НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
‹‹КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМ. ІГОРЯ СІКОРСЬКОГО››
Теплоенергетичний факультет
Кафедра автоматизації проектування енергетичних процесів і систем

КУРСОВА РОБОТА

З дисципліни ‹‹Лінгвістичне забезпечення САПР››

На тему:

«Мова програмування загального призначення Z99 та її імплементація»

Студента 3 курсу групи ТР-71
спеціальності 122: “Комп’ютерні науки”
Зуєва Михайла Олександровича
Керівник д.т.н., проф. Недашківський О.Л.

Кількість балів: _____ Оцінка: _____

Члени комісії: _____д.т.н., проф. Недашківський О.Л.
(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

Члени комісії: _____к.т.н., доц. Стативка Ю.І.
(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

Члени комісії: _____
(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

Київ – 2020

АНОТАЦІЯ

Курсову роботу виконано на 58 аркушах, вона містить три додатки та перелік посилань на використані джерела з шести найменувань. У роботі наведено шість рисунків та дев’ять таблиць.

Метою курсової роботи є розробка імперативної мови програмування загального призначення та її імплементація засобами мови програмування php 7.

Робота була виконана в у об’єктно орієнтованій парадигмі програмування. Для зручності використовувався пакет symfony/console, який встановлено через пакетний менеджер composer.

Ключові слова: формальна граматика, специфікація мови програмування, лексичний аналіз, таблиця символів, діаграма станів, синтаксичний аналіз, трансляція, постфіксна нотація.

ЗМІСТ

ВСТУП

1. Актуальність розробки та імплементації мов програмування

Сфера IT – одна з найбільш швидко розвиваючихся у світі. Кожен день виникає все більше різноманітних задач, які направлені на різні, часто вузькоспеціалізовані сфери, та вирішуються за допомогою комп'ютера. Тому виникає потреба у постійному удосконалені вже існуючих мов програмування, та створені нових. Також загальне розуміння внутрішнього устрою мови програмування дозволяє ефективніше її використовувати та допускати менше помилок при написанні програм. Тож можна сказати, шо розробка мови програмування – досі актуальна задача.

2. Завдання

Розробити та реалізувати транслятор з заданої мови, що включає наступні блоки:

- Лексичний аналізатор
- Синтаксичний аналізатор
- Побудова проміжної форми подання програми – польського інверсного запису (ПОЛІЗ)
- Виконання ПОЛІЗ

Мова програмування повинна містити в собі унарний мінус, обробку цілих та дійсних чисел: чотири арифметичні операції та піднесення до степеня, дужки, оператор присвоювання, оператори вводу та виводу, а також оператори умовного переходу:

```
if <лог. вираз> then <список операторів> fi
```

та циклу:

```
repeat <список операторів> until <лог. вираз>
```

3. Короткий опис роботи

У рамках курсової роботи, використовуючи php, було розроблено інтерпретатор pascal-подібної мови програмування Z99, який містить у собі

оператори вводу, виводу. умовний оператор, оператор циклу, а також основні арифметичні операції.

Спочатку була розроблена специфікація мови, яка містила у собі повний опис грамматики та поведінки різних операторів. На основі цієї специфікація було розроблено інтерпретатор, що складається з чотирьох компонентів:

1. Лексичний аналізатор, який виділяє потік токенів з вхідної програми
2. Синтаксичний аналізатор, який приймає потік токенів та будує синтаксичне дерево, грунтуючись на граматиці мови
3. Семантичний аналізатор, який будує таблиці ідентифікаторів, токенів і міток, а також виконує функції транслятора по перетворенню синтаксичного дерева у ПОЛІЗ-програму
4. Інтерпретатор, який виконую ПОЛІЗ-програму.

СПЕЦИФІКАЦІЯ МОВИ Z99

Для опису граматики мови Z99 використовується розширена форма Бекуса – Наура. Усі слова, що починаються з великої літери вважаються нетерміналами (нетермінальними символами). Ланцюжки, які знаходяться між одинарними, або подвійними лапками, або починаються з маленької літери – вважаються терміналами. Для візуалізації граматики використовуються синтаксичні діаграми Вірта.

Мета символ	Значення
=	Визначається як
|	Альтернатива
[x]	0 або 1 екземпляр
{ x }	0 або більше екземплярів x
(x | y)	Групування: будь-який з x або y
Zxy	Нетермінал
zxy	Термінал
`1`	Термінал
``1``	Термінал

Таблиця 1 - Прийнята нотація РБНФ

1. Лексика

У програмі мовою Z99 можуть використовуватись лексичні елементи, які можна поділити наступним чином: спеціальні символи, ідентифікатори, беззнакові цілі константи, беззнакові дійсні константи, логічні константи та ключові слова.

1.1 Спеціальні символи

До спеціальних символів належать арифметичні оператори: «+», «-», «*», «/», «^», оператори відношень: «==», «<=», «<», «>», «>=» , «<>», оператор присвоювання: «=», та знаки пунктуації: «.», «,», «:», «;».

1.2 Ідентифікатори

Ідентифікатор може складатись з літер та цифр, але починатися лише з літери. Довжина ідентифікатора не обмежена.

```
Ident = Letter { Letter | Digit }
```

Обмеження:

- Жоден ідентифікатор не може збігатись із ключовим (вбудованим, зарезервованим) словом.

Семантика:

- Якщо елемент може бути визначений і як ідентифікатор, і як ключове слово, то він вважається ключовим словом.
- Якщо елемент може бути визначений і як ідентифікатор, і як логічна константа, то він вважається логічною константою.

1.3 Константи

Лексичний аналізатор знаходить лише беззнакові цілі константи IntNum, беззнакові дійсні константи RealNum та логічні константи BoolConst.

```
IntNum = Digit { Digit }
RealNum = `.` UnsignedInt | UnsignedInt `.` [ UnsignedInt ]
BoolConst = `true` | `false`
```

Обмеженння:

- Кожна константа повинна мати тип, а величина константи повинна знаходитись у діапазоні репрезентативних значень для її типу.

Семантика:

- Кожна константа має тип, визначений її формою та значенням.

1.4 Ключові слова

```
KeyWord = `program` | `var` | `begin` | `end.` | `read` | `write` | `repeat` | `until` | `if` | `then` | `fi`
```

2. Типи

Мова Z99 обробляє значення трьох типів: int, real, bool.

```
Type = `int` | `real` | `bool`
```

1. Цілий тип int може бути представлений оголошеною змінною типу int, або константою IntNum. Діапазон значень залежить від реаліза-ції.

2. Дійсний тип real може бути представлений оголошеною змінною типу real, int, або константою RealNum.
3. Логічний тип bool може бути представлений оголошеною змінною типу bool, або константою BoolConst (true або false). У деяких випадках числові типи можуть конвертуватися до логічного за правилом: 0 – false, а усі інші значення – true.

3. Синтаксис

3.1 Вирази

Вираз - це комбінація операторів і операндів у певній послідовності, що визначає порядок обчислення значення. Мова Z99 підтримує арифметичні та логічні вирази. Значення, обчислене за арифметичним виразом, має тип real (дійсне число) або int (ціле число). Значення, обчислене за логічним виразом, має тип bool: true або false. Найвищий пріоритет в унарного мінуса та унарного плюса, а далі, у порядку зменшення пріоритету слідують MultOp, AddOp та RelOp. Послідовність двох або більше операторів з однаковим пріоритетом асоціативна.

```
Expression = AritmExpression | BoolExpr
BoolExpr = AritmExpression RelOp AritmExpression
AritmExpression = Term | Term AddOp AritmExpression
Term = SignedFactor | SignedFactor MultOp Term
SignedFactor = [AddOp] Factor
Factor = Ident | Constant | `(` AritmExpression `)`
```

Обмеження:

- Використання змінної, з невизначеним на момент обчислення виразу значенням, викликає помилку.

3.2 Арифметичні оператори

Арифметичні оператори поділяються на бінарні, та унарні. Всі бінарні оператори у виразах є лівоасоціативними, окрім піднесення до степені.

До бінарних відносяться оператори додавання, віднімання, ділення, множення та піднесення до степені. Унарними є плюс та мінус.

4. Программа

Кожна програма починається з термінала program та ідентифікатора програми. Ідентифікатор у подальшому ніяк не використовується в програмі. Після термінала var розміщується список декларацій DeclarList, де вказуються ідентифікатори змінних та призначенні їм типи. Програма не може містити неоголошену змінну.

```
Program = `program` Ident
    `var` DeclarList `;`
    `begin` StatementList `;` `end.`
```

Тіло програми починається з ключового слова «begin» та складається з StatementList — це одна, або більше інструкцій з роздільником крапка з комою: «;». В кінці кожної програми знаходиться ключове слово «end.».

5. Оголошення

Список декларацій DeclarList — це одна, або більше декларацій. Роздільником декларацій є крапка з комою: «;».

Оголошення (декларація) — це список змінних, які розділені комою, і в кінці якого стоїть двокрапка, та один з підтримуваних мовою Z99 типів. Усі перечислені до двокрапки змінні будуть вважаться цього типу.

```
DeclarList = Declaration { `;` Declaration}
Declaration = IndentList `:` Type
IndentList = Indent { `,` Indent}
Ident = Letter { Letter | Digit }
```

Приклад:

```
var a, b: int;
    c, d: real;
    e: bool;
```

6. Інструкції

6.1 Оператор присвоювання

Синтаксис:

```
Assign = Ident AssignOp Expression
AssignOp = `=`
```

Обмеження:

- Тип змінної з ідентифікатором Ident має відповідати типу виразу праворуч оператора `=`. При цьому тип int може не явно конвертуватись в тип real, але не навпаки.
- Ідентифікатор обов`язково має бути оголошений у місці оголошення змінних.
- Тип bool не може використовуватись в арифметичних операціях.

Семантика:

- Змінна Ident набуває значення порахованого Expression

6.2 Оператори введення та виведення

Оператори введення та виведеня починаються с ключовго слова read або write відповідно, а далі приймають список змінних у дужках.

Синтаксис:

```
Input = read `(` IdentList `)`
Output = write `(` IdentList `)`
```

Обмеження:

- Вивід змінної з невизначеним значення викликає помилку

Семантика:

- Оператор, в залежності від свого типу, по черзі зчитує, або виводить змінні зі списку.
- Программа автоматично визначає тип введеного значення, та намагається присвоїти його змінній за правилами аналогічними оператору присвоєння.

6.3 Умовний оператор

Синтаксис:

```
BranchStatement = `if` Expression `then` StatementList `fi`
```

Семантика:

- Якщо Expression має значення true, то виконуються інструкції зі StatementList. Інакше програма виконується далі.

6.4 Оператор циклу

Синтаксис:

```
RepeatStatment = `repeat` StatementList `until` BoolExpr
```

Семантика:

- Виконуємо StatementList, після чого рахуємо значення BoolExpr, повторюємо ці дії доки BoolExr повертає true.

АРХІТЕКТУРА, МЕТОДИ ТА АЛГОРИТМИ

1. Архітектура

Інтерпретатор мови програмування Z99 складається з чотирьох основних компонентів: Lexer (лексичний аналізатор), Parser (синтаксичний аналізатор), Semantic Analyzer (семантичний аналізатор), Interpreter (інтерпретатор).

