НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ І. СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп’ютерних систем

«До захисту допущено»
Завідувач кафедри
_____ В.П. Тарасенко
(підпис)
“___”_____2019р.

Дипломний проект

освітньо-кваліфікаційного рівня “Бакалавр”
з напряму підготовки 6.050102 “Комп’ютерна інженерія”
на тему: «ГЕНЕРАТОР СИНТАКСИЧНИХ АНАЛІЗАТОРІВ ДЛЯ LL(*) ГРАМАТИК»

Виконав: студент 4 курсу, групи КВ-51
Шулєпов Владислав Вікторович _____ (підпис)

Керівник доц., доц., к.т.н. Марченко О.І. _____ (підпис)

Консультант з нормоконтролю доц., доц., к.т.н. Клятченко Я.М. _____ (підпис)

Рецензент доц., доц., к.т.н. Заболотня Т.М _____ (підпис)

Київ – 2019

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет прикладної математики

Кафедра системного програмування та спеціалізованих комп’ютерних систем
Освітньо-кваліфікаційний рівень “Бакалавр”
Напрям підготовки 6.050102 “Комп’ютерна інженерія”

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ В.П. Тарасенко
“___” _____ 2019 р.

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЕКТ СТУДЕНТУ

Шулєпову Владиславу Вікторовичу

1. Тема проекту «ГЕНЕРАТОР СИНТАКСИЧНИХ АНАЛІЗАТОРІВ ДЛЯ LL(*) ГРАМАТИК»,
керівник проекту Марченко Олександр Іванович, к.т.н., доцент,
затверджені наказом по університету від “22” травня 2019 року № 1330-С
2. Строк подання студентом проекту: “13” травня 2019 р.
3. Вихідні дані для дипломного проектування: див. Технічне завдання.
4. Перелік задач, які потрібно вирішити:
   – розробити загальну структуру програми-транслятора;
   – виконати програмну реалізацію парсера граматики;
   – розробити побудовник LR(1)-таблиць;
   – виконати програмну реалізацію синтаксичного аналізатора LR(1);
   – виконати тестування програми-транслятора.

5. Перелік обов’язкового ілюстративного матеріалу:
   - взаємозв’язок модулів аналізатора (схема структурна);
   - алгоритм роботи програми (блок-схема);
   - модуль графічного інтерфейсу (схема структурна);
   - шаблони згенерованих класів (UML-діаграма).

6. Консультанти:

Питання	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Клятченко Я.М., доц., к.т.н.		

7. Дата видачі завдання: “___” _____ 2019 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи та питань, які мають бути розроблені відповідно до завдання	Термін виконання	Примітка
1.	Видача завдання на дипломне проектування	10.02.2019	
2.	Вивчення літератури за тематикою проекту	15.04.2019	
3.	Розробка та узгодження технічного завдання	30.04.2019	
4.	Аналіз існуючих рішень	05.05.2019	
5.	Розробка програми розбору граматик	08.05.2019	
6.	Розробка лексичного аналізатора	10.05.2019	
7.	Розробка синтаксичного аналізатора	12.05.2019	
8.	Розробка графічного інтерфейсу	14.05.2019	
9.	Відлагодження програмного продукту	16.05.2019	
10.	Підготовка пояснювальної записки	18.05.2019	
11.	Підготовка графічної частини дипломного проекту	20.05.2019	
12.	Оформлення документації дипломного проекту	25.05.2019	

Студент _____ Шулєпов В.В.

Керівник проекту _____ Марченко О.І.

АНОТАЦІЯ

Кваліфікаційна робота включає пояснювальну записку (52 с., 19 рис., 1 табл., 4 додатки).

Об’єкт розробки — розробка комп’ютерної програми генератора синтаксичних аналізаторів для LL(*) граматик.

Даний програмний комплекс створений для автоматизації, спрощення та пришвидшення процесу розробки програмного забезпечення, що створюється у галузях, пов’язаних з розробкою нових мов програмування. До нього входять наступні елементи:

- програма розбору LL(*) граматик, що перетворює текст опису граматики у її внутрішнє представлення та генерує програмний код функцій лексичного та синтаксичного аналізаторів;
- програма лексичного аналізатора, що виконує лексичний розбір вхідного тексту за заданою граматикою за допомогою згенерованих функцій;
- програма синтаксичного аналізатора, що генерує внутрішнє представлення вхідного тексту у вигляді синтаксичного дерева;
- простий приклад на основі серії оптимізацій вхідного коду, що демонструє практичне використання розробленого програмного комплексу;
- графічний інтерфейс, що забезпечує можливість використання програмного комплексу звичайним користувачем.

Програмний комплекс забезпечує можливість на практиці ознайомитись з описом граматик мов програмування; задати власну мову за допомогою LL(*) граматики; демонструє роботу лексичного та синтаксичного аналізаторів у складі компілятора/транслятора.

Ключові слова: LL(*) граматика, лексичний аналізатор, синтаксичний аналізатор, дерево розбору.

ABSTRACT

The qualifying work includes an explanatory note (52 p., 19 images, 1 table, 4 annexes).

The object of development — the development of a computer program generator parser for LL (*) grammar.

This software package is designed to automate, simplify, and speed up the software development process created in the fields related to the development of new programming languages. It includes the following elements:

- LL (*) grammar parsing program, which transforms the grammar description text into its internal representation and generates the program code of the functions of lexical and syntactic analyzers;
- program of lexical analyzer, which performs lexical analysis of input text in a given grammar using generated functions;
- a parser parser program that generates an internal representation of the input text as a syntax tree;
- A simple example based on a series of optimizations of the input code, which demonstrates the practical use of the developed software complex;
- A graphical interface that provides the ability to use the software complex by an ordinary user.

The program complex provides the opportunity to practice the description of the grammar of programming languages; Set your own language using LL (*) grammar; Demonstrates the work of lexical and parser analyzers in the compiler / translator.

Key words: LL (*) grammar, lexical analyzer, parser, parsing tree.

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
			Документація загальна			
			Новорозроблена			
	А4	ІАЛЦ.467200.002 ТЗ	Генератор синтаксичних	4		
			аналізаторів LL(*)			
			граматик.			
			Технічне задання			
	А4	ІАЛЦ. 467200.003 ТП	Генератор синтаксичних	3		
			аналізаторів LL(*)			
			граматик.			
			Відомість технічного			
			проекту			
	А4	ІАЛЦ. 467200.004 ПЗ	Генератор синтаксичних	53		
			аналізаторів LL(*)			
			граматик.			
			Пояснювальна записка			
	А4	ІАЛЦ. 467200.005 Д1	Генератор синтаксичних	1		
			аналізаторів LL(*)			
			граматик.			
			Взаємозв'язок модулів			
			аналізатора			

				ІАЛЦ.467200.001 ОА	
Змін.	Арк.	№ докум.	Підпис	Дата	
Розробив		Шулєпов В.В..			Генератор синтаксичних аналізаторів для LL(*) граматик
Перевірив		Марченко О.І.			
Н. контроль		Клятченко Я.М.			Опис альбому
Затвердив		Тарасенко В.П.			

Літ.	Аркуш	Аркушів
	1	2

КПІ ім. Ігоря Сікорського, ФПМ, КВ-51

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
	А4	ІАЛЦ. 467200.006 Д2	Генератор синтаксичних	1		
			аналізаторів LL(*)			
			граматик.			
			Алгоритм роботи програми			
	А4	ІАЛЦ. 467200.007 Д3	Генератор синтаксичних	1		
			аналізаторів LL(*)			
			граматик.			
			Модуль графічного			
			інтерфейсу			
	А4	ІАЛЦ. 467200.008 Д4	Генератор синтаксичних	1		
			аналізаторів LL(*)			
			граматик.			
			Шаблон згенерованих			
			класів			

					ІАЛЦ. 467200.001 ОА	Арк.
						2
Змін.	Арк.	№ докум.	Підпис	Дата		

ЗМІСТ

стор.

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ……..2
2. ПІДСТАВИ ДЛЯ РОЗРОБКИ…….……...…….2
3. МЕТА ВИКОНАННЯ…….…….……..…….2
4. ДЖЕРЕЛА РОЗРОБКИ…….……..2
5. ТЕХНІЧНІ ВИМОГИ…….…….3
   - 5.1 Вимоги програми розбору LL(*) граматик...……3
   - 5.2 Вимоги лексичного аналізатора.....……3
   - 5.3 Вимоги синтаксичного аналізатора…….....……...…..3
   - 5.4 Вимоги до програмного забезпечення…….……...….4
   - 5.5 Вимоги до апаратного забезпечення……...……..4
6. ЕТАПИ РОЗРОБКИ……...……..4

					ІАЛЦ.467200.002 ТЗ		
Зм.	Арк.	№ докум.	Підп.	Дата			
Розроб.		Шулєпов В.В.			Генератор синтаксичних аналізаторів для LL(*) граматик	Літ.	Аркуш 1 / Аркушів 4
Перевір.		Марченко О.І.					
Н. контр.		Клятченко Я.В.			Технічне завдання	КПІ ім. І. Сікорського ФПМ, КВ-51	
Затв.		Тарасенко В.П.					

1. НАЙМЕНУВАННЯ І ОБЛАСТЬ ЗАСТОСУВАННЯ

Найменування роботи – «Генератор синтаксичних аналізаторів для LL(*) граматик».

Область застосування: використання при розробці трансляторів та компіляторів, при аналізі та оптимізації програмного коду.

2. ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на дипломне проектування, затверджене кафедрою системного програмування і спеціалізованих комп'ютерних систем Національного технічного університету України «Київський Політехнічний Інститут».

3. ЦІЛЬ І ПРИЗНАЧЕННЯ РОБОТИ

Метою даного проекту є створення програми генератора синтаксичних аналізаторів для мов програмування, які можна описати за допомогою LL(*) граматик.

4. ДЖЕРЕЛА РОБОТИ

Джерелами роботи є конспект лекцій з курсу «Інженерія програмного забезпечення 1: Основи програмування трансляторів», науково-технічна література з теорії розробки компіляторів та статті у мережі Інтернет.

5. ТЕХНІЧНІ ВИМОГИ

5.1 Вимоги програми розбору LL(*) граматик:

- На вхід приймає текст опису граматики.
- Будує внутрішнє представлення граматики.
- Видає повідомлення про помилки при некоректних вхідних даних.
- Генерує коду функцій лексичного та синтаксичного аналізаторів для заданої граматики.
- Виходом програми є внутрішнє представлення граматики.

5.2 Вимоги до лексичного аналізатора:

- На вхід приймає лексичну частину внутрішнього представлення граматики.
- Виконує розбиття вхідного тексту на лексеми за необхідності.
- Генерує повідомлення про лексичні помилки при некоректних вхідних даних.

5.3 Вимоги до синтаксичного аналізатора:

- На вхід приймає синтаксичну частину внутрішнього представлення граматики та вхідний текст програми на мові, що задана цією граматикою.
- Використовує лексичний аналізатор для розбиття вхідного тексту на лексеми.
- На основі отриманих на етапі лексичного аналізу лексем створює внутрішнє представлення вхідної програми — дерево розбору.
- Генерує повідомлення про синтаксичні помилки при некоректних вхідних даних.
- Виходом аналізатора є внутрішнє представлення вхідного тексту програми.

5.4 Вимоги до програмного забезпечення:

- Операційна система Windows 10, Linux.
- Java Development Kit (JDK) 8.

5.5 Вимоги до апаратного забезпечення:

- Процесор MediaTek, Snapdragon, Kirin, Intel Atom;
- Оперативна пам’ять: 2 Гб.

6. ЕТАПИ РОЗРОБКИ

№ з/п	Назва етапів роботи та питань, які мають бути розроблені відповідно до завдання	Термін виконання
1.	Видача завдання на дипломне проектування	10.02.2019
2.	Вивчення літератури за тематикою проекту	15.04.2019
3.	Розробка та узгодження технічного завдання	30.04.2019
4.	Аналіз існуючих рішень	05.05.2019
5.	Розробка програми розбору граматик	08.05.2019
6.	Розробка лексичного аналізатора	10.05.2019
7.	Розробка синтаксичного аналізатора	12.05.2019
8.	Розробка графічного інтерфейсу	14.05.2019
9.	Відлагодження програмного продукту	16.05.2019
10.	Підготовка пояснювальної записки	18.05.2019
11.	Підготовка графічної частини дипломного проекту	20.05.2019
12.	Оформлення документації дипломного проекту	25.05.2019
13.	Попередній огляд матеріалів диплому на кафедрі	30.05.2019

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
			Документація загальна			
			Новорозроблена			
	А4	ІАЛЦ. 467200.004 ПЗ	Генератор синтаксичних	53		
			аналізаторів LL(*)			
			граматик.			
			Пояснювальна записка			
	А4	ІАЛЦ. 467200.005 Д1	Генератор синтаксичних	1		
			аналізаторів LL(*)			
			граматик.			
			Взаємозв'язок модулів			
			аналізатора			
	А4	ІАЛЦ. 467200.006 Д2	Генератор синтаксичних	1		
			аналізаторів LL(*)			
			граматик.			
			Алгоритм роботи програми			
	А4	ІАЛЦ. 467200.007 Д3	Генератор синтаксичних	1		
			аналізаторів LL(*)			
			граматик.			
			Модуль графічного			
			інтерфейсу			

Змін.	Арк.	№ докум.	Підпис	Дата	ІАЛЦ. 467200.003 ТП		
Розробив		Шулєпов В.В			Генератор синтаксичних аналізаторів для LL(*) граматик Відомість технічного проекту	Літ.	Аркуш 1 / Аркушів 2
Перевірив		Марченко О.І.					
Н. контроль		Клятченко Я.М.				КПІ ім Ігоря Сікорського, ФПМ, КВ-51	
Затвердив		Тарасенко В.П.					

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
	А4	ІАЛЦ. 467200.008 Д4	Генератор синтаксичних	1		
			аналізаторів LL(*)			
			граматик.			
			Шаблон згенерованих			
			класів			
		Диск CD-ROM	Текст ПЗ. Тексти програм.			
			Графічний матеріал			
			.			

Змін.	Арк.	№ докум.	Підпис	Дата	ІАЛЦ. 467200.003 ТП	Арк. 2

ЗМІСТ

стор.

