##### Міністерство освіти і науки України

Дніпропетровський національний університет

Факультет прикладної математики

Кафедра обчислювальної математики та математичної кібернетики

ДИПЛОМНА РОБОТА

|  |  |
| --- | --- |
| Тема: | Реалізація мовного процеса у середовищі Python |

|  |  |  |  |
| --- | --- | --- | --- |
|  | | **Виконавець**: | Ст. гр. ПЗ-06-1  Пискунов  Віктор Сергійович  ………………(підпис) |
| ***Допускається до захисту*** |  |  | |
| Завідувач кафедри МЗЕОМ  Проф. Байбуз О.Г.  “….”…………………. | | **Керівник**: | Доц. каф. МЗЕОМ  Горін А. К.  ………….……(підпис) |
| …………………………(підпис) | |  |  |
|  | | **Рецензент:** | ст. викл. каф. ОМ та МК  Шевельова А. Є.  ……………………(підпис) |

|  |
| --- |
| Дніпропетровськ  2010 |

# РЕФЕРАТ

Дипломна робота: “Реалізація мовного процеса у середовищі Python ” 51 с., 16 мал., 16 джерел, 3 додатки.

Об’єктом роботи є мовний процесор, у якого в якості вихідної є мова Pascal.

Мета роботи: кінцевою метою є розробка компілятора мови Pascal, який читає всю необхідну інформацію стосовно вихідної мови з конфігураційних файлів. На етапі бакалаврської роботи метою є проходження стадій лексичного, синтаксичного і семантичного аналізу і створення всіх необхідних умов для подальшої генерації машинного коду.

Методом аналізу граматики є метод рекурсивного спуску.

В процесі роботи при реалізації лексичного аналізу було впроваджено механізм регулярних виразів, при розробці синтаксичного аналізатора був розроблений формат запису РБНФ у конфігураційний файл (разом з семантичними діями), зручний для зчитування, і алгоритм читання граматики мови з конфігураційного файлу.

В результаті роботи розроблена параметризована програма, яка задовольняє меті роботи. Важливо, що можна модифікувати конфігураційні файли без модифікації мовного процесора.

***Результати роботи*** можуть бути корисними при вивченні курсу «Алгоритмічні мови и мовні процесори» і розробці мовних процесорів для створення алгоритмічних мов для цілей бізнесу і програмування побутових приладів.

***Ключові слова:*** ЛЕКСИКА, СИНТАКСИС, СЕМАНТИКА, МЕТОД РЕКУРСИВНОГО СПУСКУ, РЕГУЛЯРНІ ВИРАЗИ.

# Abstract

The purpose of this work is to show the main approaches to implementation of lexical, syntax and semantic analysis, as well as intermediate code generation. If the 2 first phases are well documented in the large amount of books, the latter phases are almost undocumented. Besides that, one of the main purposes is parameterization of product.

# ЗМІСТ

[Вступ 5](#_Toc264369147)

[1. Постановка задачі 6](#_Toc264369148)

[2. Опис існуючих методів розробки мовних процесорів 7](#_Toc264369149)

[2.1. Загальна схема трансляції 7](#_Toc264369150)

[2.2. Лексичний аналіз 9](#_Toc264369151)

[2.3. Синтаксичний аналіз 12](#_Toc264369152)

[2.4. Семантичний аналіз 18](#_Toc264369153)

[2.5. Генерація машинних команд 19](#_Toc264369154)

[3. Реалізація 21](#_Toc264369155)

[3.1. Вибір і обґрунтування мови реалізації 21](#_Toc264369156)

[3.2. Лексичний аналізатор 22](#_Toc264369157)

[3.2.1. Механізм регулярних виразів 22](#_Toc264369158)

[3.2.2. Алгоритм виділення лексем 24](#_Toc264369159)

[3.3. Синтаксичний аналізатор 25](#_Toc264369160)

[3.3.1. Граматика мови 25](#_Toc264369161)

[3.3.2. Формат граматики мови у РБНФ 26](#_Toc264369162)

[3.3.3. Формат зберігання РБНФ у конфігураційному файлі 29](#_Toc264369163)

[3.3.4. Алгоритм зчитування і аналізу граматики з конфігураційного файлу 30](#_Toc264369164)

[3.4. Семантичний аналізатор 32](#_Toc264369165)

[3.4.1. Опис семантичних дій 32](#_Toc264369166)

[3.4.2. Алгоритм зчитування правил граматики з обробкою семантичних дій 33](#_Toc264369167)

[3.5. Внутрішні структури 35](#_Toc264369168)

[3.5.1. Класи елементів мови 35](#_Toc264369169)

[3.5.2. Таблиця атрибутів 37](#_Toc264369170)