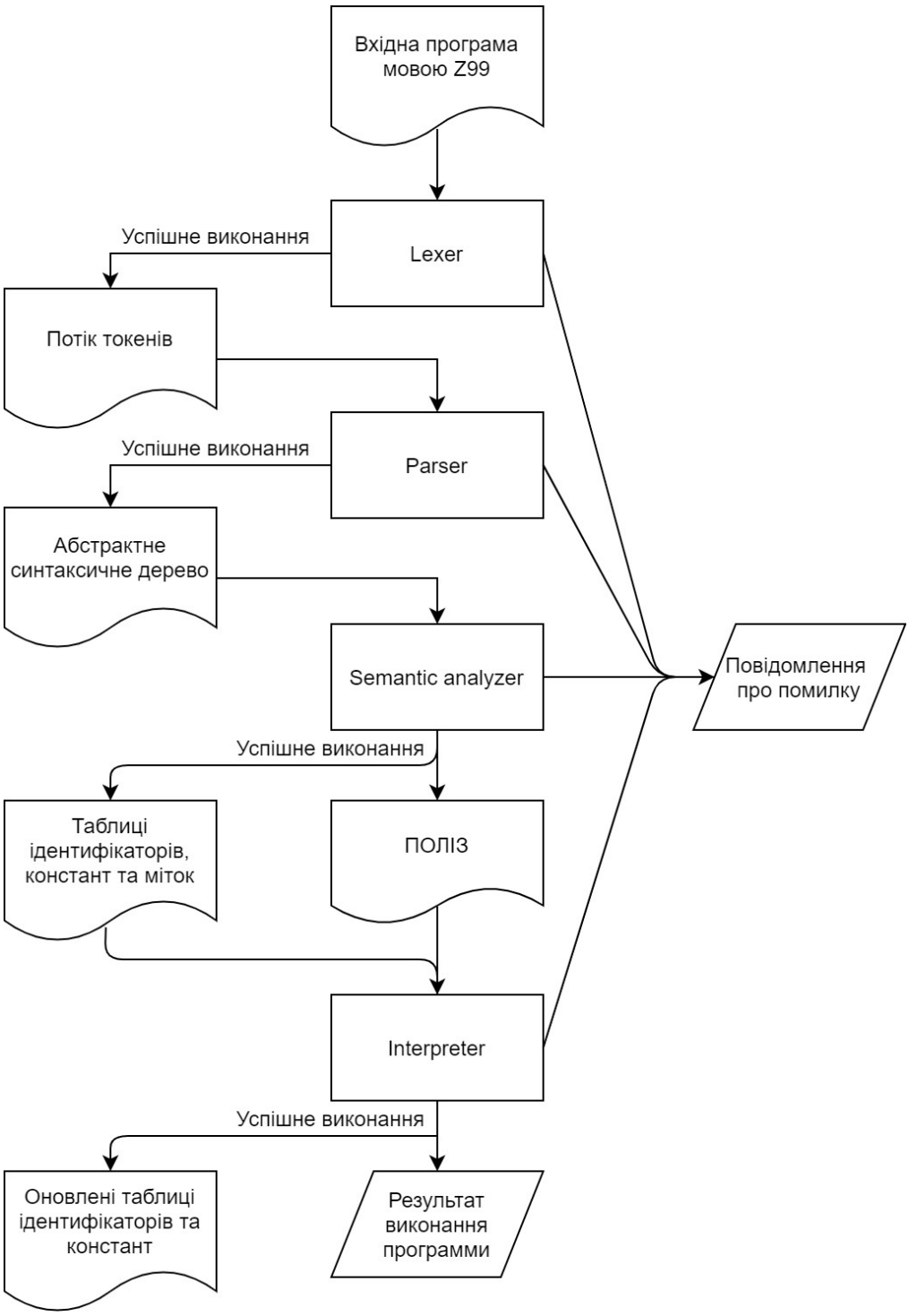


Рисунок 1 - Загальна схема роботи інтерпретатора

Кожен компонент працює незалежно один від одного, та не може безпосередньо викликати інший, а приймає лише результат його роботи, що дозволяє нам простіше їх тестувати та, у разі необхідності, гнучко змінювати будь-який з компонентів. У разі помилки компонент припиняє свою роботу, та кидає виключення.

2. Лексичний аналізатор

2.1 Функції лексичного аналізатора

Головна задача лексичного аналізатор – виділення токенів із вхідного рядка символів (тексту програми).

Токен – підрядок вхідного рядка, який містить лише термінальні символи і є найменшою одиницею мови для синтаксичного аналізу.

До побічних задач лексичного аналізатора можуть належати побудова таблиць ідентифікаторів та констант, але, оскільки, у нашому випадку, для синтаксичного аналізу вони не потрібні, то можна делегувати цю задачу на більш пізній етап трансляції: семантичному аналізатору.

У загальному випадку для роботи лексичного аналізатора потрібна таблиця токенів, а також вхідний рядок символів (вхідна програма). На виході ми повинні отримати таблицю токенів.

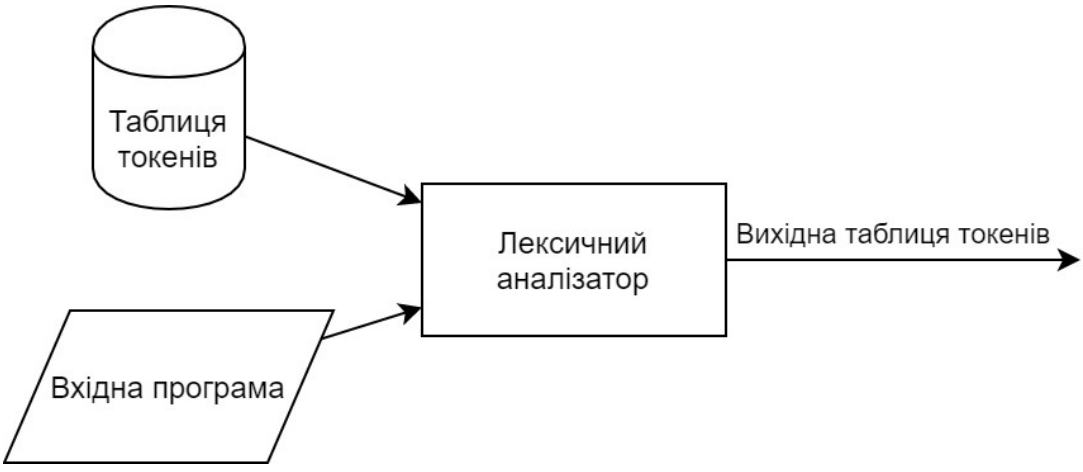


Рисунок 2 - Схема лексичного аналізатора

2.2 Діаграма станів

Діаграма станів – граф, у якого вершини позначають стани у яких може перебувати лексичний аналізатор, а ребра – символи під якими здійснюється перехід між станами.

Виділяють початкові, проміжні та кінцеві стани. Починається аналіз з початкового стану, та у кінці доходить до кінцевого в якому відбувається запис токену у, або кидається виключення з помилкою, після чого аналізатор повертається знову до початкового стану.

З ціллю зменшити кількість ребер всі символи поділяють на класи. Наприклад можна виділити такі класи: «Літера», «Цифра», «Точка» та інші в залежності від потреб.

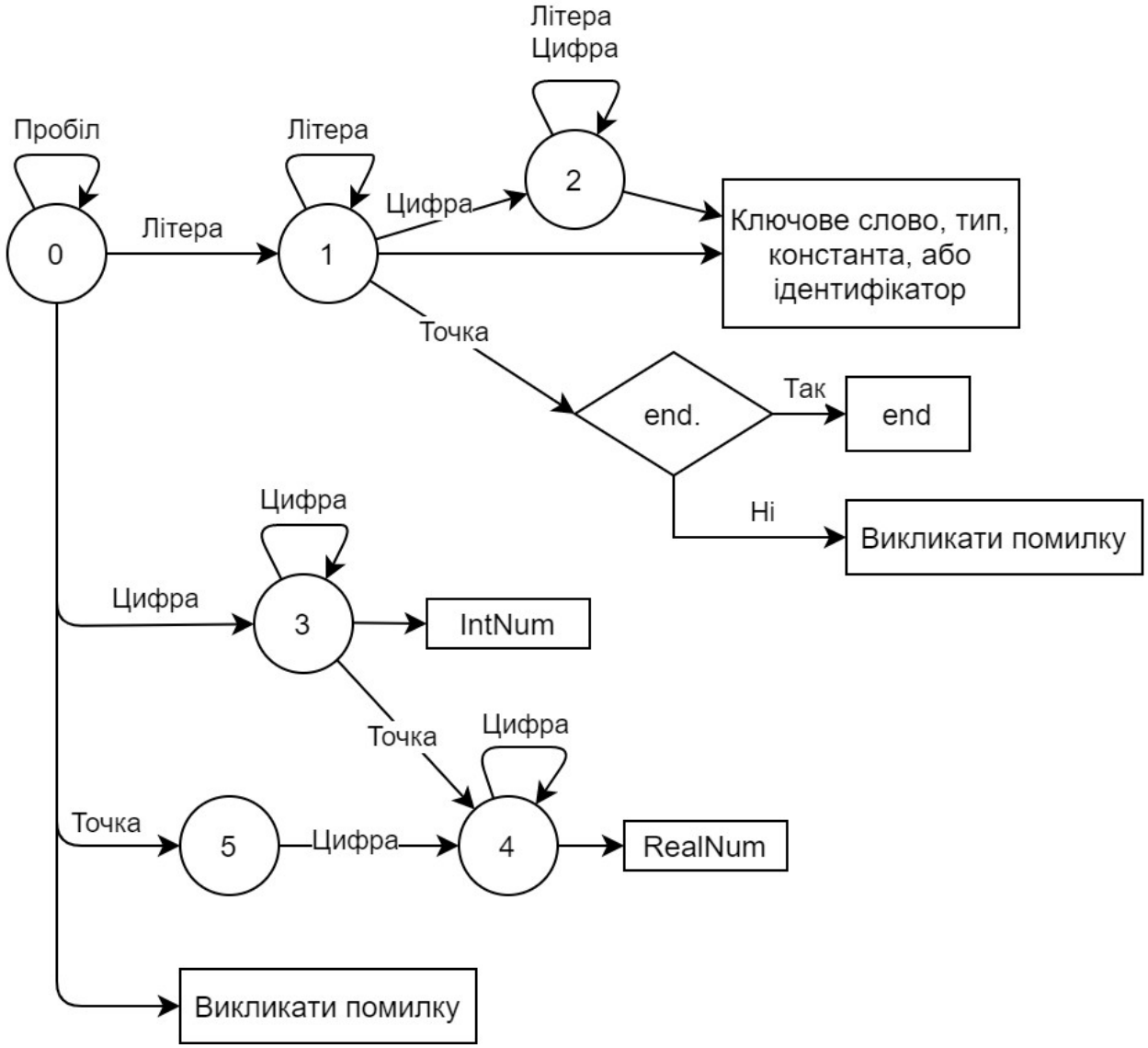


Рисунок 3 - Діаграма станів

На рисунку 3 зображена найскладніша частина діаграми станів мови Z99, яка виявляє ключові слова, константи, логічні типи, ідентифікатори, цілі та дійсні числа, а також виявляє лексичні помилки.