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ.....4
ВСТУП.....6
1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ОБҐРУНТУВАННЯ ТЕМИ ДИПЛОМНОГО ПРОЕКТУ.....8
1.1. Теорія формальних граматик.....8
1.2. Класифікація синтаксичних аналізаторів.....12
1.3. Інтеграція лексичного та синтаксичного аналізаторів.....13
1.4. Автоматизація розробки синтаксичних аналізаторів.....14
1.5. Огляд існуючих аналогів.....15
1.6. Обґрунтування теми дипломного проекту.....16
2. РОЗРОБКА КОМПОНЕНТІВ СИСТЕМИ.....18
2.1. Структура програмного комплексу.....18
2.2. Модуль аналізу граматики.....18
2.3. Реалізація лексичного аналізатора.....21
2.4. Алгоритм роботи синтаксичного аналізатора.....23
2.5. Компоненти графічного інтерфейсу.....25

2.6. Модуль налаштувань проекту.....33
2.7. Структура та призначення основних методів класів проекту.....35
3. ТЕСТУВАННЯ СИСТЕМИ.....43
3.1. Модуль тестування.....43
3.2. Обробка повідомлень про помилки.....44
ВИСНОВКИ…….....51
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ.....53

ДОДАТКИ

Додаток 1. Копії графічних матеріалів

- ІАЛЦ.045440.005 Д1. Взаємозв’язок модулів аналізатора. Схема структурна.
- ІАЛЦ.045440.006 Д2. Алгоритм роботи програми. Блок-схема.
- ІАЛЦ.045440.007 Д3. Модуль графічного інтерфейсу. Схема структурна.
- ІАЛЦ.045440.008 Д4. Шаблони згенерованих класів. UML-діаграма.

Додаток 2. Лістинги

- Лістинг програми.
- Лістинг файлу тестової граматики.

Додаток 3. Презентація

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

СПТ — система побудови трансляторів.

БНФ (BNF) — Backus-Naur form, нотація Бекуса-Наура, спосіб запису правил контекстно-вільної граматики.

РБНФ (EBNF) — extended Backus–Naur form, розширена форма Бекуса-Наура.

КЗ — контекстно-залежна (граматика, мова).

СА — синтаксичний аналізатор.

Лексер (lexer) – програма, що виконує лексичний аналіз.

Парсер (parser) – програма, що виконує синтаксичний аналіз.

Лексема — 1) слово як самостійна значеннєва одиниця; 2) послідовність машинних символів вихідного коду програми, що мають певне сукупне значення.

CSV (comma-separated values) — файловий формат, що є представленням табличних даних. Поля відокремлюються одне від одного символом коми, записи — символом нового рядка. Формат CSV є досить простим форматом даних, підтримується більшістю табличних процесорів, систем керування базами даних та мов програмування.

SQL (structured query language) — декларативна мова програмування, що використовується для взаємодії користувача з базами даних, для формування запитів, модифікування даних, створення та зміни моделі та схеми бази даних. Стандартизована Американськими Держстандартами (ANSI) в 1986.

XML (Extensible Markup Language) — стандарт побудови мов розмітки ієрархічно структурованих даних. Визначає метамову, що шляхом запровадження обмежень на структуру та зміст документів визначає предметно-орієнтовані мови розмітки даних. Ці обмеження описуються мовами схем, наприклад XML Schema, DTD або RELAX NG. Специфікації

та стандарти XML видаються комітетом W3C, поточний стандарт — версії 1.1.

DSL (domain-specific language) — мова програмування, призначена для вирішення задач конкретної предметної галузі, яка дає особливе уявлення про проблему або специфічні методи її вирішення. Є протилежність мов програмування загального призначення. DSL близька до описових мов HTML та XML.

Мок (mock) - об'єкт, що імітує поведінку реальних об'єктів контрольованим способом, реалізуючи інтерфейси справжніх об'єктів, проте не надаючи власної реальної функціональності. Один з основних елементів, що використовуються в модульному тестуванні програмного забезпечення.

Стаб (stub) - функція, що не виконує жодної осмисленої дії, і яка повертає порожній результат або власні вхідні дані без внесення змін до них. По своїй суті є заглушкою методу. Один з основних елементів, що використовуються в модульному тестуванні програмного забезпечення.

ВСТУП

Розвиток ІТ галузі, поява принципово нових задач та завдань призводить до необхідності розробки більш зручних, швидких та ефективних інструментів, в тому числі — нових мов програмування.

Ці нові мови все більше відходять від «низького» рівня програмування, прямуючи до більш «високого», використовуючи нові конструкції та підходи, зменшуючи тим самим об’єм програмного коду при написанні програм. Використання нових конструкцій, збільшення рівня абстракції досягається ускладненням граматик мов програмування, а також ускладненням програм-трансляторів. Виникає потреба в спрощенні процесу розробки цих програм.

Нові мови програмування вимагають розробки нових трансляторів/компіляторів та покращення існуючих. Наявність у мові високорівневих конструкцій потребує обробки складних граматичних конструкцій, тому з кожним роком розробка та підтримка компіляторів ускладнюється. Системи побудови трансляторів (системи генерації лексичних та синтаксичних аналізаторів) дають змогу автоматизувати, спростити та пришвидшити процес розробки.

На сьогоднішній день існує багато систем, що дозволяють згенерувати синтаксичний аналізатор по деякій граматиці. Вони називаються системам побудови трансляторів. Вони приймають на вхід граматику мови (наприклад, LR(1), LL(*), LALR тощо), описану в деякій визначеній формі (наприклад, БНФ або РБНФ), і результатом їх роботи є згенерований деякою мовою програмування (C, Java та ін.) синтаксичний аналізатор для цієї граматики.

Оскільки створення СА за допомогою СПТ є значно швидшим і набагато менш трудомістким процесом, який потребує лише правильно заданої граматики, СПТ набули досить широкого застосування.

Розроблений програмний комплекс має на меті дати користувачу можливість ознайомитись з будовою компілятора та принципами його роботи як в цілому, так і з його окремими складовими частинами, наглядно бачити, як проходить процес трансляції програми, які помилки можуть виникати в процесі лексичного та синтаксичного аналізу. Також на меті є надання можливості покращення практичних навичок в сфері описі граматик мов програмування, і, відповідно, створення нових мов.

1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ОБҐРУНТУВАННЯ ТЕМИ ДИПЛОМНОГО ПРОЕКТУ

1.1. Теорія формальних граматик

Граматика — в теорії формальних мов — спосіб опису формальної мови, виділення деякої підмножини із множини всіх слів деякого скінченного алфавіту. Розрізняються породжувальні граматики, що задають правила, за якими можна побудувати будь-яке слово мови, та аналітичні, що задають правила, за якими можна визначити, чи входить конкретне слово в задану до алфавіту заданої мови. Формальні граматики були введені американським вченим, математиком та філософом, Н. Хомським у 50-тих роках XX сторіччя.

Поняття формальних граматик

Формальна граматика — це четвірка $G = \{ N , T , P , S \}$, де:

- **T** — алфавіт термінальних символів, терміналів. Термінальні символи є алфавітом мови.
- **N** — алфавіт нетермінальних символів, нетерміналів. Нетермінали не входять в алфавіт мови.

$$T \cap$$

- **S** — аксіома, спеціально виділений нетермінальний символ з якого починається опис граматики.

$$S$$

- **P** — правила виводу, скінченна підмножина множини

$$(T \cup N)^{+} \times (T \cup N)^{*}.$$

Класифікація формальних граматик за Хомським

1. Граматики типу 0

Граматики типу 0 називають необмеженими. До них належать всі формальні граматики.

Кожна граматика типу 0 породжує мову, яку може розпізнати машина Тюринга, та навпаки: для кожної мови, яку може розпізнати машина Тюринга, існує граматика типу 0, яка здатна її породити [1].

2. Граматики типу 1

До граматик типу 1 (контекстно-залежні граматики) належать всі граматики типу 0, в яких правила виводу обмежено правилами вигляду

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

або

$$S \rightarrow \varepsilon$$

, де A — нетермінальний, а α, γ, β слова з термінальних (Σ) та нетермінальних (N) символів. Слова α та β можуть бути порожніми, але γ має містити щонайменше один символ (термінальний або нетермінальний).

Правило $S \rightarrow \varepsilon$ дозволене, якщо S не зустрічається в правій частині жодного з правил. Це правило необхідне для додавання порожнього слова ε.

Контекстно-залежні (КЗ) граматики породжують контекстно-залежні мови; тобто, кожна КЗ граматика породжує КЗ мову, і навпаки — для кожної КЗ мови існує КЗ граматика, що її породжує.

Контекстно-залежні мови можна розпізнати недетермінованою лінійно-обмеженою машиною Тюринга; тобто, недетермінованою машиною Тюринга, довжина стрічки якої обмежена [1].

3. Граматики типу 2

В кожному правилі граматики типу 2 (контекстно-вільної граматики) з лівої сторони знаходиться один нетермінальний символ, а з правої сторони — можливо порожня послідовність термінальних та нетермінальних символів. Тобто, на правила виводу накладено обмеження

$$\forall (w_1 \to w_2) \in P : (w_1 \in N) \wedge (w_2 \in (N \cup \Sigma)^*)$$

Граматики типу 2 мають лише правила виду $A \to \gamma$, A — нетермінальний символ, а γ складається з послідовності термінальних та нетермінальних символів.

Контекстно-вільні (КВ) граматики породжують КВ мови, і для кожної КВ мови існує КВ граматика, що її породжує.

Контекстно-вільні мови можуть бути розпізнані недетермінованими автоматами з магазинною пам'яттю. КВ мови використовуються в побудові синтаксису мов програмування [1].

4. Граматики типу 3

Граматики третього типу називають регулярними. До них належать граматики другого типу, правила виводу яких обмежено

$$\forall (w_1 \to w_2) \in P : (w_1 \in N) \wedge (w_2 \in \Sigma \cup (\Sigma \cdot N \vee N \cdot \Sigma) \cup \{ \varepsilon \})$$

Для кожної ліволінійної граматики існує праволінійна граматика, та навпаки.

Регулярні граматики породжують регулярні мови, і для кожної регулярної мови існує регулярна граматика, що її породжує.

Регулярні мови можна описувати також регулярними виразами. Регулярні мови можна розпізнати скінченними автоматами. Їх часто використовують для пошуку фрагментів тексту, або для визначення лексичної структури мови програмування [1].

Класи КВ-граматик

Контекстно-вільні граматики представлені наступними класами:

- LL(k)
- LR(k)
- RL(k)
- RR(k)
- SLR, LALR (підкласи LR(1)-граматик, рис. 1),

де перша літера визначає напрямок читання вхідного потоку символів (L – зліва-направо, R – справа-наліво), друга — напрямок синтакстичного розбору (L – низхідний, R – висхідний), k – кількість символів, що аналізуються наперед.

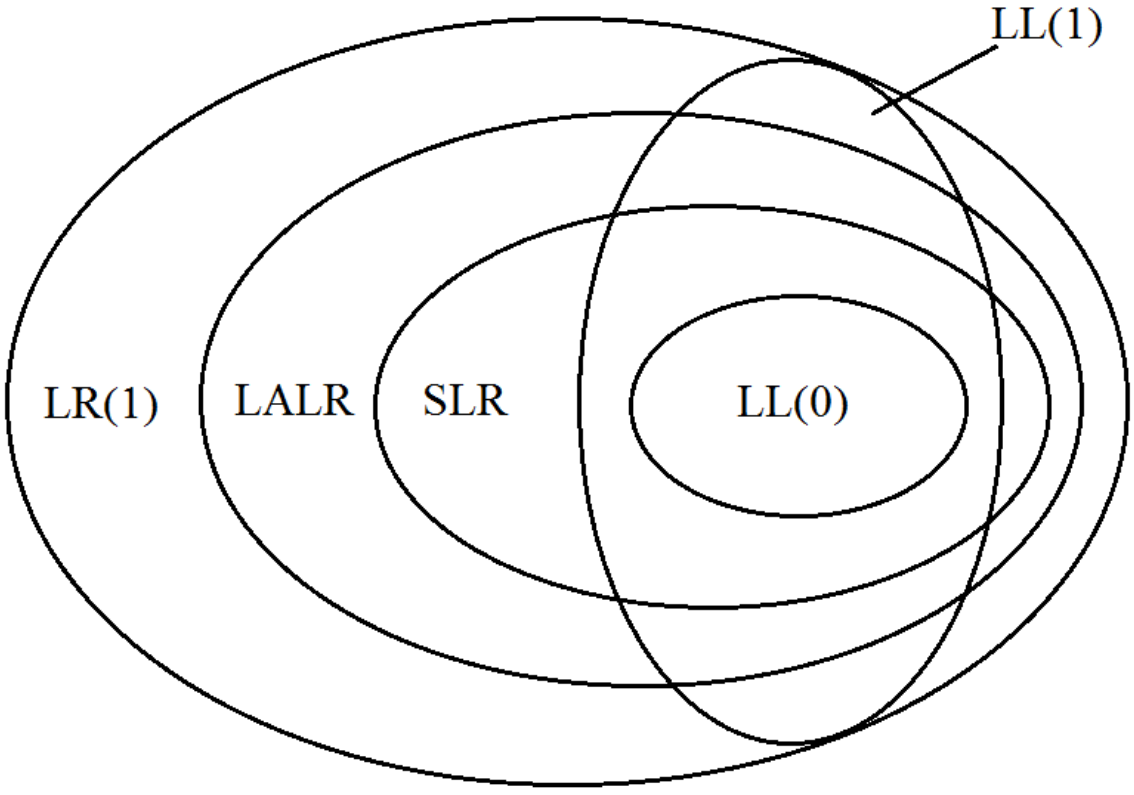


Рис. 1 Діаграма відношення потужностей основних класів КВ-граматик

LL(*)-граматика — граматика, що дозволяє розпізнати мову читаючи вхідний потік символів зліва-направо, алгоритмом низхідного синтаксичного розбору з довільною кількістю символів, що можуть бути проаналізовані.

Для класів КВ-граматик існують відповідні класи синтаксичних аналізаторів (рис. 2), що реалізують розбір мови за даною граматикою [2].

