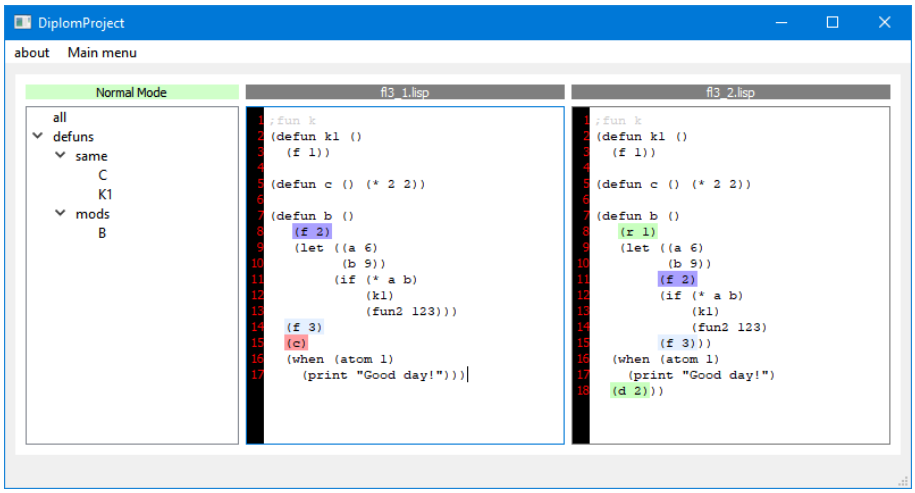

Рис. 4.7 – Виділення переміщеного s-виразу

Якщо натиснути на ім’я ідентифікатора в області навігації, то в разі news та dels ідентифікаторів їхній відповідний код з’явиться у відповідній області перегляду коду. У разі same та mods ідентифікаторів в областях перегляду коду з’явиться відповідний їм код. У будь-якому разі номера рядків будуть відповідати місцеположенню коду def-s-виразу у вхідному файлі. Приклад виділення конкретного ідентифікатора зображено на рис. 4.8. Ця можливість дуже корисна для перегляду модифікацій def-s-виразів із класу mods.

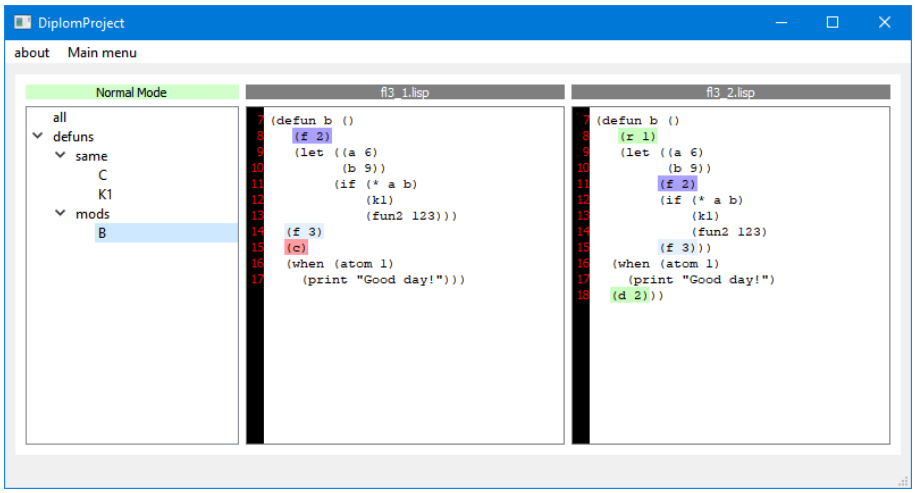

Рис. 4.8 – Виділення ідентифікатора в дереві навігації

ВИСНОВОК

У цьому дипломному проєкті було розроблено засіб порівняння різних версій програм на мові LISP з використанням абстрактного семантичного дерева, який дає змогу структурно порівнювати різні версії програм на мові LISP, що може покращити ефективність та продуктивність розробки програмного забезпечення на мові LISP.

Тема цього дипломного проєкту була обрана як результат того, що на ринку немає подібного засобу для мови LISP.

Розроблений засіб має такі особливості:

- наявність зручного графічного інтерфейсу;
- зручна демонстрація лексичних, синтаксичних чи семантичних помилок, якщо вони наявні у вхідних файлах;
- проводить порівняння між відповідними def-s-виразами та є можливість побачити результати цього порівняння окремо від інших def-s-виразів;
- виявляються s-вирази, що були додані, видалені чи переміщені в новій версії програми на мові LISP.

Подальший розвиток цього засобу полягає в таких речах:

- додавання підтримання більш складної семантики, яка є у мові LISP;
- додавання нових можливостей до GUI;
- покращення та оптимізація алгоритму порівняння абстрактних семантичних дерев;
- додати в засіб можливість структурного поєднання файлів із текстами різних версій програм на мові LISP, який буде спиратися на результати структурного порівняння.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. File comparison: URL: https://en.wikipedia.org/wiki/File_comparison (дата звернення: 17.04.2021)
2. Comparison of file comparison tools: URL: https://www.wikiwand.com/en/Comparison_of_file_comparison_tools (дата звернення: 18.04.2021)
3. Meld: URL: https://meldmerge.org/ (дата звернення 18.04.2021)
4. Semantic Merge: URL: https://www.semanticmerge.com/ (дата звернення 19.04.2021)
5. Common Lisp HyperSpec: URL: http://www.lispworks.com/documentation/HyperSpec/Front/ (дата звернення 21.04.2021)
6. Common Lisp testing library Rove: URL: https://github.com/fukamachi/rove (дата звернення 15.05.2021)