3. Синтаксичний аналізатор

3.1 Функції синтаксичного аналізатора

Синтаксичний аналізатор визначає у вхідному тексті мовні конструкції, що описується граматикою. На вхід він отримує побудований лексичним аналізатором потік токенів та будує абстрактне синтаксичне дерево.

3.2 Види синтаксичного аналізу

Розразняють дві категорії алгоритмів для синтаксичного розбору: низхідний (зверху-вниз) та висхідний (снизу-вверх). У мові Z99 для парсингу використовується низхідний алгоритм «метод рекурсивного спуску», який будує дерево від корня до кінцевих вузлів.

Ми починаємо з головного правила

```
Program = `program` Ident
    `var` DeclarList `;`
    `begin` StatementList `;` `end.`
```

та намагаємось поступово замінити найлівіший нетермінал цього правила на його праву частину. Тобто спочатку підставляємо лексему `program`, потім переходимо до правила Ident, та намагаємось замінити це правило на його праву частину і так далі.

При низхідному розборі з’являється проблема повернень: якщо нетермінал визначається правилом у якому є кілька альтернатив, наприклад

```
Factor = Ident | Constant | `(` AritmExpression `)`
```

то як дізнатися котрим із правил замінити Factor? Для цього можна вибрати будь який з варіантів з припущенням, що він вірний. У разі, якщо виникне помилка, то потрібно повернутися до цього правила та вибрати інший варіант. Також одним з варіантів вирішення цієї задачі може будет перегляд контексту

навколо правила, для прийняття правильного рішення. У подальшому ми будемо застосовувати обидва цих варіанти залежно від ситуації.

4. Семантичний аналізатор та трансляція

4.1 Функції семантичного аналізатору

Семантичний аналізатор обробляє абстрактне синтаксичне дерево, та перевіряє його на семантичне узгодження з визначенням мови. У нашому випадку він також формує таблиці ідентифікаторів, констант, міток, і проводить трансляцію інструкцій у ПОЛІЗ. Фактично це не прямі його задачі, але розподілення цих обов'язків по відповідним їм компонентам лише ускладнить програму та не принесе значної вигоди.

4.2 Визначення ПОЛІЗ

ПОЛІЗ (Польский інверсний запис) – постфіксна форма запису математичних і логічних операцій, для операторного представлення програми.

Постфіксна форма означає, що всі аргументи розміщуються перед знаком операції. Наприклад інфіксний вираз 7 – 2 у постфіксній формі буде записаний як 7 2 -, більше прикладів наведено у таблиці 1. Зверніть увагу, що інструкція присвоєння теж вважається інфіксним виразом з оператором «=».

№	Оператор	Вираз	Постфіксна форма
1	Інфіксний	a / b - 5	a b / 5 -
2	Інфіксний	i = 4	i 4 =
3	Префіксний	-10	10 u-

Таблиця 2 – Приклади запису виразів

У постфіксній формі запису виразів виникає необхідність у чіткому поділу бінарних та унарних операторів: неможливо позначити їх однаковими символами. Для цього додамо літеру «u» перед унарними операторами.

4.2 Алгоритм трансляція виразів та операцій присвоєння

Якщо позначити вираз E, а його постфіксну нотацію через ПОЛІЗ(E), наприклад, якщо E = «i = 2», то ПОЛІЗ(E) = «i 2 =». Тоді для трансляції виразів у ПОЛІЗ можна використати рекурсивний алгоритм з наступними правилами:

1. Якщо E – змінна, або константа, то ПОЛІЗ(E) = E
2. Якщо E – вираз виду «E1 op E2», де op – інфіксний бінарний оператор, то ПОЛІЗ(E) = «ПОЛІЗ(E1) ПОЛІЗ(E2) op»
3. Якщо E – вираз виду «op E1», де op – префіксний унарний оператор, то ПОЛІЗ(E) = «ПОЛІЗ(E1) op»
4. Якщо E – вираз виду «(E1)», де op – префіксний унарний оператор, то ПОЛІЗ(E) = «ПОЛІЗ(E1) op» (Дужки ігноруються).

У загальному випадку існує більше правил, але для нашої мови досить цих чотирьох.

Розглянемо, як буде працювати цей алгоритм на наступному прикладі:

```
result = (-a + b) / 2
```

Від синтаксичного аналізатору ми отримаємо дерево, яке у спрощеному вигляді можна подати як на рисунку 4.

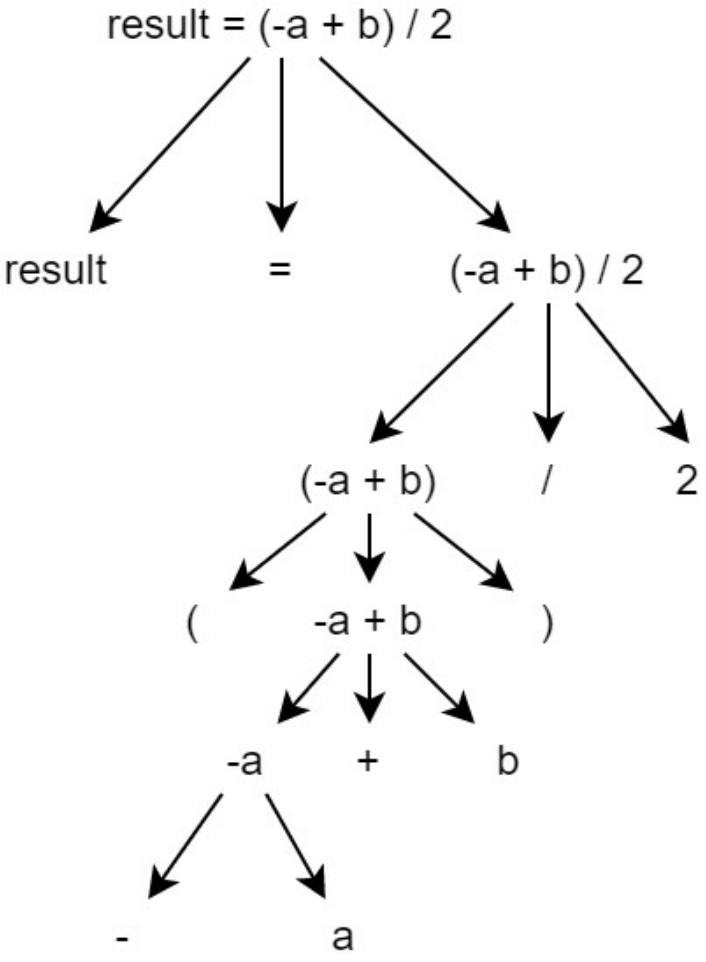


Рисунок 4 - Синтаксичне дерево арифметичного виразу

Для обробки цього дерева використаємо наступний алгоритм: спочатку знаходимо ПОЛІЗ від лівого операнду (якщо він існує), потім від правого, а потім від оператора. Якщо зустрічаємо дужки, то ігноруємо їх. Отже, порядок виконання дій буде наступний:

Спочатку отримаємо `‘ПОЛІЗ(«result = (-a + b) / 2»)’`

Спрощуємо за правилом 2: `‘ПОЛІЗ(«result») ПОЛІЗ(«(-a + b) / 2») =’`

Спрощуємо за правилом 1: `‘result ПОЛІЗ(«(-a + b) / 2») =’`

За правилом 2: `‘result ПОЛІЗ(«(-a + b)») ПОЛІЗ(«2») / =’`

За правилом 4: `‘result ПОЛІЗ(«-a + b») ПОЛІЗ(«2») / =’`

За правилом 2: `‘result ПОЛІЗ(«-a») ПОЛІЗ(«b») + ПОЛІЗ(«2») / =’`

За правилом 3: `‘result ПОЛІЗ(«a») u- ПОЛІЗ(«b») + ПОЛІЗ(«2») / =’`

Далі декілька спрощень за правилом 1 і отримуємо результат:

`‘result a u- b + 2 / =’`

4.3 Мітки та оператори безумовного переходу

Для подальшої трансляції циклів та умовних операторів потрібно ввести декілька нових понять.

Спершу введемо поняття мітки – це особливий ідентифікатор, який вказує на якесь місце (адресу) у коді. В мові Z99 не можливо використовувати мітки при написанні програми, але вони з’являються на етапі трансляції у ПОЛІЗ.

З мітками будуть працювати два оператори безумовного переходу:

1. Jump – унарний оператор, операндом для якого може бути лише мітка. При обробці інтерпретатор повинен перемістится на адресу, що вказана для мітки, та почати виконання з інструкції, яка знаходиться під цією адресою.
2. JumpIf (JF) – бінарний оператор, лівим операндом якого є вираз, який можна привести до логічного типа, а правим – мітка. Якщо значення

виразу виявилось хибою (false), то оператор переміститься на адресу мітки, аналогічно як спрацював би Jump. Але якщо значення виразу – істинна (true), то нічого не відбувається.

4.4 Трансляція умовного оператору if

Оператор розгалуження у граматиці записаний правилом branchStatement

```
BranchStatement = `if` Expression `then` StatementList `fi`
```

Семантика оператора виглядає наступним чином:

1. Якщо expression має значення true, переходимо до statementList
2. Якщо expression має значення false, переходимо на мітку L0.

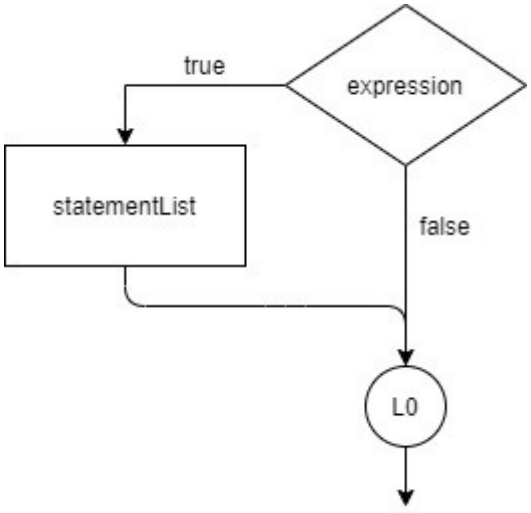


Рисунок 5 - Блок-схема оператору if

У ПОЛІЗ це виглядає як:

n	n + 1	n + 2	n + 3	n + 4
ПОЛІЗ(expression)	Label : L0	Binary : JF	ПОЛІЗ(statementList)	…

Таблиця 3 – Схема постфікс-коду оператора if

При цьому у таблиці міток є запис:

Мітка	Адреса
L0	n + 4

Таблиця 4 – Таблиця міток

Де n – номер інструкції ПОЛІЗ(expression). При цьому, звичайно, ПОЛІЗ(expression) та ПОЛІЗ(statementList) будуть займати більше ніж один номер інструкції, але для простоти залишимо так.

Для досягнення потрібного результату, використаємо наступний алгоритм трансляції:

1. Додаєм в код ПОЛІЗ(expression)
2. Генеруємо мітку та додаємо її до ПОЛІЗ коду
3. Додаємо в код оператор jumpIf
4. Додаємо в код ПОЛІЗ(statementList)
5. Записуємо в таблицю міток наступний номер інструкції ПОЛІЗ коду, як адресу для мітки створеної у другому пункті.