1.2. Класифікація синтаксичних аналізаторів

Рис. 2. Класифікація синтаксисичних аналізаторів

Низхідний метод синтаксичного розбору, що є відповідає LL-граматиці полягає у розборі вхідної послідовності починаючи від кореневого правила граматики (аксіоми).

Існує два основних варіанти реалізації низхідного синтаксичного аналізатора:

- передбачаючий парсер — аналізатор, що в разі необхідності вибору однієї з декількох альтернатив правила здатний однозначно вирішити, по якій альтернативі має розкриватися поточне правило. Такий вибір альтернативи (передбачення) здійснюється на основі аналізу k вхідних символів. Працює за лінійний час;
- парсер з поверненням — реалізація аналізатора, що при виборі альтернативи розкриває кожну альтернативу послідовно і, в разі помилки, повертається до місця розгалуження. Може мати експоненційний час роботи.

1.3. Інтеграція лексичного і синтаксичного аналізаторів

Виділяється два підходи до інтеграції лексичного і синтаксичного аналізаторів при розробці компіляторів:

- двопрохідний компілятор, що перетворює вхідний файл на заданій мові у послідовність лексем (перший прохід) та будує дерево розбору (або інше подання граматики) з отриманої послідовності за правилами граматики заданої мови;
- однопрохідний компілятор, при роботі якого основним є синтаксичний аналізатор, який при необхідності обробки нової лексеми делегує аналіз лексичному аналізатору, який виконує розбір вхідного тексту до кінця нової лексеми та повертає управління програмі синтаксичного аналізатора.

Перевагами двопрохідного компілятора є простота реалізації та можливість ітерації по послідовності лексем в процесі синтаксичного аналізу.

Недоліком алгоритму двопрохідного аналізу є менша швидкість роботи в порівнянні з однопрохідним та більші затрати пам’яті через необхідність зберігати одночасно структури даних для послідовності лексем та дерева розбору.

Можливим є зменшення затрат пам’яті при двопрохідному аналізі за рахунок звільнення пам'яті, виділеної під вже оброблені лексеми (за відсутності необхідності ітерації по послідовності). Однак, така реалізація призводить до збільшення складності алгоритму та зменшення його швидкодії.

Додатковою перевагою алгоритму однопрохідного аналізу є можливість об’єднання (повного або часткового) реалізації лексичного та синтаксичного аналізаторів, що дозволяє уніфікувати правила граматики, використовувати алгоритм синтаксичного аналізу для лексичного розбору. Такий підхід є менш ефективним, ніж використання окремої програми-лексера, але може бути зручним при розробці прототипів компіляторів. Даний підхід використаний в розробленому проекті для зменшення залежності від сторонніх бібліотек або готових продуктів.

Таким чином, для реалізації в проекті обрано алгоритм однопрохідного низхідного аналізу.

1.4. Автоматизація розробки синтаксичних аналізаторів

Виділяється два основних метода автоматизації генерації синтаксичних аналізаторів:

- генератори компіляторів
- парсер-комбінатори

Генератор компіляторів (compiler-compiler/compiler generator) – програмний комплекс, що призначений для створення парсера,

інтерпретатора або компілятора з певного виду формального опису граматики, зазвичай — БНФ або РБНФ.

Парсер-комбінатор — функція вищого порядку, що приймає на вхід декілька парсерів та повертає новий парсер, що є комбінацією вхідних. Зазвичай, парсер-комбінатори дозволяють описувати парсери за допомогою мови програмування, без необхідності використовувати додаткові види нотації [3].

Парсер-комбінатор є більш простим і менш потужним інструментом в порівнянні з генератором парсерів і використовується для обробки предметно-орієнтованих мов програмування.

Недоліки парсер-комбінаторів:

- менша потужність та гнучкість (підтримують лише низхідний аналіз LL(1) граматик)
- менша швидкість роботи
- процес розробки не піддається автоматизації

Оскільки генератори компіляторів позбавлені таких недоліків, для реалізації проекту обрано алгоритм їх роботи.

1.5. Огляд існуючих аналогів

До існуючих СПТ належать генератори лексерів Flex, lex, re2c, генератори парсерів Yacc, Bison, Parsec та інші. Програмні комплекси JavaCC та ANTLR здатні генерувати як лексичну, так і синтаксичну частину аналізаторів.

Flex та lex являють собою генератори лексичних аналізаторів мовою C. Зазвичай використовуються разом з Yacc/Bison.

re2c – генератор лексичних аналізаторів для мови C.

Parsec – бібліотека для створення парсер-комбінаторів мовою Haskell. Існують версії для мов Erlang, Elixir, OCaml, F#, C# та Java.

Yacc (Yet Another Compiler-Compiler) – генератор LALR парсерів для операційної системи Unix.

JavaCC (Java Compiler Compiler) – генератор лексичних та синтаксичних аналізаторів, написаний мовою Java. Підтримує LL(k) граматики записані за допомогою EBNF.

ANTLR (Another Tool For Language Recognition) - генератор лексерів та парсерів для LL(*) граматик описаних за допомогою EBNF [4].

Спільною рисою перелічених програм є відсутність графічного інтерфейсу користувача, що ускладнює процес роботи з ними для недосвідчених користувачів.

1.6. Обґрунтування теми дипломного проекту

Розвиток IT галузі, поява принципово нових задач та завдань призводить до необхідності розробки більш зручних, швидких та ефективних інструментів, в тому числі — нових мов програмування.

Ці нові мови все більше відходять від «низького» рівня програмування, прямуючи до більш «високого», використовуючи нові конструкції та підходи, зменшуючи тим самим об’єм програмного коду при написанні програм. Використання нових конструкцій, збільшення рівня абстракції досягається ускладненням граматик мов програмування, а також ускладненням програм-трансляторів. Виникає потреба в спрощенні процесу розробки цих програм.

Нові мови програмування вимагають розробки нових трансляторів/компіляторів та покращення існуючих. Наявність у мові високорівневих конструкцій потребує обробки складних граматичних конструкцій, тому з кожним роком розробка та підтримка компіляторів ускладнюється. Системи побудови трансляторів (системи генерації лексичних та синтаксичних аналізаторів) дають змогу автоматизувати, спростити та пришвидшити процес розробки.

На сьогоднішній день існує багато систем, що дозволяють згенерувати синтаксичний аналізатор по деякій граматиці. Вони називаються системам побудови трансляторів. Вони приймають на вхід граматику мови (наприклад, LR(1), LL(*), LALR тощо), описану в деякій визначеній формі (наприклад, БНФ або РБНФ), і результатом їх роботи є згенерований деякою мовою програмування (C, Java та ін.) синтаксичний аналізатор для цієї граматики.

Оскільки створення СА за допомогою СПТ є значно швидшим і набагато менш трудомістким процесом, який потребує лише правильно заданої граматики, СПТ набули досить широкого застосування.

2. РОЗРОБКА КОМПОНЕНТІВ СИСТЕМИ

2.1. Структура програмного комплексу

Розроблений комплекс може бути розділений на чотири логічні частини:

- компоненти, що відповідають за обробку вхідної граматики та генерацію лексичного та синтаксичного аналізаторів
- компоненти графічного інтерфейсу користувача
- демонстраційний модуль
- тестова система

Структура взаємозв’язків компонентів системи представлена на додатку 1. Алгоритм роботи програми представлений на додатку 2.

2.2. Модуль аналізу граматики

Структура модуля аналізу граматики розробленої програми представлена на рис. 3.

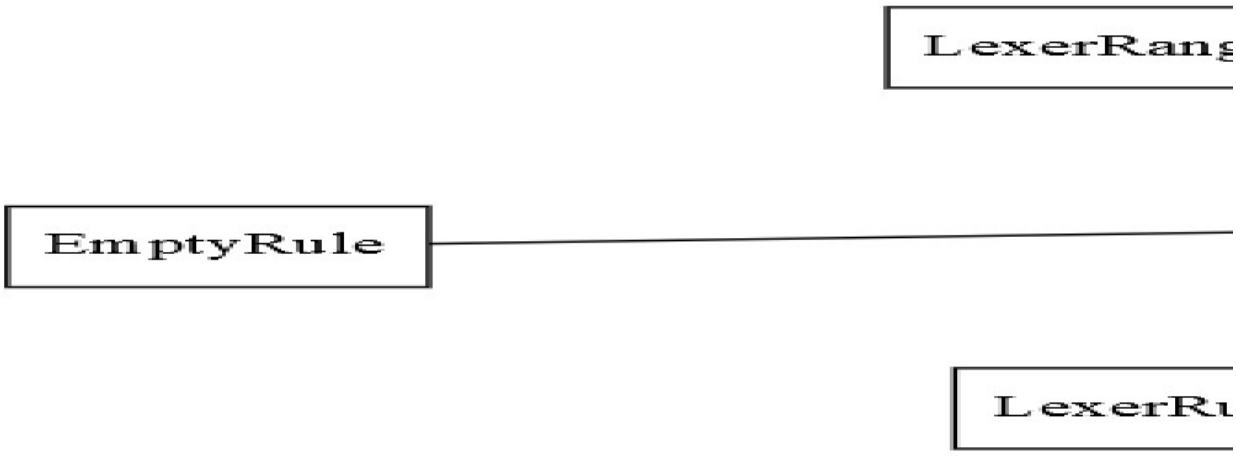


Рис. 3. Взаємозв’язки класів, що входять до модуля аналізу граматики

Клас com.blacksun.GrammarGen відповідає за процес обробки вхідного файлу, виокремлення окремих правил граматик та перетворення їх у внутрішнє подання.

Структура граматики представлена наступними класами:

- com.blacksun.utils.rule.RuleSet – представлення правила граматики, що складається з імені правила, його типу та всіх можливих альтернатив.

Підтримується три типи правил:

- PART – правило граматики, що відповідає за розбір складової частини термінального символу. Виконує аналіз одного символа з вхідної послідовності, не створює нових структур даних.
- LEXER – правило, що відповідає за розбір термінального символу та, в разі відсутності помилок аналізу, створює програмне представлення термінального символу.
- PARSER – правило, що є внутрішнім поданням синтаксичної частини граматики (нетермінального символу).

Тип правила задається перед його іменем в файлі граматики. Тип за замовчуванням - PARSER.

Правило вигляду lexer rule1 → rule1_1 | rule1_2 | rule1_3 ;

буде представлено екземпляром класу RuleSet з типом LEXER, іменем rule1 та списком альтернатив [rule1_1, rule1_2, rule1_3].

- com.blacksun.utils.rule.RuleAlternative — представлення альтернативи правила. Складається з послідовності нетерміналів.
- com.blacksun.utils.rule.Rule — базовий клас для представлення нетермінального символу. Реалізує шаблон “Abstract factory” (рис. 4), надаючи інтерфейс для створення конкретної реалізації даного класу при розборі граматики (табл. 1).

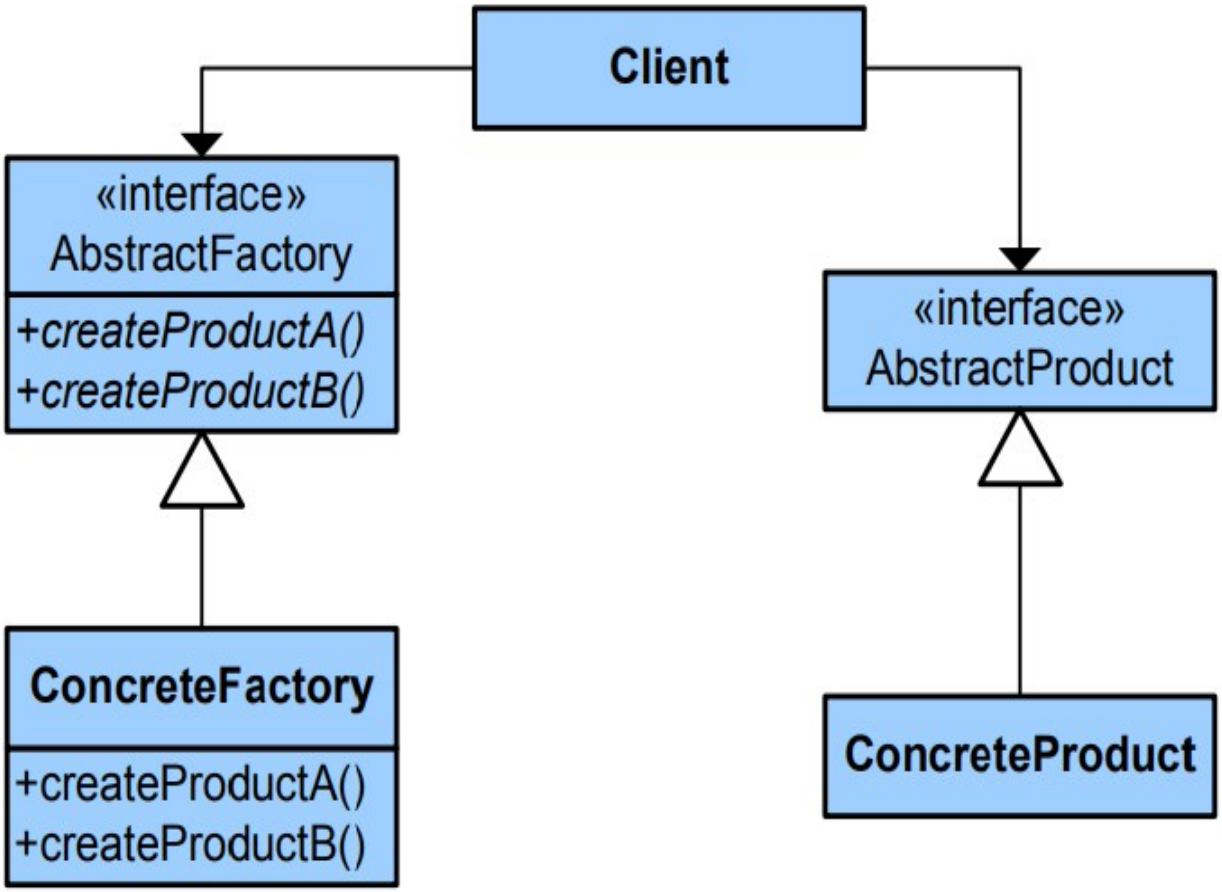


Рис. 4. UML діаграма структури шаблону проектування “Abstract factory”

Ім’я класу	Опис	Приклад
LexerRule	Представлення терміналу, що заданий конкретним символом або його кодом (ASCII або UTF-8)	part <A-rule> → ‘A’; lexer <space> → 32;
LexerRangeRule	Представлення терміналу, що заданий діапазоном символів або їх кодів	lexer <digit> → ‘0’..‘9’; lexer <digit> → 48.. 57;
PartRule	Представлення нетермінального символу	<rule1> → <rule2>;
EmptyRule	Представлення порожньої альтернативи	<empty-rule> → ;

KeywordRule	Представлення терміналу, що заданий конкретною лексемою	<if-rule> → IF ;

Табл. 1. Реалізації класу Rule

2.3. Реалізація лексичного аналізатора

Виділяється два підходи до інтеграції лексичного і синтаксичного аналізаторів при розробці компіляторів:

- двопрохідний компілятор, що перетворює вхідний файл на заданій мові у послідовність лексем (перший прохід) та будує дерево розбору (або інше подання граматики) з отриманої послідовності за правилами граматики заданої мови;
- однопрохідний компілятор, при роботі якого основним є синтаксичний аналізатор, який при необхідності обробки нової лексеми делегує аналіз лексичному аналізатору, який виконує розбір вхідного тексту до кінця нової лексеми та повертає управління програмі синтаксичного аналізатора.