4.5 Трансляція оператору циклу repeat

Оператор циклу записаний у граматиці правилом repeatStatement:

```
RepeatStatment = `repeat` StatementList `until` BoolExpr
```

Семантика оператора:

1. Виконуємо statementList
2. Поки boolExpr має значення true – виконуємо statementList (переходимо до мітки L0)
3. Якщо boolExpr має значення false – переходимо до мітки L1

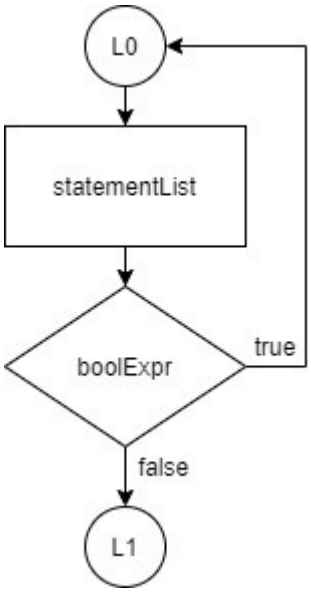


Рисунок 6 – Блок-схема оператору циклу repeat

У ПОЛІЗ це виглядає як:

n	n + 1	n + 2	n + 3	n + 4	n + 5	n + 6
ПОЛІЗ(SL)	ПОЛІЗ(boolExpr)	Label: L1	Binary: JF	Label: L0	Unary: Jump	…

Таблиця 5 – Схема постфікс-коду оператору repeat

Де SL – скорочення від statementList.

Таблиця міток:

Мітка	Адреса
L0	n
L1	n + 6

Таблиця 6 – Таблиця міток

Для цього використаємо наступний алгоритм трансляції:

1. Запам’ятаємо наступний номер інструкції ПОЛІЗ коду, наприклад у змінну start
2. Додаємо ПОЛІЗ(statementList)
3. Додаємо ПОЛІЗ(boolExpr)
4. Генеруємо мітку (умовно назвемо її L1) та додаємо її до ПОЛІЗ коду
5. Додаємо оператор jumpIf
6. Генеруємо мітку (умовно назвемо її L0) та додаємо її до ПОЛІЗ коду
7. Додаємо оператор Jump

8. Записуємо в таблицю міток адресу start для мітки L0
9. Записуємо в таблицю міток наступний номер інструкції ПОЛІЗ коду, як адресу для мітки L1.

4.6 Трансляція операторі введення read

Оператор введення read у граматиці записаний правилом input:

```
Input = read `(` IdentList `)`
```

Виконаємо read унарним оператором, який буде зчитувати значення для змінної з вершини стеку.

Оскільки identList може складатися з декількох ідентифікаторів, кожен з яких потрібно зчитати, то скористаємося наступним алгоритмом трансляції:

1. Для кожного ідентифікатора з identList:
   a. Додати ідентифікатор до ПОЛІЗ коду
   b. Додати оператор read

Таким чином для інструкції мовою z99

```
read(a, b)
```

буде сформований ПОЛІЗ:

`Ident: a`	`Unary: read`	`Ident: b`	`Unary: read`

Таблиця 7 – Схема постфікс-коду оператора read

4.7 Трансляція оператору виведення write

Оператор виведення write у граматиці записаний правилом output:

```
Output = write `(` IdentList `)`
```

Оператор write транслюється аналогічно оператору введення read, за винятком того, що після кожного ідентифікатора у ПОЛІЗ замість read додається унарний оператор write, який друкує значення ідентифікатора з вершини стеку.

Таким чином для інструкції мовою z99

```
write(a, b)
```

буде сформований ПОЛІЗ:

`Ident: a`	`Unary: write`	`Ident: b`	`Unary: write`

Таблиця 8 – Схема постфікс-коду оператора write

5. Інтерпретатор

5.1 Функції інтерпретатора

Інтерпретатор отримує на вхід ПОЛІЗ-програму, яку повинен виконати, а також допоміжну інформацію: таблиці ідентифікаторів, констант і міток. На виході ми отримаємо оновлені таблиці ідентифікаторів та констант і результат роботи програми.

5.2 Метод інтерпретації

Для інтерпретації пройдемось зліва направо по кожному елементу ПОЛІЗ коду і виконаємо наступні дії:

1. Якщо елемент – ідентифікатор, мітка або константа, то покладемо його на стек.
2. Якщо елемент – бінарний оператор, то:
    a. Знімаємо з вершини стека правий операнд
    b. Знімаємо з вершини стеку лівий операнд
    c. Виконуємо операцію в залежності від типу оператора, та кладемо результат, якщо він існує, на стек.
3. Якщо елемент – унарний оператор, то:
    a. Знімаємо з вершини стеку операнд
    b. Виконуємо операцію в залежності від типу оператора, та кладемо результат, якщо він існує, на стек.

Якщо після обробки усіх елементів ПОЛІЗ-програми стек буде порожній, то інтерпретацію можна вважати успішною, інакше – кидаємо виключення про помилку інтерпретації.

ПРОГРАМНА РЕАЛІЗАЦІЯ

1. Реалізація лексичного аналізатора

Лексичний аналізатор працює за допомогою діаграми станів, яка програмно реалізується, на основі графу, классом Z99Lexer\FSM\FSM:

```
class FSM
{
    public const DEFAULT_STATE = 'default';

    private $graph;
    private $start_state;
    private $states = [];

    public function __construct()
    {
        $this->graph = new Graph();
    }

    /**
     * Додає початкову вершину, з якої буде починатися розбір.
     */
    public function addStart($id): State
    {
        $vertex = $this->createVertex($id);
        $this->start_state = $this->addStateByVertex($id,
$vertex);
        return $this->start_state;
    }

    /**
     * Додає проміжну вершину.
     */
    public function addState($id): State
    {
        $vertex = $this->createVertex($id);
        $vertex->setAttribute('graphviz.color', 'green');
        return $this->addStateByVertex($id, $vertex);
    }

    /**
     * Додає ребро між вершинами.
     * $trigger - символ для якого це ребро працює.
     */
    public function addTrigger($trigger, $from, $to): void
```

```
        {
            $from = $this->states[$from];
            $to = $this->states[$to];

            if ($trigger === self::DEFAULT_STATE) {
                $from->setDefault($to);
            } else {
                $from->addTrigger($trigger, $to);
            }
        }

        /**
         * Додає фінальну вершину
         *
         * $needNext - логічний флаг, який повідомляє лексеру чи
         * потрібно переходити до наступного символа, перед тим як
         * повернутись до початкової вершини.
         *
         * $callback - функція, в якій ми можемо записати новий
токен,
         * яка викликається з параметрами:
         * LexerWriterInterface $writer - інтерфейс, який містить
методи
         *                                    для запису нового токену,
         * string $string - підрядок лексеми,
         * int $line - номер рядка.
         * Якщо виникла помилка розпізнавання токену, то стан
повинен
         * кинути виключення LexerException.
         *
         * Наприклад:
         * static function (LexerWriterInterface $writer,
         *                  string $string, int $line)
         * {
         *     if ($string !== 'end.') {
         *         throw new LexerException('Unknown keyword.',
         *                                  $string, $line);
         *     }
         *
         *     $writer->addToken($line, $string, 'Keyword');
         * }
         */
        public function addFinalState($id, callable $callback,
                                      $needNext = true): State
        {
            $vertex = $this->createVertex($id);
            $vertex->setAttribute('graphviz.color', 'blue');
```

```
            $vertex->setAttribute('final', true);
            $vertex->setAttribute('callback', $callback);
            $vertex->setAttribute('needNext', $needNext);
            return $this->addStateByVertex($id, $vertex);
        }

        private function createVertex($id): Vertex
        {
            $vertex = $this->graph->createVertex($id);
            $vertex->setAttribute('graphviz.label', $id);
            return $vertex;
        }

        private function addStateByVertex($id, Vertex $vertex):
State
        {
            return $this->states[$id] = new State($vertex);
        }

        public function getStartState(): State
        {
            if ($this->start_state === null) {
                throw new LogicException('Final state not set');
            }

            return $this->start_state;
        }
    }
```

Також створимо клас, який буде зберігати інформацію про токен:

```
    class Token
    {
        private $line;
        private $string;
        private $type;

        public function __construct(int $line, string $string,
                                    string $type, ?int $index =
null)
        {
            $this->line = $line;
            $this->string = $string;
            $this->type = $type;
        }

        public function getLine(): int
        {
```

```
            return $this->line;
        }

        public function getString(): string
        {
            return $this->string;
        }

        public function getType(): string
        {
            return $this->type;
        }
    }
```

Другий параметр який потрібний для роботи лексичного аналізатора – вхідний рядок. Щоб підтримувати різні джерела, з яких може зчитуватись рядок створимо єдиний інтерфейс:

```
    interface CharStreamInterface
    {
        public const EOF = "\u{0000}";

        /**
         * Returns next character from input stream.
         * Return EOF in the end of stream.
         *
         * @return string
         * @throws OutOfRangeException
         */
        public function read() : string;
    }
```

Та його реалізацію, яка працює за файлами:

```
    class FileStream implements CharStreamInterface
    {
        private $source;
        private $current = 0;
        private $length;

        /**
         * FileStream constructor.
         */
        public function __construct($filename)
        {
            $this->source =
str_split(file_get_contents($filename));
            $this->length = count($this->source);
```

```
        }

        public function read(): string
        {
            if ($this->current === $this->length) {
                $this->current++;
                return CharStreamInterface::EOF;
            }

            if ($this->current > $this->length) {
                throw new OutOfRangeException('Input stream '
                                              . 'has already
ended.');
            }

            return $this->source[$this->current++];
        }
    }
```

Тепер можна написати клас лексичного аналізатора, який приймає на вхід клас діаграми станів (FSM) та об’єкт, що реалізує CharStreamInterface і повертає масив токенів. Цей клас, також буде підтримувати LexerWriterInterface, і передавати сам себе у функцію, що викликається при досягненні кінцевого стану.

```
    class Lexer implements LexerWriterInterface
    {
        private $tokens = [];
        private $stream;
        private $fsm;
        private $state;

        public function __construct(CharStreamInterface $stream,
                                    FSM $fsm)
        {
            $this->stream = $stream;
            $this->fsm = $fsm;
            $this->state = $fsm->getStartState();
        }

        /**
         * Виконує лексичний аналіз: створює таблицю токенів
         */
        public function tokenize() : void
        {
            $char = $this->stream->read();
            if ($char === CharStreamInterface::EOF) {
```

```
                return;
            }

            $string = '';
            $line = 1;

            $done = false;

            while (!$done) {
                if ($char === CharStreamInterface::EOF) {
                    $done = True;
                }

                if (in_array($char,["\n", "\r\n", "\n\r'"], true))
{
                    $line++;
                }

                $this->state = $this->state->getNextState($char);

                if ($this->state !== $this->fsm->getStartState()) {
                    $string .= $char;
                }

                if ($this->state->isFinal()) {
                    $this->state->handle($this, $string, $line);
                    $string = '';

                    if (!$done && $this->state->isNeedNext()) {
                        $char = $this->stream->read();
                    }

                    $this->state = $this->fsm->getStartState();
                } elseif (!$done) {
                    $char = $this->stream->read();
                }
            }
        }

        public function getTokens(): array
        {
            return $this->tokens;
        }
    }
```

2. Реалізація синтаксичного аналізатора

Синтаксичний аналізатор працює за методом рекурсивного спуску. Для кожного нетерміналу створено метод, який намагається відобразити праву частину відповідного правила. Він може викликати інші методи, або самого себе для пошуку інших нетерміналів, або перевіряти наявність токена у вхідному потоці. На початку метод створює вершину синтаксичного дерева, та додає до цієї вершини дітей відповідно граматиці мови. Якщо вершина разом з дітьми вдачно сформувалась, то метод повертає цю вершину.

Для зручності роботи та незалежності від джерела токенів створимо інтерфейс потоку токенів:

```
interface TokenStreamInterface
{
    public const EOF = "\u{0000}";

    /**
     * Goes to the next character from input stream.
     * Return EOF in the end of stream.
     *
     * @throws OutOfRangeException
     */
    public function next() : Token;

    /**
     * Look next character from input stream.
     * Return EOF in the end of stream.
     *
     * @throws OutOfRangeException
     */
    public function lookAhead() : Token;

    /**
     * Returns to previous position.
     */
    public function back() : void;

    /**
     * Returns current token
     */
    public function current() : Token;

    /**
```

```
         * Returns id of current position
         */
        public function remember() : int;

        /**
         * Go to position id (which can be received
         *                    from remember() method)
         */
        public function goTo($id) : void;
    }
```

Класс парсера буде приймати цей інтерфейс через конструктор і працювати з ним. Розглянемо клас Parser, в якому прибрано всі методи нетерміналів, а залишено тільки допоміжні методи для синтаксичного аналізу.

```
    class Parser
    {
        private $stream;

        public function __construct(TokenStreamInterface $stream)
        {
            $this->stream = $stream;
        }

        /**
         * Перевіряє на співпадання наступний токен з потоку.
         * $lexem - назва потрібного токену.
         * Якщо токен співпав - повертає вершину синтаксичного
дерева
         * з потрібним токеном, інакше - null.
         */
        private function match($lexeme): ?Node
        {
            if ($this->stream->lookAhead()->getType() === $lexeme) {
                $root = new Node($lexeme);
                $token = $this->stream->next();
                $root->addChild(new Node($token->getString(),
                                         $token->getLine()));
                return $root;
            }

            return null;
        }

        /**
         * Працює аналогічно match, але кидає виключення, якщо
         * токен не співпав.
```

```
         */
        private function matchOrFail($lexeme): Node
        {
            if ($result = $this->match($lexeme)) {
                return $result;
            }

            throw new ParserException("Expected $lexeme",
                                      $this->stream->lookAhead());
        }

        /**
         * Перевіряє наступний токен потоку на співпадання хоча
         * б з одним переданим токеном. Кидає виключення, у разі,
         * якщо не було виявлено співпадань.
         */
        private function matchOneOfLexeme(array $lexemes): Node
        {
            foreach ($lexemes as $lexeme) {
                if ($result = $this->match($lexeme)) {
                    return $result;
                }
            }

            throw new ParserException('Expected one of this lexemes
'
                . implode(', ', $lexemes), $this->stream-
>lookAhead());
        }

        /**
         * Перевіряє потік токенів на співпадання з правилом.
         * $rule - назва метода, який перевіряє правило.
         * Повертає null, якщо правило не співпало.
         */
        private function matchRule(string $rule): ?Node
        {
            $position = $this->stream->remember();
            try {
                 return $this->$rule();
            } catch (ParserException $e) {
                $this->stream->goTo($position);
            }

            return null;
        }
```

```
        /**
         * Перевіряє потік токенів на співпадання хоча б з одним
         * з переданих правил.
         * Кидає виключення, якщо не було співпадань.
         */
        private function matchOneOfRules(array $rules): Node
        {
            foreach ($rules as $rule) {
                if ($result = $this->matchRule($rule)) {
                    return $result;
                }
            }

            throw new ParserException('Expected one of this rules '
                . implode(', ', $rules), $this->stream-
>lookAhead());
        }

        /**
         * Виконує передану функцію, поки в ній не виникне помилка.
         * Перед виконанням запом’ятовує позицію в потоці токенів,
         * та у випадку помилки повертається до неї.
         */
        private function repeatedMatch(callable $function): void
        {
            while (true) {
                $position = $this->stream->remember();
                try {
                    $function();
                } catch (ParserException $e) {
                    $this->stream->goTo($position);
                    break;
                }
            }
        }
    }
```

Повний код парсера з усіма методами для кожного нетерміналу занадто великий, що привести його, тут. Тому розглянемо лише декілька правил:

```
    /**
     * program
     *     : Program Ident Var declareList Semi
     *       Begin statementList Semi End EOF
     *     ;
     * @return Node
     */
```

```
    public function program(): Node
    {
        $root = new Node('program');
        $root->addChild($this->matchOrFail('Program'));
        $root->addChild($this->matchOrFail('Ident'));
        $root->addChild($this->matchOrFail('Var'));
        $root->addChild($this->declareList());
        $root->addChild($this->matchOrFail('Semi'));
        $root->addChild($this->matchOrFail('Begin'));
        $root->addChild($this->statementList());
        $root->addChild($this->matchOrFail('Semi'));
        $root->addChild($this->matchOrFail('End'));
        $root->addChild($this->matchOrFail('EOF'));
        return $root;
    }

    /**
     * statementList
     *     : statement (Semi statement)*
     *     ;
     * @return Node
     */
    public function statementList(): Node
    {
        $root = new Node('statementList');
        $root->addChild($this->statement());
        $this->repeatedMatch(function () use ($root) {
            $tokens = [$this->matchOrFail('Semi'), $this-
>statement()];
            $root->addChild($tokens[0]);
            $root->addChild($tokens[1]);
        });
        return $root;
    }

    /**
     * statement
     *     : assign
     *     | input
     *     | output
     *     | branchStatement
     *     | repeatStatement
     *     ;
     * @return Node
     */
    public function statement(): Node
    {
```

```
        $root = new Node('statement');
        $root->addChild($this->matchOneOfRules(['assign',
            'input', 'output', 'branchStatement',
'repeatStatement']));
        return $root;
    }
```

3. Реалізація семантичного аналізатору та транслятора

Семантичний аналізатор отримує на вхід дерево, та повинен побудувати таблиці ідентифікаторів, міток та констант, а також виконати трансляцію коду у ПОЛІЗ.

3.1 Таблиці ідентифікаторів, міток, констант та їх побудова

Для кожної сутності реалізовано реалізовано свій клас:

Ідентифікатор реалізується класом Z99Compiler\Entity\Identificator, та має обов’язкові поля: id, name, та опціональні type, value.

Константа реалізується класом Z99Compiler\Entity\Constant, та має поля: id, value, type.

Мітка реалізується класом Z99Compiler\Entity\Label, та має лише поле name.

Для таблиць створені окремі класи Z99Compiler\Tables\IdentifiersTable та Z99Compiler\Tables\ConstantsTable, Z99Compiler\Tables\LabelsTable. Які містять в собі методи для роботи с таблицями, такі як: додати новий елемент, пошук елемента та інші.

Фомування цих таблиць відбувається на етапі семантичного аналізу.

Таблиця ідентифікаторів створюється на основі блоку декларації, який знаходиться перед початком програми, та починається зі слова var. Після чого створюється таблиця констант, на основі всіх знайдених в коді констант.

Метод declareList приймає вершину синтаксичного дерева declareList, та викликає метод declaration для кожного потомка цієї вершини з ім’ям declaration.

```
    protected function declareList(Node $node): void
    {
        $children = Tree::getChildrenOrFail($node);

        foreach ($children as $child) {
            if ($child->getName() === 'declaration') {
                $this->declaration($child);
```

```
            }
        }
    }
```

Метод declaration спочатку, за допомогою методу identList, збирає масив імен ідентифікаторів, а потім визначає їх тип. Після чого записує кожен ідентифікатор і таблицю.

```
    protected function declaration(Node $node): void
    {
        $children = Tree::getChildrenOrFail($node);

        $type = null;
        $identifiers = [];

        foreach ($children as $child) {
            if ($child->getName() === 'identList') {
                $identifiers = $this->identList($child);
            } elseif ($child->getName() === 'Type') {
                $type = $child->getChildren()[0]->getName();
            }
        }

        foreach ($identifiers as $identifier) {
            $this->identifiers->addIdentifier($identifier, $type);
        }
    }

    protected function identList(Node $node): array
    {
        $children = Tree::getChildrenOrFail($node);
        $identifiers = [];

        foreach ($children as $child) {
            if ($child->getName() === 'Ident') {
                $identifiers[] = $child->getChildren()[0]-
>getName();
            }
        }
        return $identifiers;
    }
```

Для побудови таблиці констант, рекурсивно знаходимо усі константи в дереві, та додаємо їх у таблицю. Для пошуку використовується метод findConstants, який повертає масив вершин з ім’ям constant.

```
    private function findConstants(Node $node): array
    {
        if ($node->getName() === 'constant') {
            return [$node];
        }

        $constants = [];

        if (($children = $node->getChildren()) !== null) {
            foreach ($children as $child) {
                $array = $this->findConstants($child);
                foreach ($array as $item) {
                    $constants[] = $item;
                }
            }
        }

        return $constants;
    }
```

Потім для кожної вершини з масиву викликаємо метод constant, який виконує запис до таблиці:

```
    public function constant(Node $node): void
    {
        $child = $node->getFirstChild();
        $value = $child->getFirstChild()->getName();
        $this->constants->addConstant($value);
    }
```

При цьому при додаванні клас таблиці сам визначить тип константи, перевірить, що її ще немає в таблиці та згенерує для неї унікальний id.

```
    public function addConstant($value): Constant
    {
        if (filter_var($value, FILTER_VALIDATE_INT) !== FALSE) {
            $type = 'int';
        } elseif (filter_var($value, FILTER_VALIDATE_FLOAT) !==
FALSE) {
            $type = 'real';
        } elseif ($value === 'true' || $value === 'false') {
            $type = 'bool';
        } else {
            throw new RuntimeException('Unknown constant type '
                                       . get_debug_type($value));
```

```
        }
        if (($constant = $this->find($value)) !== null) {
            return $constant;
        }
        $constant = $this->add($value, $type);
        return $constant;
    }
```

У подальшому робота с ідентифікаторами, константами, або мітками виконуються тільки через об’єкти таблиць, адже в них виконуються додаткові перевірки, такі як відповідність типів.

3.2 Реалізація транслятора

Транслятор працю схожим чином на синтаксичний аналізатор. Для кожного нетерміналу створюється метод, який знає, як обробляти цей нетермінал. Наприклад метод boolExpr викликає спочатку метод arithmExpr для лівого операнду, потім для правого, а потім додає оператор у результуючий масив с кодом.

```
    /**
     * Handle boolExpr rule
     * "arithmExpression RelOp arithmExpression"
     */
    public function boolExpr(Node $node): void
    {
        $children = Tree::getChildrenOrFail($node);
        $this->arithmExpression($children[0]);
        $this->arithmExpression($children[2]);
        $this->RPNCode[] = $this->relOp($children[1]);
    }
```

Метод arithmExpr працює схожим чином, тож ми знаємо що після виконання цього метода в результуючому масиві буде записана правильна послідовність ПОЛІЗ коду, що відповідає цьому правилу.