Перевагами двопрохідного компілятора є простота реалізації та можливість ітерації по послідовності лексем в процесі синтаксичного аналізу.

Недоліком алгоритму двопрохідного аналізу є менша швидкість роботи в порівнянні з однопрохідним та більші затрати пам’яті через необхідність зберігати одночасно структури даних для послідовності лексем та дерева розбору.

Можливим є зменшення затрат пам’яті при двопрохідному аналізі за рахунок звільнення пам'яті, виділеної під вже оброблені лексеми (за відсутності необхідності ітерації по послідовності). Однак, така реалізація призводить до збільшення складності алгоритму та зменшення його швидкодії.

Додатковою перевагою алгоритму однопрохідного аналізу є можливість об’єднання (повного або часткового) реалізації лексичного та синтаксичного аналізаторів, що дозволяє уніфікувати правила граматики, використовувати

алгоритм синтаксичного аналізу для лексичного розбору. Такий підхід є менш ефективним, ніж використання окремої програми-лексера, але може бути зручним при розробці прототипів компіляторів. Даний підхід використаний в розробленому проекті для зменшення залежності від сторонніх бібліотек або готових продуктів.

Таким чином, для реалізації в проекті обрано алгоритм однопрохідного низхідного аналізу.

Для підтримки роботи програми в режимі однопрохідного аналізу при використанні синтаксичного аналізатора в якості лексичного було розроблено обгортку, що надає можливість більш зручного використання алгоритму синтаксичного аналізу в роботі лексера.

Клас com.blacksun.Lexer представляє базову реалізацію абстрактного лексичного аналізатора; надає можливість генерації більш інформативних повідомлень про лексичні помилки; надає можливість використання аналізатора як для вхідного рядку з текстом, так і для аналізу текстових файлів; надає методи обробки коментарів для граматик, що представляють собою опис мови програмування.

Лексична частина граматики представлена класами LexerRule, LexerRangeRule та KeywordRule (табл. 1). Дані класи є програмним поданням термінальним символів мови. В процесі лексичного аналізу вхідна послідовність символів аналізується та представляється конкретним класом, що дозволяє полегшити подальший синтаксичний аналіз.

Обробка правил, що мають декілька альтернатив відбувається за допомогою таблиць передбачення (lookahead table) та, за необхідності, – за допомогою алгоритму аналізу з поверненням.

2.4. Алгоритм роботи синтаксичного аналізатора

Вхідними даними для синтаксичного аналізатора є внутрішнє подання граматики, аксіома граматики та вхідна послідовність символів.

Реалізація синтаксичного аналізатора дозволяє використання в якості аксіоми граматики будь-якого правила, що належить даній граматиці. Такий підхід надає можливість розбору та аналізу конкретної конструкції мови та полегшує модульне тестування розробленої системи. Додатковою перевагою даного підходу є можливість одночасного використання при аналізі декількох незалежних граматик, що може бути корисним при одночасному розборі та аналізі "основної" мови програми та мов, що використовуються в якості рядкових літералів (SQL-запити, описові конструкції DSL). Таким чином, розроблена програма аналізатора може бути використана в ширшому колі задач, що мають специфічні вимоги до програмного забезпечення.

Виділяється два основних метода автоматизації генерації синтаксичних аналізаторів:

- генератори компіляторів
- парсер-комбінатори

Генератор компіляторів (compiler-compiler/compiler generator) – програмний комплекс, що призначений для створення парсера, інтерпретатора або компілятора з певного виду формального опису граматики, зазвичай — БНФ або РБНФ.

Парсер-комбінатор — функція вищого порядку, що приймає на вхід декілька парсерів та повертає новий парсер, що є комбінацією вхідних. Зазвичай, парсер-комбінатори дозволяють описувати парсери за допомогою мови програмування, без необхідності використовувати додаткові види нотації [3].

Парсер-комбінатор є більш простим і менш потужним інструментом в порівнянні з генератором парсерів і використовується для обробки предметно-орієнтованих мов програмування.

При аналізі граматики для кожного її правила визначається список термінальних символів, якими може починатися дане правило. Цей список використовується для вибору правильної альтернативи для правил виду

A → B C | D E,

де A — нетермінальний, B, C, D, E – термінальні або нетермінальні символи мови.

При використанні для ініціалізації аналізу правила, що не є аксіомою граматики, множина термінальних символів, якими може починатися кожне правило обчислюється лише для правил, що можуть бути виведені з даного, що дозволяє обмежити використання обчислювальних ресурсів та прискорити роботу аналізатора.

Кожен клас, що представляє символ граматики реалізує метод аналізу вхідного символу і визначення, чи відповідає він поточному правилу та метод обробки помилки розбору (генерація відповідного повідомлення, відновлення після помилки або збереження інформації про помилку для алгоритму розбору з поверненням). Реалізація методів залежить від типу правила, його вигляду та режиму роботи аналізатора.

Так, екземпляр класу Rule, що представляє правило

<single-digit> → ‘0’ .. ‘9’;

визначає, чи є доступним для аналізу поточний символ з вхідної послідовності, за необхідності зчитує символ, перевіряє, чи входить поточний символ в допустимий діапазон значень, в разі успішності перевірки створює екземпляр лексеми, іменем якої є ім’я поточного правила, а значенням — значення поточного символу. Додатково лексема зберігає позицію символу у вхідній послідовності (дані можуть бути використані для генерації інформативних повідомлень про помилки). В разі, якщо поточний символ не задовольняє правило, відбувається генерація повідомлення про помилку, що містить інформацію про поточну позицію аналізатора у вхідній

послідовності, ім’я поточного символу та ім’я очікуваного символу (символів).

При роботі в режимі передбачаючого аналізу додатково відбувається визначення альтернативи, яка починається з даного символу та виклик методу її розбору. Повідомлення про помилку в такому випадку генерується за відсутності альтернативи, що здатна розпізнати поточний символ.

2.5. Компоненти графічного інтерфейсу

При розробці графічного інтерфейсу програми було використано стандартну графічну бібліотеку мови Java – Swing, компоненти якої містяться в пакетах javax.swing та java.awt.

Компоненти графічного інтерфейсу програми представлені пакетом com.blacksun.gui, що містить наступні класи:

- MainFrame — головне вікно програми, в якому розміщені всі графічні компоненти (рис. 5);
- InputPanel — вікно відображення вмісту вхідного файлу, що аналізується;
- OutputPanel — вікно відображення результатів аналізу (дерева розбору) (рис. 6-8);
- ErrorFrame — вікно для виведення повідомлень про виявлені помилки (рис. 9);
- Toolbar — панель інструментів, що надає інтерфейс для роботи з файловою системою користувача та компонентами розробленої системи (рис. 5, 10);
- TreePanel — компонент, що відповідає за відображення дерева розбору в інтерактивному вигляді (за замовчуванням використовується текстове представлення) (рис. 7-8);
- SettingsFrame — вікно налаштувань системи (рис. 10).

Рис. 5. Головне вікно програми, панель інструментів (зверху)

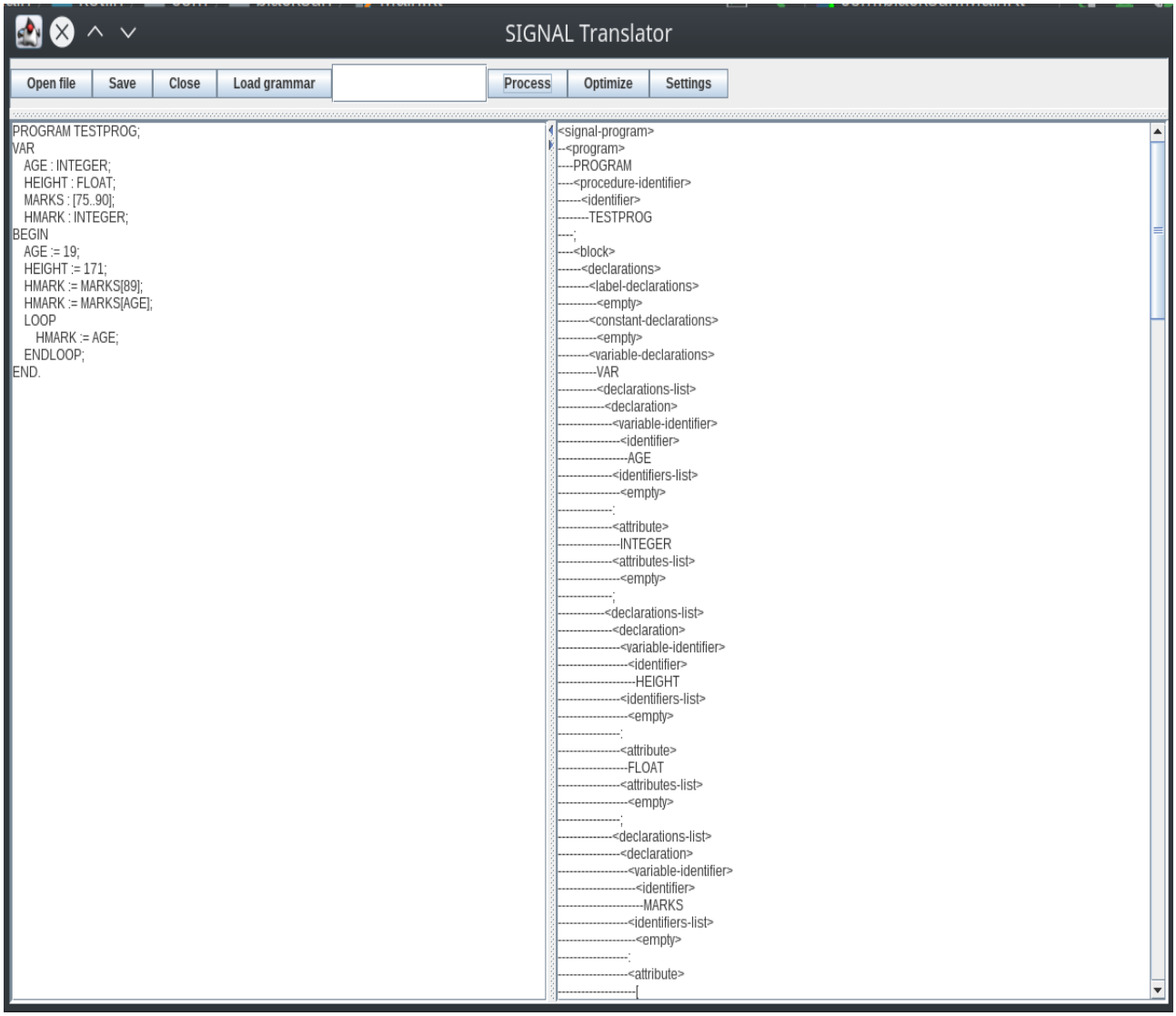


Рис.6. Вікно відображення вхідного файлу (зліва) та текстове подання дерева розбору (справа)

Вікно відображення вхідного файлу дозволяє переглядати та змінювати файл, що аналізується. Це збільшує наочність роботи та пришвидшує роботу над помилками у вхідному коді. Можливим є використання вікна для редагування граматики, що дозволяє швидко протестувати зміни у граматиці та їх вплив на побудову дерева розбору для конкретних вхідних даних.