Оскільки, аналогічно синтаксичному аналізатору, код занадто великий, приведемо тут лише приклад трансляції деяких операторів:

Трансляція умовного оператору:

```
    /**
     * Handle branch Statement
     * "If expression Then statementList Semi Fi"
     */
```

```
    public function branchStatement(Node $branchStatement): void
    {
        $children = Tree::getChildrenOrFail($branchStatement);
        $this->expression($children[1]);
        $endLabel = $this->RPNCode[] = new Label(
                                            $this-
>generateLabelName());
        $this->RPNCode[] = new BinaryOperator('jumpIf', 'JF');
        $this->statementList($children[3]);
        $end = count($this->RPNCode);
        $this->labelsTable->add($endLabel, $end);
    }
```

Ім’я мітки генерується за принципом літера «l» плюс наступний вільний номер.

```
    /**
     * Generate unique name for label
     */
    private function generateLabelName(): string
    {
        return 'l' . $this->labelNum++;
    }
```

Трансляція оператору вводу:

```
    /**
     * Handle input statement
     * "Input LBracket identList RBracket"
     */
    public function input(Node $input): void
    {
        $children = Tree::getChildrenOrFail($input);
        $identList = $children[2];
        foreach ($identList->getChildren() as $item) {
            if ($item->getName() === 'Ident') {
                $this->RPNCode[] = $this->ident($item);
                $this->RPNCode[] = new UnaryOperator('read',
'Input');
            }
        }
    }
```

Трансляція оператору циклу:

```
    /**
     * Handle repeat Statement
     * "Repeat statementList Semi Until boolExpr"
     */
```

```
    public function repeatStatement(Node $repeatStatement): void
    {
        /**
         * ~start~ statementList
         * boolExpr startLabel JF
         * endLabel Jump ~end~
         */
        $children = Tree::getChildrenOrFail($repeatStatement);
        $start = count($this->RPNCode);
        $this->statementList($children[1]);
        $this->boolExpr($children[4]);
        $endLabel = $this->RPNCode[] = new Label(
                                              $this-
>generateLabelName());
        $this->RPNCode[] = new BinaryOperator('jumpIf', 'JF');
        $startLabel = $this->RPNCode[] = new Label(
                                              $this-
>generateLabelName());
        $this->RPNCode[] = new UnaryOperator('jump', 'Jump');
        $end = count($this->RPNCode);
        $this->labelsTable->add($startLabel, $start);
        $this->labelsTable->add($endLabel, $end);
    }
```

4. Реалізація інтерпретатора

Інтерпретатор, через конструктор створює стек і отримує ПОЛІЗ код, таблиці міток, ідентифікаторів та констант.

```
    public function __construct(array $RPNCode,
                                ConstantsTable $constants,
                                IdentifiersTable $identifiers,
                                LabelsTable $labels)
    {
        $this->RPNCode = $RPNCode;
        $this->constants = $constants;
        $this->identifiers = $identifiers;
        $this->stack = new SplStack();
        $this->labels = $labels;
    }
```

Основний процесс інтерпретації відбувається у методі process, який за допомогою циклу обробляє кожен елемент ПОЛІЗ-програми, а номер поточного елементу зберігає у полі $current. Якщо оператор виявився невідомим, то кидається виключення з помилкою інтерпретації.

```
    public function process(): void
```

```
    {
        $size = count($this->RPNCode);
        while ($this->current < $size) {
            $item = $this->RPNCode[$this->current];
            if ($item instanceof Constant
                || $item instanceof Identifier
                || $item instanceof Label) {
                $this->stack->push($item);
            } elseif ($item instanceof BinaryOperator) {
                $this->binaryOperator($item);
            } elseif ($item instanceof UnaryOperator) {
                $this->unaryOperator($item);
            } else {
                throw new RuntimeException('Unknown item '
                                           . get_class($item));
            }
            $this->current++;
        }
    }
```

Бінарні оператори обробляються функцією binaryOperator, яка спочатку підіймає зі стеку правий операнд, потім лівий, а потім в залежності від типу оператору виконує різні дії: якщо це арифметичний, або логічний оператор, то рахується результат і кладеться на стек. Оператор присвоювання, за допомогою таблиці ідентифікаторів, присвоює лівому оператору значення правого. При цьому в середині таблиці перевіряється існування ідентифікатора та відповідність типів. JumpIf працює згідно специфікації.

```
    private function binaryOperator(BinaryOperator $operator): void
    {
        $right = $this->stakPop();
        $left = $this->stakPop();

        if ($operator->isAddOp() || $operator->isMultOp()) {
            $result = $this->calculate($operator, $left, $right);
            $constant = $this->constants-
>addConstant($result['value']);
            $this->stack->push($constant);
            return;
        }

        if ($operator->isRelOp()) {
            $result = $this->calculateBool($operator, $left,
$right);
```

```
            $constant = $this->constants-
>addConstant($result['value']);
            $this->stack->push($constant);
            return;
        }

        if ($operator->isAssignOp()) {
            $this->identifiers->changeValue($left->getId(),
$right);
            return;
        }

        if ($operator->isJF()) {
            $this->jumpIf($left, $right);
            return;
        }

        throw new RuntimeException('Unknown binary operator '
                                   . $operator->getType());
    }
```

Унарні оператори обробляються функцією unaryOperator, яка знімає з верхівки стеку операнд і виконує з ним дію в залежності від оператора: унарний плюс нічого не робить, та повертає операнд у стек, унарний мінус змінює знак операнда і також повертає його у стек. Оператор введення зчитує рядок з консолі. після чого перетворює його у константу, та присвоює операнду (ідентифікатору). Оператор Jump змінює номер поточної інструкції, на той що вказаний у операнді (мітці).

```
    private function unaryOperator(UnaryOperator $operator): void
    {
        $operand = $this->stakPop();

        if ($operator->isPlus()) {
            $this->stack->push($operand);
            return;
        }

        if ($operator->isMinus()) {
            $value = -$operand->getValue();
            $constant = $this->constants->addConstant($value);
            $this->stack->push($constant);
            return;
        }
```

```
        if ($operator->isOutput()) {
            echo $operand->getValue() . ' ';
            return;
        }

        if ($operator->isInput()) {
            $value = readline();
            $constant = $this->constants->addConstant($value);
            $this->identifiers->changeValue($operand->getId(),
                                            $constant);
            return;
        }

        if ($operator->isJump()) {
            $this->jumpTo($this->labels->getAddress($operand));
            return;
        }

        throw new RuntimeException('Unknown unary operator '
                                   . $operator->getType());
    }
```

ТЕСТУВАННЯ

1. Базовий приклад

Програма зчитує три числа, та знаходить середнє арифметичне серед модулів цих чисел.

```
1.  program positiveAverage
2.  var i: int;
3.      sum, value: real;
4.  begin
5.      sum = 0.0;
6.      i = 1;
7.      repeat
8.          read (value);
9.          if value < 0 then
10.             value = -value;
11.         fi;
12.         sum = sum + value;
13.         i = i + 1;
14.     until i <= 3;
15.     sum = sum / 3;
16.     write(sum);
17. end.
```

Вхідні дані	Результат програми
1, 2, 3	2
5, -5, 5	5
10, -3, 46	19.666666666667

Таблиця 9 – набір тестів для базового прикладу програми

Запустивши лексичний аналізатор отримаємо наступний набір токенів (скорочена версія, тому що повний список токенів завеликий):

```
[{
    "line": 1,
    "type": "Program",
    "string": "program"
```

```
},
{
    "line": 1,
    "type": "Ident",
    "string": "positiveAverage"
},
...
{
    "line": 16,
    "type": "RBracket",
    "string": ")"
},
{
    "line": 16,
    "type": "Semi",
    "string": ";"
},
{
    "line": 17,
    "type": "End",
    "string": "end."
}]
```


Цей набір токенів передаємо у синтаксичний аналізатор, та отримаємо синтаксичне дерево.

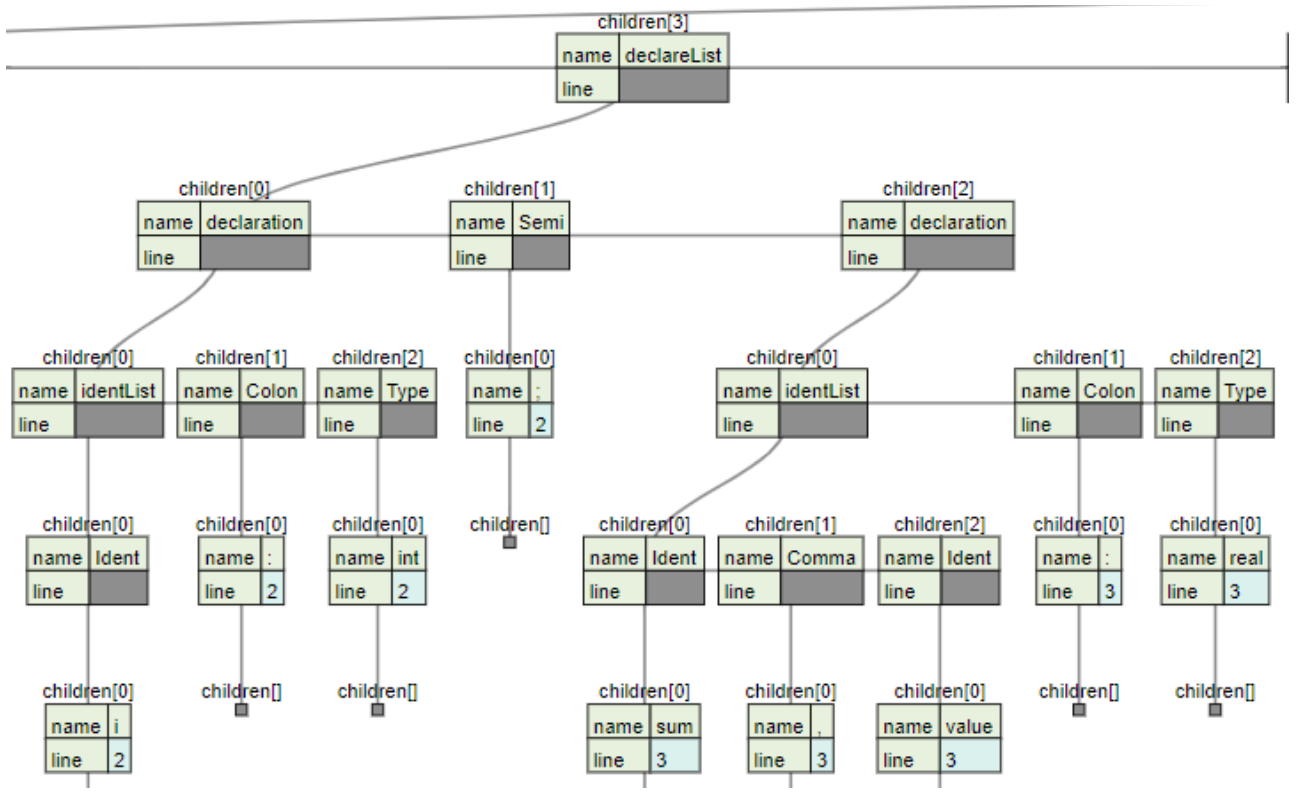


Рисунок 7 – Фрагмент синтаксичного дерева до базового прикладу

Семантичний аналізатор видасть наступний ПОЛІЗ код та таблиці:

RPN:

0	1	2	3	4
(real : sum)	(real : 0.0)	(AssignOp : =)	(int : i)	(int : 1)

5	6	7	8	9
(AssignOp : =)	(real : value)	u(Input : read)	(real : value)	(int : 0)