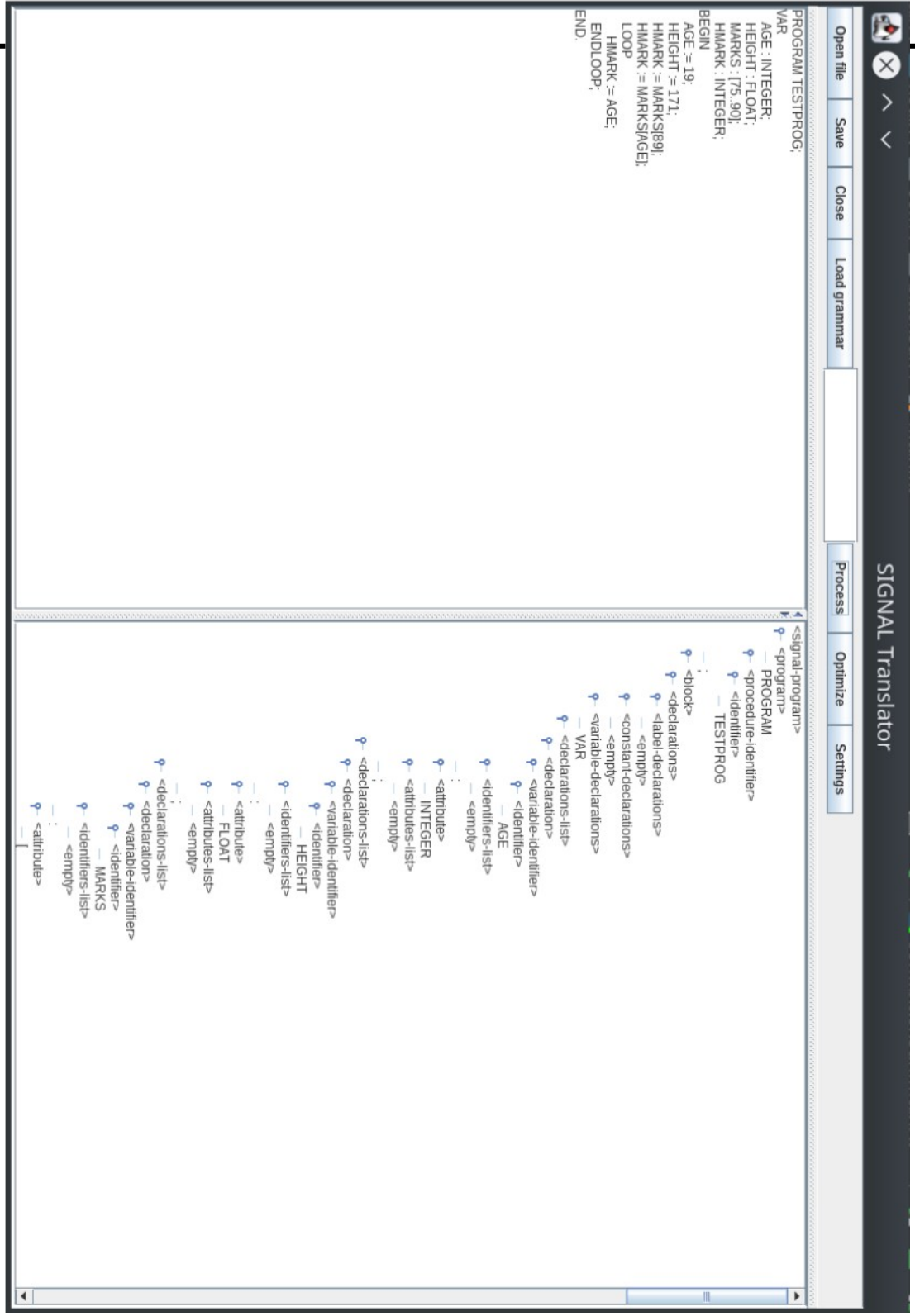


Рис. 7. Інтерактивне подання дерева розбору (розгорнутий вигляд)

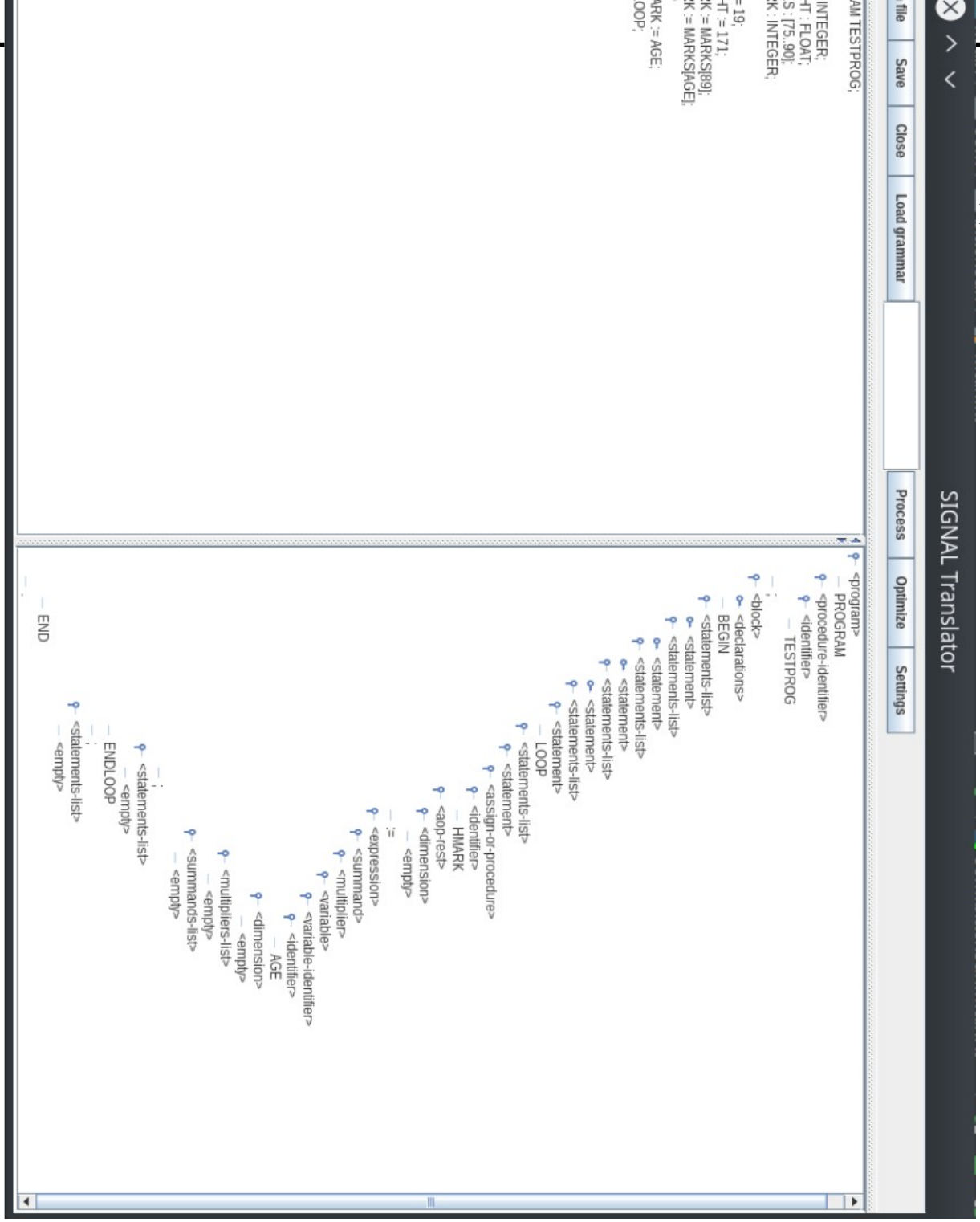


Рис. 8. Інтерактичне подання дерева розбору з відображенням частини дерева

Результат роботи програми (дерево розбору) може бути поданий у текстовому (рис. 6) та інтерактивному (рис. 7-8) вигляді. Текстове подання результату може бути передане як вхідні дані для інших програм, що приймають внутрішнє представлення конструкцій мови саме в такому вигляді. Інтерактивне подання є зручним для ручного аналізу отриманих

результатів, представлення структури програми у вигляді дерева та перегляду окремих частин дерева (конкретні вирази та конструкції мови).

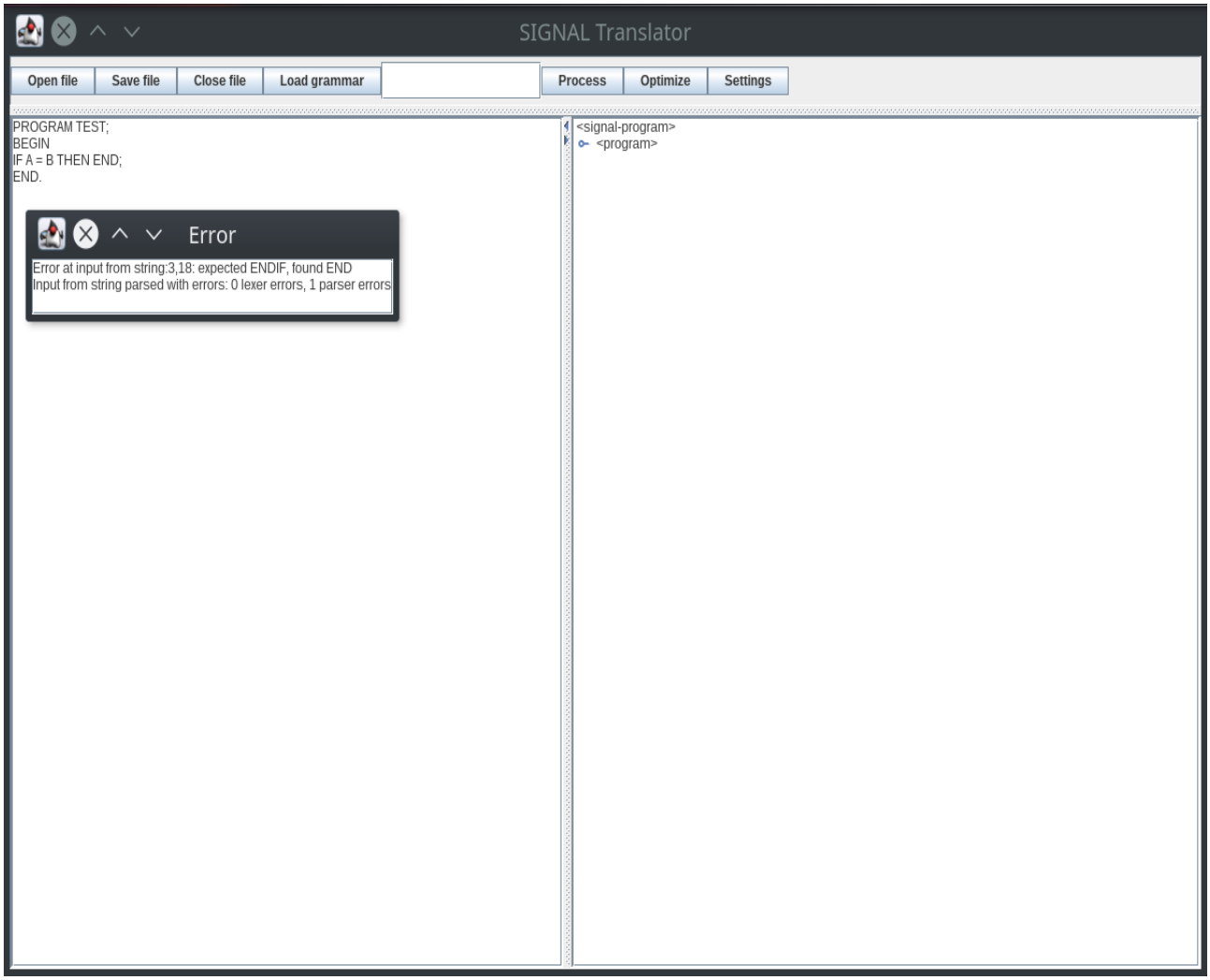


Рис. 9. Вікно з повідомленням про синтаксичну помилку

Рис. 10. Панель інструментів у вигляді піктограм та вікно налаштувань

Пакет com.blacksun.gui.util містить класи, що реалізують інтеграцію стандартних графічних компонентів бібліотеки Swing та налаштувань проекту. Пакет містить наступні класи:

- LFrame — реалізація вікна з підтримкою багатомовного інтерфейсу;
- LIcon — реалізація кнопки, що підтримує багатомовний інтерфейс та може бути відображена як у текстовому вигляді, так і за допомогою
- заданої піктограми;
- LImage — реалізація піктограми, що розроблена для зручної роботи з ресурсами проекту;
- LCheckBox — компонент, що представляє перемикач з багатомовним інтерфейсом;

- ForceCheckBox — представляє розширену версію класу LCheckBox, що підтримує роботу з ресурсами та налаштуваннями проекту;
- SFileChooser — компонент, що реалізує роботу з файловою системою користувача.

На рис. 11-12 представлено вигляд вікна програми та вікна налаштувань при зміні мови інтерфейсу.

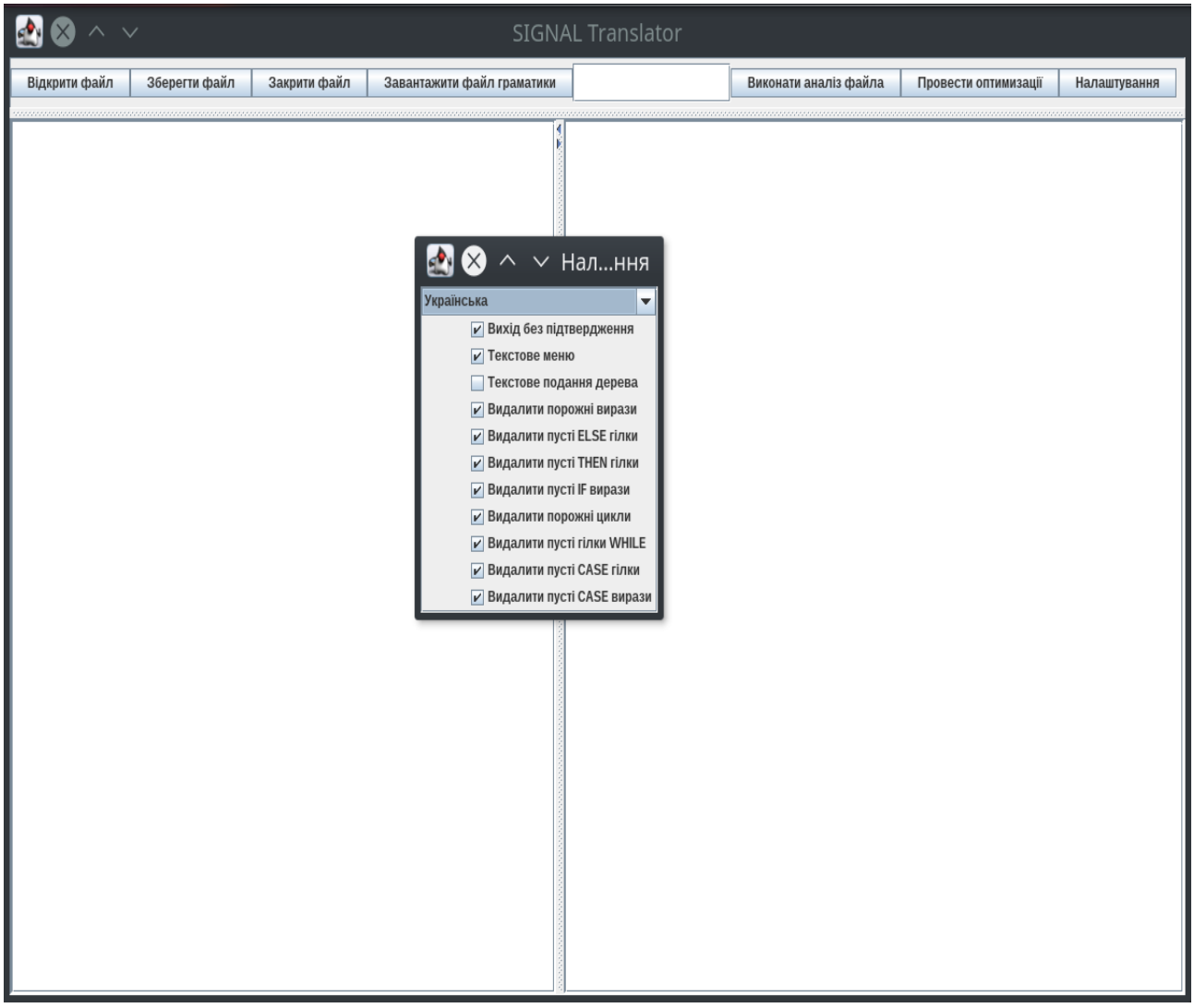


Рис. 11. Україномовний інтерфейс користувача

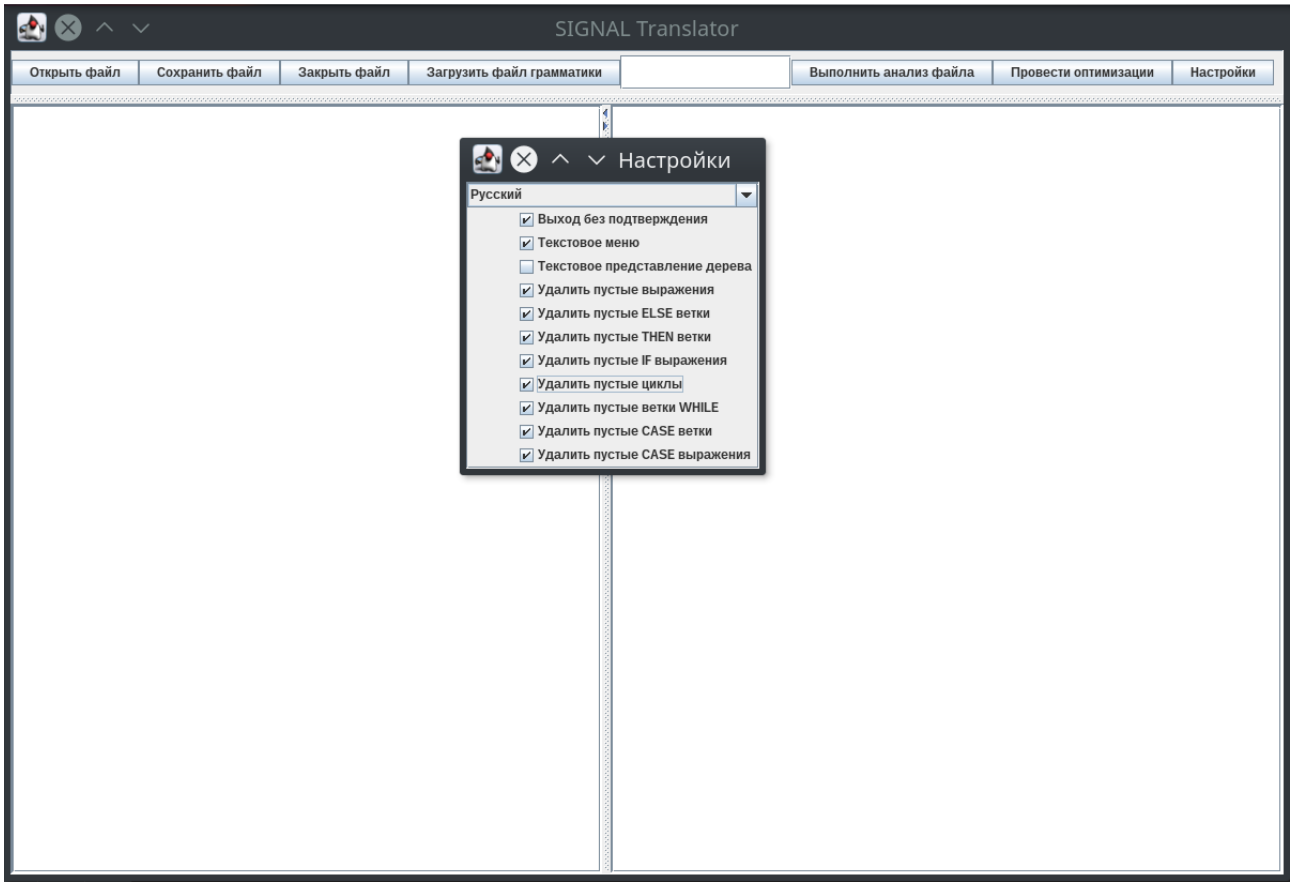


Рис. 12. Російськомовний інтерфейс користувача

2.6. Модуль налаштувань проекту

Клас com.blacksun.settings.Settings відповідає за роботу з параметрами та налаштуваннями проекту, забезпечує збереження налаштувань проекту між запусками програми.

Налаштування проекту зберігаються в директорії проекту ($HOME/.SIGNAL в ОС UNIX, $USER\Documents\ and\ Settings\.SIGNAL в ОС Windows, може бути змінено шляхом конфігурації проекту) у вигляді CSV файлу. Клас Settings надає інструменти для ініціалізації налаштувань, зміни налаштувань в ході роботи програми, відновлення налаштувань із зовнішнього файлу та збереження (запису) налаштувань у відповідний файл. Оскільки поточна реалізація налаштувань використовує лише скалярні дані, відсутня необхідність реалізації складної логіки серіалізації комплексних

структур даних, тому збереження даних у CSV форматі дозволяє зменшити витрати ресурсів (обчислювальної потужності та пам’яті) при роботі програми. Налаштування задаються парою <ключ, значення>.

Клас com.blacksun.settings.Force реалізує методи, що забезпечують можливість роботи з налаштуваннями, що відповідають підтвердженню певних дій користувача (вихід з програми без додаткового підтвердження, виконання певної оптимізації коду, автоматичне збереження файлів при закритті програми тощо). Методи класу забезпечують перевірку правильності значення налаштування, в разі використання помилкового значення (використання в якості логічної змінної виразу, що має інший тип даних) встановлюють значення за замовчуванням.

Клас com.blacksun.settings.Language забезпечує роботу з зовнішніми файлами, що представляють словники користувацького інтерфейсу. При ініціалізації екземпляру класу відбувається визначення списку доступних мов інтерфейсу та завантаження поточної активної мови. Кожна мова представлена окремим CSV файлом, що є набором пар <ім’я текстового літералу, що використовується в програмі — значення, що буде відображено користувачу при встановлені даної мови>. Кожен файл додатково містить змінну, що зберігає локалізовану назву мови.

Методи класу надають можливість зміни поточної мови та додавання словників нових мов в ході роботи програми, проте заміна значень текстових компонентів графічного інтерфейсу призводить або до наявності в інтерфейсі програми тексту кількома різними мовами, або до необхідності повного оновлення інтерфейсу, що вимагає значних витрат ресурсів. Таким чином, зміна мови інтерфейсу відбувається при перезапуску всієї програми, що є стандартною поведінкою для багатьох прикладних програм, що підтримують багатомовний інтерфейс користувача.

2.7. Структура та призначення основних методів класів проекту

Клас com.blacksun.GrammarGen:

- initGrammar(path: String): Unit — виконує аналіз файлу граматики, що заданий іменем;
- initGrammar(file: File): Unit — виконує аналіз файлу граматики;
- parseLine(line: String): Unit — виконує попередню обробку рядка, прочитаного з файлу граматики, додає оброблений рядок у внутрішний буфер, викликає метод parseRule в разі, якщо в буфері наявне нове повністю прочитане правило граматики;
- parseRule(): Unit - виконує перетворення тексту правила граматики у його внутрішнє подання;
- parseParts(part: String): Pair<String, String> - допоміжний метод визначення імені та типу поточного правила;
- parse(text: String, rule: String): Node — виклик аналізатора для розбору вхідної послідовності, заданої текстовим рядком, за заданим правилом (за замовчуванням — аксіома граматики);
- parse(file: File, rule: String): Node — аналогічно методу parse(text: String, rule: String), але вхідна послідовність міститься в заданому файлі;
- registerKeyword(name: String): Unit — додає вказане слово в список ключових слів граматики (для граматик, що задають мову програмування);
- savepoint(length: Int): Unit - створює точку, в яку аналізатор може повернутися в разі виникнення помилки розбору (для роботи в режимі аналізу з поверненням);
- rollback(): Unit - повернення на точку, створену методом savepoint в разі виникнення помилки (для роботи в режимі аналізу з поверненням).

Клас com.blacksun.Lexer

- init(text: String): Unit – ініціалізує компоненти класу для роботи з вхідною послідовністю, що задана текстовим рядком. Виконується ініціалізація позиції аналізатора у вхідній послідовності, скидання лічильника помилок та видалення тексту останньої помилки;
- init(file: File): Unit – ініціалізує компоненти класу для роботи з вхідною послідовністю, що міститься в заданому файлі, виконується аналогічно попередньому методу;
- read(): Int — читання коду нового символу з вхідної послідовності;
- add(): Unit – додавання останнього прочитаного символу до поточної лексеми;
- skip(): Unit – метод обробки символів, що не несуть смислової навантаження (пробільні символи);
- error(): Unit – метод генерації повідомлення про помилку;
- getToken(): Token — повертає поточну лексему;
- createTokenNode(): Node — створення нового вузла дерева розбору, значенням якого є поточна лексема;
- getTokenNode(): Node? - повертає поточний вузол дерева розбору (за його наявності);
- getErrors(): Int — повертає поточну кількість лексичних помилок;
- errorMsg(): String – повертає текст повідомлення про останню оброблену помилку.

Клас com.blacksun.utils.rule.Rule:

- parse: () -> Node — метод, що містить автоматично згенеровану реалізацію алгоритму розбору вхідної послідовності для даного правила граматики;
- computeFirst: () -> List<Int> - згенерований метод, що визначає множину кодів термінальних символів, з яких може починатися дане правило; використовується при роботі в режимі передбачаючого аналізу;

- computeNames: () -> List<String> - згенерований метод, що визначає множину правил, що безпосередньо виводяться з даного; використовується при роботі в режимі аналізу з поверненням;
- check(value: Any): Boolean — згенерований метод, що визначає, чи може поточний символ вхідної послідовності бути проаналізованим та обробленим за допомогою даного правила;
- toString(): String — повертає текстове представлення класу.

Всі класи пакету com.blacksun.utils.rule містять аналогічні методи та відрізняються лише конкретною реалізацією, що залежить від подання правила в заданій граматиці.

Клас com.blacksun.utils.Token є представленням лексеми та має наступні методи:

- constructor(name: String) — метод, що відповідає за створення екземпляру класу з заданим початковим значенням лексеми;
- operator plusAssign(char: Int): Unit — метод, що дозволяє додати новий символ до значення лексеми;
- equals(other: Any?): Boolean — метод визначення ідентичності двох лексем або лексеми та її текстового подання;
- name(): String — повертає значення поточної лексеми у вигляді текстового рядка;
- toString(): String — текстове подання лексеми, за замовчуванням — еквівалент методу name.

Клас com.blacksun.utils.node.Node є реалізацією вузла дерева розбору та має наступні методи:

- print(tab: String, depth: Int): Unit – виводить дерево розбору в текстовому вигляді (рис. 5);

- operator plusAssign(node: Node): Unit – додає заданий вузол у піддерево;
- operator plusAssign(nodes: List<Node>): Unit - додає всі задані вузли у піддерево;
- operator get(index: Int): Node — повертає вузол піддерева за його індексом;
- traverse(func: (Node) -> Unit, pre: Boolean, stop: (Node) -> Boolean, filter: (Node) -> Boolean) — абстрактний метод обходу дерева; викликає функцію func на кожному вузлі, що задовільняє умову, задану функцією filter; функція stop є предикатом завершення обходу поточного піддерева, прапорець pre відповідає за порядок обходу (прямий чи зворотній);
- traversePre(func: (Node) -> Unit, stop: (Node) -> Boolean, filter: (Node) -> Boolean) — реалізація алгоритму обходу дерева у прямому порядку;
- traversePost(func: (Node) -> Unit, stop: (Node) -> Boolean, filter: (Node) -> Boolean) — реалізація алгоритму обходу дерева у зворотньому порядку;
- toString(): String — текстове представлення класу.