10	11	12	13	14
(RelOp : <)	(Label : l0 : 17)	(JF : jumpIf)	(real : value)	(real : value)

15	16	17	18	19
u(Minus : -)	(AssignOp : =)	(real : sum)	(real : sum)	(real : value)

20	21	22	23	24
(Plus : +)	(AssignOp : =)	(int : i)	(int : i)	(int : 1)

25	26	27	28	29
(Plus : +)	(AssignOp : =)	(int : i)	(int : 3)	(RelOp : <=)

```
-------------------------------------------------------------------------------------------------------------
 30                | 31                | 32                | 33                | 34                |
-------------------------------------------------------------------------------------------------------------
 ( Label : l1 : 34) | (JF : jumpIf)     | ( Label : l2 : 6)  | u(Jump : jump)    | (real : sum)      |
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
 35                | 36                | 37                | 38                | 39                |
-------------------------------------------------------------------------------------------------------------
 (real : sum)      | (int : 3)         | (Slash : /)       | (AssignOp : =)    | (real : sum)      |
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
 40                | 41                | 42                | 43                | 44                |
-------------------------------------------------------------------------------------------------------------
 u(Output : write)  |
-------------------------------------------------------------------------------------------------------------
```

Identifiers:

Id	Name	Type	Value
@00	i	int	Undefined
@01	sum	real	Undefined
@02	value	real	Undefined

Labels:

Name	Address
l0	17
l2	6
l1	34

Constants:

Id	Value	Type
@00	0.0	real
@01	1	int
@02	0	int
@03	3	int

Після чого можемо запустити інтерпретатор, та перевірити роботу програми на тестовому прикладі.

```
D:\122\Compilator\Z99Compiler>php compiler.php run example\positiveAverage.z99
10
-3
46
19.666666666667
```

```
Identifiers:
Id   Name       Type       Value
@00  i          int        4
@01  sum        real       19.666666666667
@02  value      real       46

Constants:
Id   Value      Type
@00  0.0        real
@01  1          int
@02  0          int
@03  3          int
@04  10         int
@05  false      bool
@06  2          int
@07  true       bool
@08  -3         int
@09  13         int
@10  46         int
@11  59         int
@12  4          int
@13  19.666666666667 real
```

Як бачимо базовий приклад на кожному з етапів виконався правильно.

2. План тестування

№	Тип випробування	Очікуваний результат	Кількість тестів
1	Лексична помилка	Повідомлення про помилку	1
2	Синтаксична помилка	Повідомлення про помилку	2
3	Вкладений if	Успішне виконання	1
4	Вкладений repeat	Успішне виконання	1
5	Використання арифметичного виразу в умові if	Успішне виконання	1
6	Введення даних	Помилки при невідповідності типів	3

Таблиця 9 – План тестування

3. Тестування за планом

Випробування 1: Лексична помилка

Тест 1: Символ, який не існує у граматиці у рядку 4

```
1    program test
2    var a: int;
3    begin
4        a \ 5;
5    end.
```

Результат:

```
Lexer failed with error 'Unknown char.' in line 4
String: \
```

Висновок: Помилка виявлена правильно

Випробування 2: Синтаксична помилка

Тест 1: Відсутність ключового слова begin

```
1    program test
2    var a: int;
3        a = 5;
4    end.
```

Результат:

```
Parsing failed with error 'Expected Begin' in line 3
Token: @03  Ident      'a'          1
```

Висновок: Помилка виявлена правильно

Тест 2: Відсутність крапки з комою у рядку 4

```
1    program test
2    var a: int;
3    begin
4        a = 5
5    end.
```

Результат:

```
Parsing failed with error 'Expected Semi' in line 5
Token: @05  End        'end.'     NULL
```

Висновок: Рядок помилки збільшений на 1, що цілком нормальна поведінка. У всьому іншому помилка виявлена правильна.

Випробування 3: Вкладений if

```
1   program IncludedBranch1
2   var a, b: int;
3   begin
4       a = 4;
5       b = 5;
6       if true then
7           if a < b then
8               write(b);
9           fi;
10      fi;
11  end.
```

Результат:

```
D:\122\Compilator\Z99Compiler>php compiler.php run example\IncludedBranch1.z99
5
```

ПОЛІЗ-програма:

```
-------------------------------------------------------------------------------------------------------------
 0                | 1                | 2                | 3                | 4                |
-------------------------------------------------------------------------------------------------------------
 (int : a)        | (int : 4)        | (AssignOp : =)   | (int : b)        | (int : 5)        |
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
 5                | 6                | 7                | 8                | 9                |
-------------------------------------------------------------------------------------------------------------
 (AssignOp : =)   | (bool : true)    | ( Label : l0 : 16) | (JF : jumpIf)    | (int : a)        |
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
 10               | 11               | 12               | 13               | 14               |
-------------------------------------------------------------------------------------------------------------
 (int : b)        | (RelOp : <)      | ( Label : l1 : 16) | (JF : jumpIf)    | (int : b)        |
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
 15               | 16               | 17               | 18               | 19               |
-------------------------------------------------------------------------------------------------------------
 u(Output : write)  |
-------------------------------------------------------------------------------------------------------------
```

Таблиці:

Identifiers:

Id	Name	Type	Value
@00	a	int	4
@01	b	int	5

Labels:

Name	Address
l1	16
l0	16

```
Constants:
Id   Value      Type
@00  4          int
@01  5          int
@02  true       bool
```

Висновок: програма виконалась правильно, виведено значення «5».

Випробування 4: Вкладений repeat

```
program IncludedRepeat
var i, j: int;
    sum: real;
begin
    sum = 0;
    i = 0;
    repeat
        j = 0;
        repeat
            sum = sum + i + j;
            j = j + 1;
        until j <= 1;
        i = i + 1;
    until i < 2;
    write(sum);
end.
```

Результат:

```
D:\122\Compilator\Z99Compiler>php compiler.php run example\IncludedRepeat.z99
4
```

ПОЛІЗ-програма:

0	1	2	3	4
(real : sum)	(int : 0)	(AssignOp : =)	(int : i)	(int : 0)
5	6	7	8	9
(AssignOp : =)	(int : j)	(int : 0)	(AssignOp : =)	(real : sum)
10	11	12	13	14
(real : sum)	(int : i)	(int : j)	(Plus : +)	(Plus : +)
15	16	17	18	19
(AssignOp : =)	(int : j)	(int : j)	(int : 1)	(Plus : +)
20	21	22	23	24
(AssignOp : =)	(int : j)	(int : 1)	(RelOp : <=)	(Label : l0 : 28)
25	26	27	28	29
(JF : jumpIf)	(Label : l1 : 9)	u(Jump : jump)	(int : i)	(int : i)
30	31	32	33	34
(int : 1)	(Plus : +)	(AssignOp : =)	(int : i)	(int : 2)
35	36	37	38	39
(RelOp : <)	(Label : l2 : 40)	(JF : jumpIf)	(Label : l3 : 6)	u(Jump : jump)
40	41	42	43	44
(real : sum)	u(Output : write)			

Таблиці:

Identifiers:

Id	Name	Type	Value
@00	i	int	2
@01	j	int	2
@02	sum	real	4

Labels:

Name	Address
l1	9
l0	28
l3	6
l2	40

```
Constants:
Id   Value      Type
@00  0          int
@01  1          int
@02  2          int
@03  true       bool
@04  false      bool
@05  4          int
```

Висновок: Программа виконалась правильно.

Випробування 5: Використання арифметичного виразу в умові if

```
1  program ArithmBranch
2  var a, b: int;
3  begin
4      a = 4;
5      b = 5;
6      if a + (b / 0.3) then
7          write(a);
8      fi;
9  end.
```

Результат:

```
D:\122\Compilator\Z99Compiler>php compiler.php run example\ArithmBranch.z99
4
```

ПОЛІЗ-програма:

```
-------------------------------------------------------------------------------------------------------------
 0                | 1                | 2                | 3                | 4                |
-------------------------------------------------------------------------------------------------------------
 (int : a)        | (int : 4)        | (AssignOp : =)   | (int : b)        | (int : 5)        |
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
 5                | 6                | 7                | 8                | 9                |
-------------------------------------------------------------------------------------------------------------
 (AssignOp : =)   | (int : a)        | (int : b)        | (real : 0.3)     | (Slash : /)      |
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
 10               | 11               | 12               | 13               | 14               |
-------------------------------------------------------------------------------------------------------------
 (Plus : +)       | ( Label : l0 : 15) | (JF : jumpIf)    | (int : a)        | u(Output : write) |
-------------------------------------------------------------------------------------------------------------
```

Таблиці:

```
Identifiers:
Id   Name       Type       Value
@00  a          int        4
@01  b          int        5
Constants:
Id   Value      Type
@00  4          int
@01  5          int
@02  0.3        real
@03  16.666666666667 real
@04  20.666666666667 real
```

```
Labels:
Name       Address
l0         15
```

Висновок: Програма виконалась правильно, адже результат виразу в умові був відмінний від нуля, а отже конвертується у значення true.

Випробування 6: Введення даних

Тест 1: Зчитування змінних логічного типу

```
1    program ReadBool
2    var flag: bool;
3    begin
4        read(flag);
5        write(flag);
6    end.
```

Результат 1:

```
D:\122\Compilator\Z99Compiler>php compiler.php run example\ReadBool.z99
true
true
```

```
Identifiers:
Id   Name       Type       Value
@00  flag       bool       true
```

```
Constants:
Id   Value      Type
@00  true       bool
```

Результат 2:

```
D:\122\Compilator\Z99Compiler>php compiler.php run example\ReadBool.z99
false
false
Identifiers:
Id   Name       Type       Value
@00  flag       bool       false


Constants:
Id   Value      Type
@00  false      bool
```

Результат 3:

```
D:\122\Compilator\Z99Compiler>php compiler.php run example\ReadBool.z99
1.4

In IdentifiersTable.php line 138:

  Cannot set variable (flag : bool) to real
```

Результат 4:

```
D:\122\Compilator\Z99Compiler>php compiler.php run example\ReadBool.z99
6

In IdentifiersTable.php line 138:

  Cannot set variable (flag : bool) to int
```

Результат 5:

```
D:\122\Compilator\Z99Compiler>php compiler.php run example\ReadBool.z99
RandomString

In ConstantsTable.php line 36:

  Unknown constant type string
```

(Програма спрацювала правильно, адже в нашій мові немає типу string, і цей рядок неможливо конвертувати до будь якого з існуючих типів)

Тест 2: Перевіримо зчитування змінних дійсного типу:

```
1    program ReadReal
2    var ident: real;
3    begin
4        read(ident);
5        write(ident);
6    end.
```

Результат 1:

```
D:\122\Compilator\Z99Compiler>php compiler.php run example\ReadReal.z99
-0.6
-0.6

Identifiers:
Id   Name       Type       Value
@00  ident      real       -0.6

Constants:
Id   Value      Type
@00  -0.6       real
```

Результат 2:

```
D:\122\Compilator\Z99Compiler>php compiler.php run example\ReadReal.z99
276
276

Identifiers:
Id   Name       Type       Value
@00  ident      real       276

Constants:
Id   Value      Type
@00  276        int
```

(Усе правильно, адже наша програма передбачає присвоєння типу int змінній типу real, але не навпаки)