Клас com.blacksun.gui.Main є основним класом модуля графічного інтерфейсу користувача і реалізує основні методи для інтеграції графічного інтерфейсу та логічної частини аналізатора. Методи класу:

- init — конструктор, що встановлює початковий стан вікна та відповідає за створення графічних компонентів;
- windowDeiconified(p0: WindowEvent?): Unit — метод, що визначає поведінку вікна при згортанні вікна. За замовчування поведінка вікна відповідає поведінці класу вікна з стандартної графічної бібліотеки, додаткові дії не виконуються;
- windowIconified(p0: WindowEvent?): Unit — метод, що визначає поведінку вікна при розгортанні вікна. За замовчування поведінка вікна

відповідає поведінці класу вікна з стандартної графічної бібліотеки, додаткові дії не виконуються;

- windowOpened(p0: WindowEvent?): Unit — метод, що визначає поведінку вікна при першому відкритті вікна. За замовчування поведінка вікна відповідає поведінці класу вікна з стандартної графічної бібліотеки, додаткові дії не виконуються;
- windowClosed(p0: WindowEvent?): Unit — метод, що визначає поведінку вікна після закриття вікна. За замовчування поведінка вікна відповідає поведінці класу вікна з стандартної графічної бібліотеки, додаткові дії не виконуються;
- windowActivated(p0: WindowEvent?): Unit — метод, що визначає поведінку вікна при встановленні вікна активним в системі. За замовчування поведінка вікна відповідає поведінці класу вікна з стандартної графічної бібліотеки, додаткові дії не виконуються;
- windowDeactivated(p0: WindowEvent?): Unit — метод, що визначає поведінку вікна при встановленні вікна неактивним в системі. За замовчування поведінка вікна відповідає поведінці класу вікна з стандартної графічної бібліотеки, додаткові дії не виконуються;
- windowClosing(p0: WindowEvent?): Unit — метод, що визначає поведінку вікна при спробі закриття вікна. Підміняє стандартну поведінку вікна викликом методу close();
- close(): Unit — виконує перевірку на необхідність виведення вікна підтвердження виходу з програми, виводить вікно за необхідності, при підтвердженні виходу або за відсутності вікна здійснює вихід з програми;
- quit(): Unit — виконує звільнення ресурсів, виділених під компоненти графічного інтерфейсу, зберігає поточний стан програми (активний файл, активна граматика, обрана мова інтерфейсу та інші налаштування), закриває вікно програми;

- openFile(): Unit — створює вікно вибору файлу в файловій системі користувача, виконує читання файлу та запускає відображення вмісту файлу в області виводу;
- loadGrammar(): Unit — створює вікно вибору файлу в файловій системі користувача, виконує читання файлу граматики, встановлює обрану граматику в якості активної, ініціалізує компоненти аналізатора відповідно до завантаженої граматики. В разі наявності помилок аналізу ініціалізації граматики виводить на екран відповідне повідомлення;
- process(): Unit — виконує аналіз завантаженого вхідного файлу або введеного користувачем тексту за заданим правилом граматики (за його відсутності — за аксіомою граматики). В разі наявності помилок аналізу вхідного тексту видає на екран відповідне повідомлення. Виводить результат аналізу в область виводу у текстовому або інтерактивному вигляді, відповідно до поточних налаштувань програми;
- optimize(): Unit — запускає серію обраних оптимізації для вхідного тексту. В разі, якщо вхідні дані були оновленні після останнього аналізу, додатково запускає аналіз, що гарантує проведення оптимізації для актуальних даних. Виводить на екран повідомлення про помилки в разі їх наявності. Виводить результуюче дерево розбору в область виводу у вигляді, заданому поточними налаштуваннями програми;
- error(): Unit — виконує перевірку факту наявності або відсутності помилок, за їх наявності виводить на екран вікно з повідомленнями про виявлені помилки;
- save(): Unit — виконує збереження поточного вхідного тексту. В разі, якщо текст є вмістом файлу, відповідний файл перезаписується. За відсутності файлу на екран виводиться вікно вибору файлу, в який буде збережено текст;
- closeFile(): Unit — виконує закриття поточного активного файлу та очистку області виводу. В разу відсутності активного файлу відбувається

лише перевірка області виводу на наявність у ній вхідного тексту та його видалення за наявності.

Клас com.blacksun.gui.util.TreePanel реалізує подання дерева розбору в інтерактивному вигляді, надаючи засоби зручної роботи з результатами аналізу. Робота класу забезпечується наступними методами:

- init — виконує початкову ініціалізацію області відображення, розгортає всі елементи дерева розбору, що відображається в поточний момент;
- expandAllNodes(tree: JTree, startingIndex: Int, rowCount: Int): Unit — розгортає всі піддерева поточного елементу дерева. Параметри методу: tree — контейнер, що використовується для збереження та відображення дерева розбору, startingIndex — індекс рядка, що має бути розгорнутий, rowCount — поточна кількість рядків у відображенні дерева;

Клас com.blacksun.gui.util.TreePanel.NodeTreeModel є реалізацією інтерфейсу TreeModel, що відповідає за відображення даних у вигляді дерева. Кожен екземпляр даного класу є поданням елементу дерева розбору (термінального або нетермінального символу мови) при його виведенні на екран в інтерактивному вигляді. Реалізує поведінку для наступних стандартних методів:

- isLeaf(node: Any): Boolean — визначає, чи є даний вузол дерева листовим (вузлом, що не має піддерев);
- getChild(node: Any, index: Int): Any — за заданим індексом повертає вузол, що є піддеревом поточного вузла;
- getChildCount(node: Any): Int — повертає кількість піддерев поточного вузла.

Клас com.blacksun.gui.util.TreePanel.Renderer є реалізацією інтерфейсу TreeCellRenderer і відповідає за графічне представлення елементу дерева розбору. За замовчуванням відображує кожен вузол дерева у вигляді піктограми, що читається з файлів ресурсів проекту.

3. ТЕСТУВАННЯ СИСТЕМИ

Тестування розробленої системи реалізоване з використанням фреймворку JUnit4. Даний фреймворк забезпечує підтримку модульного тестування для мови Java.

Модульне тестування (unit testing) – метод тестування програмного забезпечення, що полягає в незалежному тестуванні кожного окремого модуля програми.

Модульне тестування вимагає ізоляції компонентів системи, що тестується, для зменшення зв'язності системи та виконання принципу "один тест має тестувати не більше одного компонента". У складних системах повна ізоляція компонентів може бути неможливою, що призводить до необхідності використання моків та стабів.

3.1. Модуль тестування

Модуль тестування розробленої системи представлений наступними класами:

- GrammarTest — відповідає за тестування компонентів аналізу граматики (класи GrammarGen та реалізації абстрактного класу Rule). Тестування відбувається шляхом ініціалізації тестової граматики (наведена в додатку 2), генерації її внутрішнього подання, розбору тестового файлу та порівнянням отриманих результатів (дерева розбору або повідомлення про помилку) з еталонними (очікуваними) результатами. Методи даного класу перевіряють коректність генерації внутрішнього подання правил граматики різного виду (з наявність альтернатив та без них, порожні правила, лексичні та синтаксичні правила, обробка коментарів тощо). Проводиться перевірка коректності обробки помилок та повідомлень про них.
- SignalGrammarTest — містить набір тестів, що перевіряють коректність роботи системи на даних, що максимально наближені до реальних. Процес тестування полягає в ініціалізації граматики мови

SIGNAL та аналізі тестових файлів, що представлені набором програм, написаних мовою SIGNAL. Оскільки всі тестові файли представляють собою робочі програми, умовою успішного проходження тестів є відсутність повідомлень про помилки. Даний підхід до тестування системи дозволяє швидко додати новий тест до складу модуля без необхідності внесення змін до алгоритму тестування.

Алгоритм тестування компонентів розробленої системи наведено на рис. 13.

3.2. Обробка повідомлень про помилки

Логування — процес виводу та/або збереження інформації про стан виконання програми. Логування в JVM-мовах виконується засобами стандартної бібліотеки або за допомогою сторонніх інструментів (Apache Log4j, SLF4J тощо). В даному проекті для реалізації логування процесу виконання програми та збереження інформації про стан програми на момент виникнення помилок використовується пакет java.util.logging стандартної бібліотеки мови Java. Він надає можливість перенаправлення повідомлень у консоль або у зовнішній файл у заданому форматі.

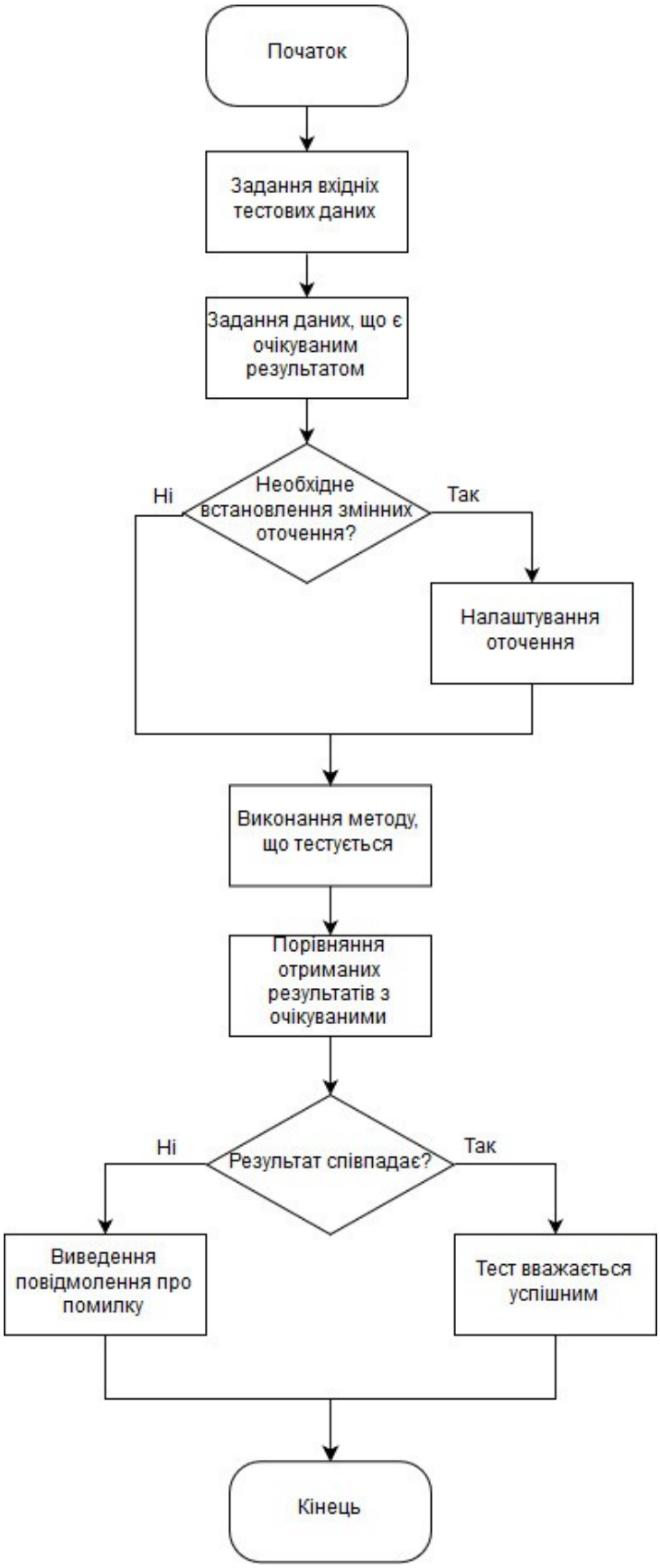


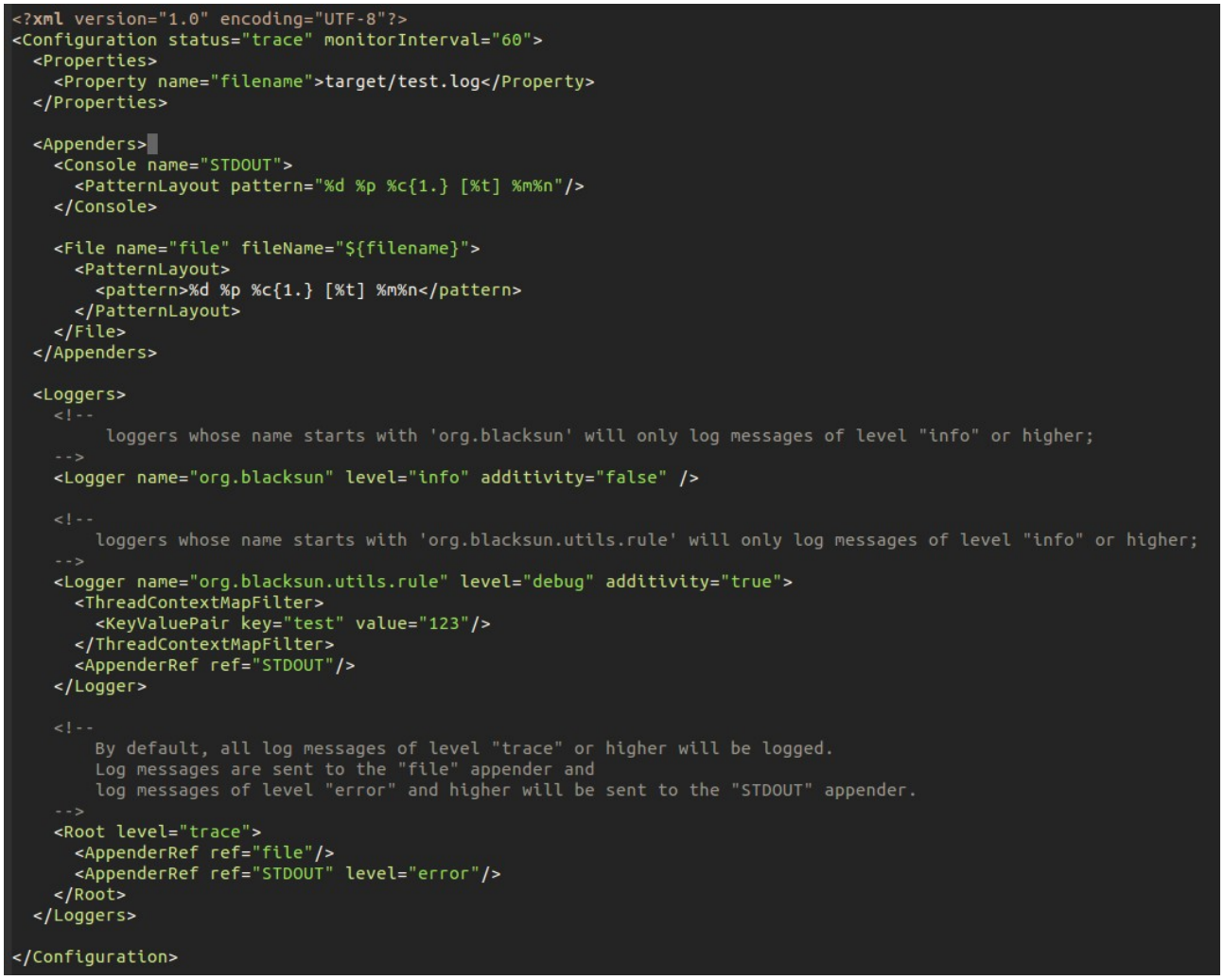
Рис. 13. Алгоритм тестування компоненту системи

Використання додаткових інструментів, що забезпечують логування процесу виконання програми дозволяє пришвидшити та значно полегшити процес розробки програмного забезпечення шляхом полегшення аналізу помилок, що виникають при роботі програми. Процес логування в розробленій системі контролюється встановленням різних рівніх інформаційних повідомлень, що мають бути відображені в ході роботи програми та гнучким налаштуванням засобів логування для кожного окремого модуля та класу проекту.