Результат 3:

```
D:\122\Compilator\Z99Compiler>php compiler.php run example\ReadReal.z99
true

In IdentifiersTable.php line 138:

  Cannot set variable (ident : real) to bool
```

Тест 3: Перевіримо зчитування змінних цілочисельного типу:

```
1  program ReadInt
2  var ident: int;
3  begin
4      read(ident);
5      write(ident);
6  end.
```

Результат 1:

```
D:\122\Compilator\Z99Compiler>php compiler.php run example\ReadInt.z99
0.5

In IdentifiersTable.php line 138:

  Cannot set variable (ident : int) to real
```

Результат 2:

```
D:\122\Compilator\Z99Compiler>php compiler.php run example\ReadInt.z99
-5
-5

Identifiers:                                    Constants:
Id   Name       Type       Value                Id   Value      Type
@00  ident      int        -5                   @00  -5         int
```

Результат 3:

```
D:\122\Compilator\Z99Compiler>php compiler.php run example\ReadInt.z99
false

In IdentifiersTable.php line 138:

  Cannot set variable (ident : int) to bool
```

ЗАГАЛЬНІ ВИСНОВКИ

Розроблений інтерпретатор повністю виконує поставленні завдання, та працює відповідно специфікації. Але, можна виділити ряд покращень, які покращили б роботу інтерпретатора та відкрили більше можливостей для подальшого розширення функціоналу:

1. Розділити обов’язки семантичного аналізатору по іншим компонентам: окремим компонентом виділити транслятор.
2. Для кращої діагностики помилок. Передавати разом з ПОЛІЗ-програмою додаткову інформацію, таку як номер рядка інструкції.
3. Провести оптимізацію коду прибравши зайві перевірки, або зробивши їх швидшими. Наприклад, існують місця у яких можна замінити порівняння рядків, на порівняння цілочисельних ідентифікаторів цих рядків.
4. Написати автоматичні тести, що перевіряли б кожен компонент інтерпретатора окремо.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Медведєва В. М. Транслятори: лексичний та синтаксичний аналізатори / В. М. Медведєва, В. А. Третяк, НТУУ «КПІ» 2012 р.
2. Медведєва В. М. Транслятори: внутрішнє подання програм та інтерпретація / В. М. Медведєва, В. А. Третяк, НТУУ «КПІ» 2015 р.
3. Ютуб плейлист «Введенние в компиляторы» на каналі LLDevLab: https://www.youtube.com/watch?v=MePynBBljeM&list=PLeQDJtBkrIiT0TMQ3muv3zvNdsmBZFOR1
4. Юрій Стативка, Матеріали до виконання лабораторної роботи №3 «Трансляція у ПОЛІЗ арифметичних виразів, та інтерпретація постфіксного коду» 2020 р.
5. Юрій Стативка, Матеріали до виконання лабораторної роботи №4 «Трансляція у ПОЛІЗ операторів розгалуження, циклу та введення-виведення» 2020 р.
6. Юрій Стативка, Матеріали до виконання лабораторної роботи №5 «Виконання ПОЛІЗ програми» 2020 р.

ДОДАТОК 1

Таблиця токенів мови Z99.

Код	Лексема	Токен	Приклад	Неформальний опис
1		Ident	a, x1, z12f	ідентифікатор
2		IntNum	123, 0, -54	ціле без знаку
3		RealNum	.3, 2.3, 1.	дійсне без знаку
4	true	BoolConst	true	істина
5	false	BoolConst	false	хиба
6	program	Program	program	термінал program
7	var	Var	var	термінал var
8	begin	Begin	begin	термінал begin
9	end.	End	end.	термінал end.
10	int	Type	int	термінал integer
11	real	Type	real	термінал real
12	bool	Type	bool	термінал bool
13	read	Read	read	термінал read
14	write	Write	write	термінал write
15	repeat	Repeat	repeat	термінал for
16	until	Until	until	термінал to
18	if	If	if	термінал if
19	then	Then	then	термінал than
20	fi	Fi	fi	термінал fi
21	=	AssignOp	=	оп. присвоєння
22	+	Plus	+	термінал +
23	-	Minus	-	термінал -
24	*	Star	*	термінал *
25	/	Slash	/	термінал /
26	<	RelOp	<	термінал <
27	<=	RelOp	<=	термінал <=
28	==	RelOp	==	термінал =
29	>=	RelOp	>=	термінал >=
30	!=	RelOp	!=	термінал !=
31	(	LBracket	(	термінал (
32	)	RBracket	)	термінал)
33	.	Dot	.	термінал .
34	,	Comma	,	термінал ,

35	:	Colon	:	термінал :
36	;	Semi	;	термінал ;
37		WS	␣,\t	
38		EOL	\n, \r\n	
39		EOF	\u0000	

ДОДАТОК 2

Повна граматика мови Z99 у нотації РБНФ.

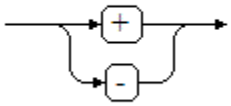
```
Program = `program` Ident \n `var` DeclarList `;` \n
`begin` StatementList `;` `end.`.

Ident = Letter { Letter | Digit }.

DeclarList = Declaration { `;` Declaration }.
Declaration = IdentList `:` Type.
IdentList = Ident { `,` Ident }.
Type = `int` | `real` | `bool`.

StatementList = Statement { `;` Statement }.
Statement = Assign | Input | Output | RepeatStatement
| BranchStatement.

Assign = Ident AssignOp Expression.

Expression = ArithmExpression | BoolExpr.
BoolExpr = ArithmExpression RelOp ArithmExpression.
ArithmExpression = Term | ArithmExpression AddOp Term.
Term = SignedFactor | SignedFactor MultOp Term.
SignedFactor = [AddOp] Factor.
Factor = Ident | Const | `(` ArithmExpression `)`.

Input = `read` `(` IdentList `)`.
Output = `write` `(` IdentList `)`.
```

```
RepeatStatement = `repeat` StatementList `until`
Expression.

IfStatement = `if` Expression `then` StatementList `fi`.

Const = IntNum | RealNum | BoolConst.
IntNum = Digit {Digit}.
RealNum = `.` IntNum | IntNum `.` [ IntNum ].

Letter = `a` | `b` | `c` | `d` | `e` | `f` | `g` | `h` |
`i`
| `j` | `k` | `l` | `m` | `n` | `o` | `p` | `q` | `r` | `s`

| `t` | `u` | `v` | `w` | `x` | `y` | `z`.
Digit = `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8`
| `9`.
BoolConst = `true` | `false`.
RelOp = `=` | `<=` | `<` | `>` | `>=` | `<>`.
AddOp = `+` | `-`.
MultOp = `*` | `/` | `^`.
```

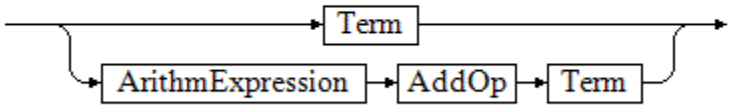
ДОДАТОК 3

Візуальне представлення граматики мови Z99 за допомогою діаграм Вірта.

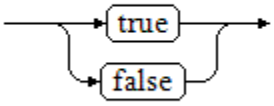
AddOp



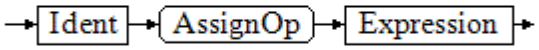
ArithmExpression



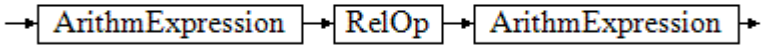
BoolConst



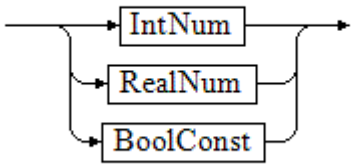
Assign



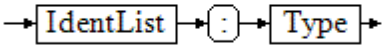
BoolExpr



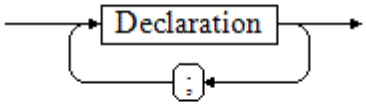
Const



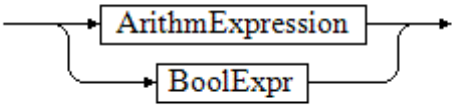
Declaration



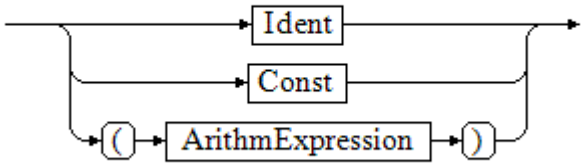
DeclarList



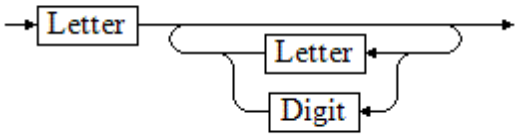
Expression



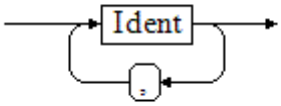
Factor



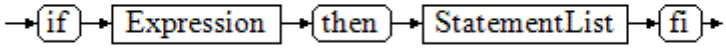
Ident



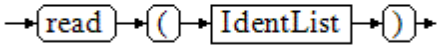
IdentList



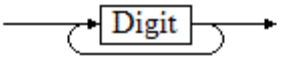
IfStatement



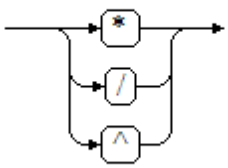
Input



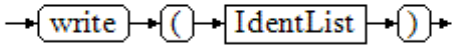
IntNum



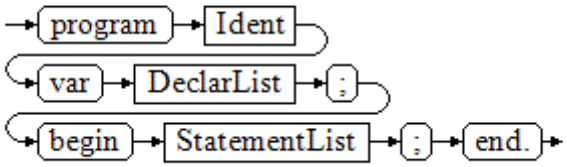
MultOp



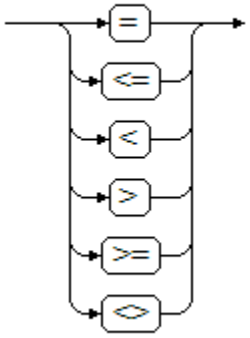
Output



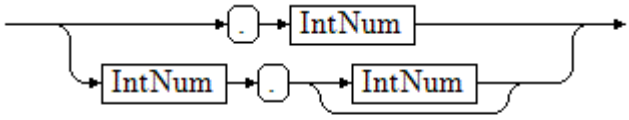
Program



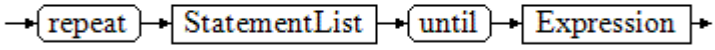
RelOp



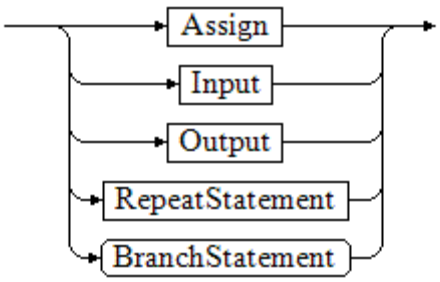
RealNum



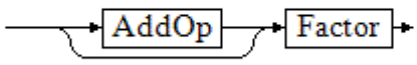
RepeatStatement



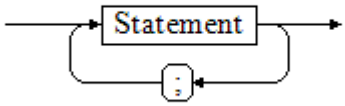
Statement



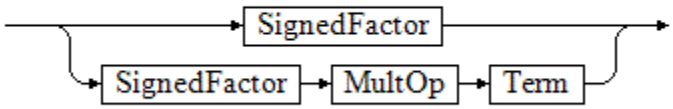
SignedFactor



StatementList



Term



Type