Основними способами налаштування процесу логування є конфігураційні класи, що реалізують опис налаштувань конструкціями мови, якою ведеться розробка, та використання зовнішніх інструментів конфігурації: XML та YAML файли, скриптові мови програмування тощо).

Використання окремих інструментів для конфігурації процесу логування є більш зручним методом, що забезпечує більшу гнучкість налаштування та більшу модульність проекту. Такі файли не вимагають перекомпіляції самого проекту при зміні налаштувань і можуть бути легко змінені або повністю замінені відповідно до поточних потреб розробки.

Налаштування процесу логування ходу роботи розробленого аналізатора забезпечується як засобами конфігураційних XML файлів, так і окремими класами проекту. XML файли забезпечується можливість встановлення форматів виводу повідомлень та дозволяють виділити окремі модулі, що потребують логування процесу роботи. Приклад конфігураційного файлу наведено на рис. 14. Додаткові класи проекту, що забезпечують власне процес логування, надають доступ до функцій та методів роботи з логуванням для інших класів та модулів проекту.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="trace" monitorInterval="60">
  <Properties>
    <Property name="filename">target/test.log</Property>
  </Properties>

  <Appenders>
    <Console name="STDOUT">
      <PatternLayout pattern="%d %p %c{1.} [%t] %m%n"/>
    </Console>

    <File name="file" fileName="${filename}">
      <PatternLayout>
        <pattern>%d %p %c{1.} [%t] %m%n</pattern>
      </PatternLayout>
    </File>
  </Appenders>

  <Loggers>
    <!--
        loggers whose name starts with 'org.blacksun' will only log messages of level "info" or higher;
    -->
    <Logger name="org.blacksun" level="info" additivity="false" />

    <!--
        loggers whose name starts with 'org.blacksun.utils.rule' will only log messages of level "info" or higher;
    -->
    <Logger name="org.blacksun.utils.rule" level="debug" additivity="true">
      <ThreadContextMapFilter>
        <KeyValuePair key="test" value="123"/>
      </ThreadContextMapFilter>
      <AppenderRef ref="STDOUT"/>
    </Logger>

    <!--
        By default, all log messages of level "trace" or higher will be logged.
        Log messages are sent to the "file" appender and
        log messages of level "error" and higher will be sent to the "STDOUT" appender.
    -->
    <Root level="trace">
      <AppenderRef ref="file"/>
      <AppenderRef ref="STDOUT" level="error"/>
    </Root>
  </Loggers>

</Configuration>
```

Рис. 14. Конфігураційний файл модуля логування

Клас com.blacksun.Logger представляє собою обгортку над стандартними класами пакету java.util.logging та відповідає за збереження інформації про процес роботи програми. Інформація може бути виведена як на екран (консоль або безпосередньо у вигляді компоненту графічного інтерфейсу), так і у окремий файл.

Основні методи, що забезпечують роботу з класом:

- info(msg: String): Unit – відповідає за обробку інформаційних повідомлень (процес аналізу граматики, процес розбору вхідної послідовності) (рис. 15);

- warn(msg: String): Unit – відповідає за обробку повідомлень про помилки, що не призводять до збоїв в роботі програми (помилка при розборі при роботі в режимі аналізу з поверненням) (рис. 16);
- err(msg: String): Unit – обробка повідомлень про всі інші види помилок (помилки аналізу граматики, лексичні та синтаксичні помилки) (рис. 17);
- setOutput(path: String): Unit – перенаправляє всі повідомлення у файл з зазначеним іменем;
- addOutput(path: String): Unit – додає новий файл для виводу помилок;
- removeHandlers(): Unit – відключає виведення помилок у файли.

```
INFO: Creating RuleSet <delimiters>
мая 27, 2019 7:44:51 PM com.blacksun.Logger info
INFO: Creating RuleAlternative <delimiter> <delimiter-list>
мая 27, 2019 7:44:51 PM com.blacksun.Logger info
INFO: Creating PartRule <delimiter>
мая 27, 2019 7:44:51 PM com.blacksun.Logger info
INFO: Creating PartRule <delimiter-list>
мая 27, 2019 7:44:51 PM com.blacksun.Logger info
INFO: Creating RuleSet <delimiter-list>
мая 27, 2019 7:44:51 PM com.blacksun.Logger info
INFO: Creating RuleAlternative <delimiter> <delimiter-list>
мая 27, 2019 7:44:51 PM com.blacksun.Logger info
INFO: Creating PartRule <delimiter>
мая 27, 2019 7:44:51 PM com.blacksun.Logger info
INFO: Creating PartRule <delimiter-list>
мая 27, 2019 7:44:51 PM com.blacksun.Logger info
INFO: Creating RuleAlternative <empty>
мая 27, 2019 7:44:51 PM com.blacksun.Logger info
INFO: Creating PartRule <empty>
мая 27, 2019 7:44:51 PM com.blacksun.Logger info
INFO: Creating RuleSet <if-stmt>
мая 27, 2019 7:44:51 PM com.blacksun.Logger info
INFO: Creating RuleAlternative IF <identifier> THEN <identifier>
мая 27, 2019 7:44:51 PM com.blacksun.Logger info
INFO: Creating KeywordRule IF
мая 27, 2019 7:44:51 PM com.blacksun.Logger info
INFO: Creating PartRule <identifier>
мая 27, 2019 7:44:51 PM com.blacksun.Logger info
INFO: Creating KeywordRule THEN
мая 27, 2019 7:44:51 PM com.blacksun.Logger info
INFO: Creating PartRule <identifier>
```

Рис. 15. Фрагмент логу з інформаційними повідомленнями

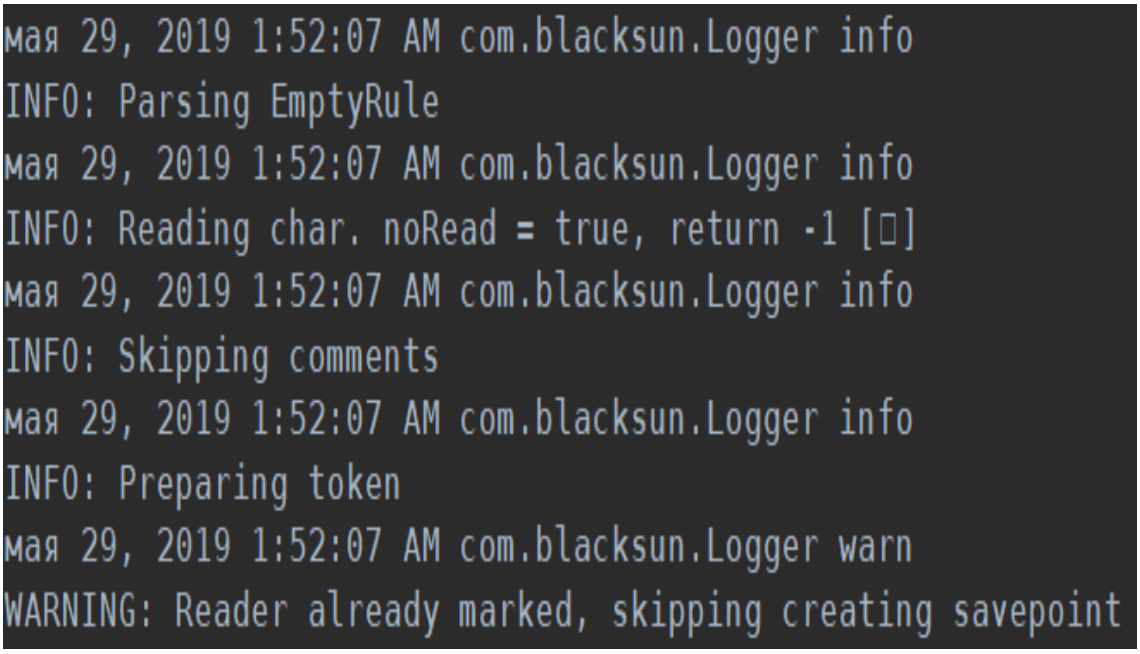


Рис. 16. Фрагмент логу з повідомленням про некритичну помилку

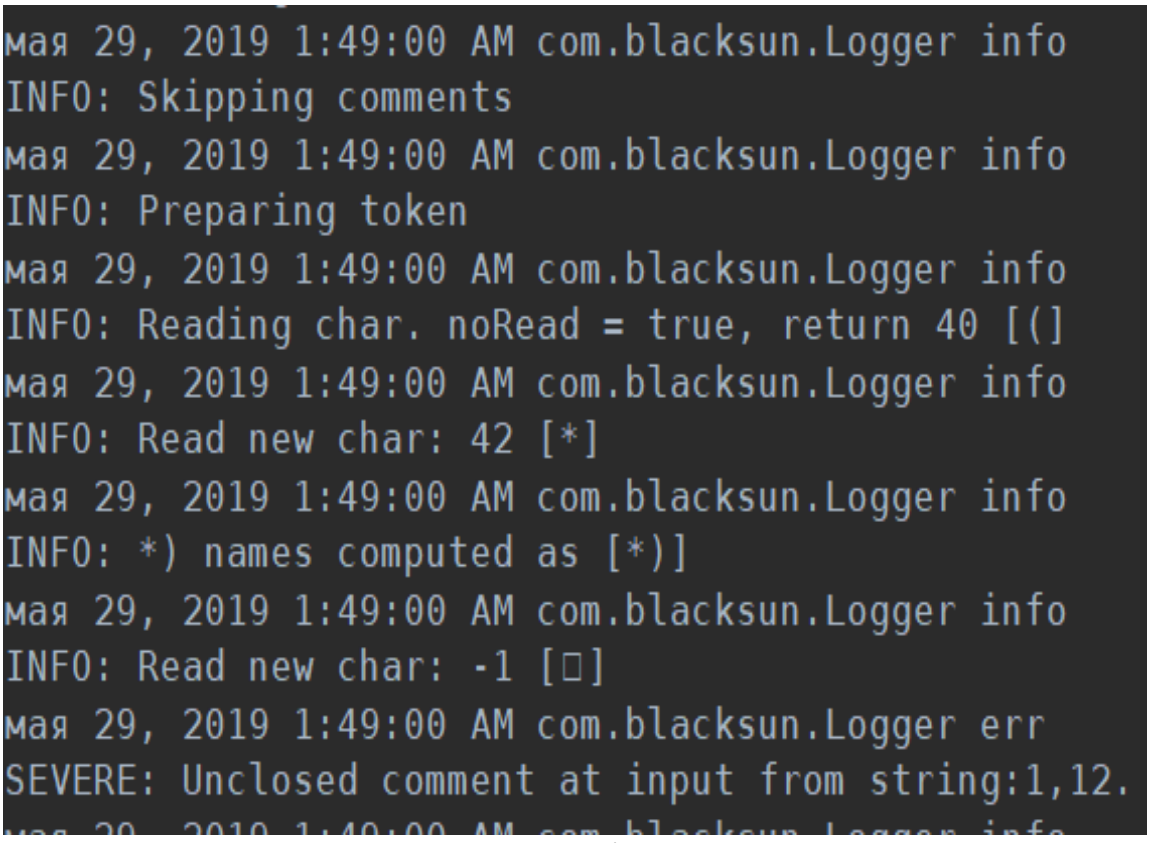


Рис. 17. Фрагменту логу з повідомленням про помилку

Наявність файлу з логом ходу виконання програми та інформацією про помилки та стан програми на момент їх появи дає змогу більш швидко та ефективно виявляти недоліки у роботі програми та виправляти їх. Інформація про хід виконання може бути корисною при аналізі ефективності роботи програми (її швидкодії) як у загальному випадку, так і при виявленні даних, що спричиняють значне вповільнення програми.

ВИСНОВКИ

Розроблений програмний комплекс дозволяє ознайомитись з будовою компілятора та принципами його роботи як в цілому, так і з його складовими частинами окремо. Він надає можливість експериментувати з різними модулями системи, і побачити, як проходить процес трансляції програми, які помилки можуть під час нього виникати, одразу після вводу своїх даних. Також є можливість покращити свої практичні навички в описі граматик мов програмування, а відповідно, і створення нових мов.

Системи побудови трансляторів дають змогу автоматизувати, спростити та пришвидшити процес розробки програмного забезпечення, що створюється у галузях, пов’язаних з розробкою нових мов програмування, підтримкою існуючих трансляторів, компіляторів та інтерпретаторів, тому розроблений проект може бути використаний при розробці зазначених програмних продуктів.

Переваги розробленого комплексу:

- можливість застосовувати на різних операційних системах;
- можливість використання сторонніх інструментів лексичного аналізу;
- можливість генерації власного лексичного аналізатора у вигляді, аналогічному синтаксичному аналізатору;
- можливість перевизначення методів обробки граматики, що дозволяє використання граматики заданої в будь-якому зручному форматі;
- наочність результатів виконання програми, наявність графічного інтерфейсу користувача;
- наявність логування процесу роботи програми.

Недоліки програми:

- лексичний аналізатор, що генерується розробленою програмою може бути менш ефективним, ніж спеціалізовані програми;
- згенерований аналізатор підтримує лише LL(*) граматики;

- відсутня можливість друку виводу та збереження коду згенерованих класів аналізатора;
- використовується досить простий алгоритм відновлення після помилок.

Можливі модифікації:

- покращення реалізації генератора лексичних аналізаторів;
- додавання підтримки інших видів граматик;
- модифікація алгоритму відновлення після помилок;
- додавання можливості паралельної обробки декількох файлів;
- можливість виводу та збереження згенерованого коду в окремий файл;

кешування граматики та/або згенерованих класів.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Noam Chomsky. Three models for the description of language. — 1956. — T. Vol.2. — С. 113–124. — (IRE Transactions on Information Theory)

2. Noam Chomsky. On certain formal properties of grammars. — 1959. — T. Vol.2. — С. 137–167. — (Information and Control)

3. Burge, William H. (1975). Recursive Programming Techniques. The Systems programming series. Addison-Wesley.

4. Parr, Terence (May 17, 2007), The Definitive Antlr Reference: Building Domain-Specific Languages (1st ed.), Pragmatic Bookshelf, p. 376, ISBN 978-0-9787392-5-6