[4. Аналіз отриманих результатів. Тестування 38](#_Toc264369171)

[Висновки 43](#_Toc264369173)

[Література 44](#_Toc264369174)

[Додатки 46](#_Toc264369175)

[Додаток А. Файл metadata.yml 46](#_Toc264369176)

[Додаток Б. Файл Rules.yml 48](#_Toc264369177)

[Додаток В. Вигляд правил граматики без семантичних дій 50](#_Toc264369178)

# Вступ

Зараз у всьому світі існує велика кількість різноманітних мов програмування. Більше того, кожного року з’являються нові і нові мови. І для них створюють нові компілятори. Хоча принципи побудови компіляторів по великому рахунку однакові для всіх мов, є велика кількість нюансів у цьому питанні. Як відомо, стадії лексичного і синтаксичного аналізу дуже добре розглянуті у багатьох джерелах. Проте вже стадія семантичного аналізу, і тим паче генерації коду є комерційною таємницею кожної фірми. І цю інформацію у відкритому доступі знайти вкрай складно. І це не дивно, адже компанії-гіганти на зразок Borland, Microsoft заробляють на цьому свої гроші.

Тому основною метою даної роботи є якраз вивчення нюансів, що з’являються під час проходження двох вищезгаданих стадій розробки мовного процесора. Результати роботи будуть використані при вивченні курсу «Алгоритмічні мови та мовні процесори».

# Постановка задачі

Як відомо, під час реалізації мовного процесора можна виділити декілька стадій: лексичний аналіз, синтаксичний аналіз, семантичний аналіз, генерацію проміжного коду і генерацію машинного коду. Перші дві стадії добре описані у багатьох джерелах, останні ж є переважно комерційною таємницею кожної компанії, і відкритого доступу до них немає. Тому задачею дипломної роботи є

* Реалізувати лексичний, синтаксичний, семантичний аналізатори і генерацію проміжного коду
* Показати основні підходи до реалізації семантичного аналізу і генерації проміжного коду
* Відкрити нюанси, що трапляються під час реалізації вищевказаних стадій і показати методи їх вирішення
* Підготувати все необхідне для генерації машинних команд.
* Важливо параметризувати мовний процесор, щоб можна було модифікувати граматику мови без втручання у вихідний код програмного продукту.

Отримані знання в ході роботи можна буде використовувати для навчання студентів в курсі «Алгоритмічні мови і мовні процесори».

# Опис існуючих методів розробки мовних процесорів

## Загальна схема трансляції

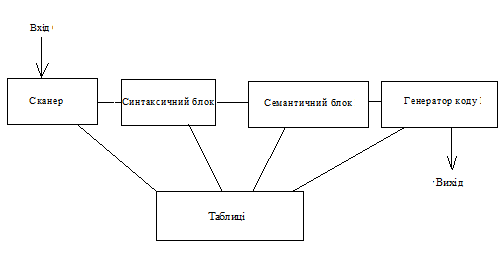
Кожний мовний процесор можна зобразити чотирма з’єднаними блоками, які будемо називати лексичним блоком, синтаксичним блоком, семантичним блоком і генератором коду. Ці блоки мають доступ до набору таблиць, де розміщується глобальна інформація про програму. Зв’язки між блоками можна зобразити наведеною схемою (рис.1). 

Рисунок 1. Загальна схема трансляції

На вхід мовного процесора подається вихідний код на Паскалі, який у випадку правильності можна розпізнати як послідовність слів з символів вхідного алфавіту.

Завданням сканеру є розділити вхідний потік символів на послідовність слів, перевірити належності кожного наступного слова до заданої мови та сформулювати таблицю підходящих лексем для синтаксичного блоку.

Синтаксичний блок приймає на вхід послідовність лексем, перевіряє правильність послідовності слів в даному тексті. При відсутності синтаксичних помилок, на виході матимемо правильну послідовність лексем для семантичного блоку.

Семантичний блок має за мету перевірити правильність змісту послідовності лексем, отриманих з попереднього блоку, а також сформулювати таблицю (або таблиці) атрибутів.

Генератор коду має за мету перекласти або оптимізувати правильний за результатами перевірки в попередніх блоках текст та передати його на вихід.

Можна виділити ще стадію проміжного коду, який генерується перед безпосередньою генерацією машинних команд.

## Лексичний аналіз

Завданням лексичного аналізатору є сканування та оцінка вхідного потоку символів. На етапі сканування лексичний аналізатор можна реалізувати у формі кінцевих автоматів, побудованих з регулярних виразів.

„**Розпізнавач** (recognizer) – програма, що отримує на вхід рядок х та відповідає „yes ” у випадку, якщо рядок х є реченням мови, або „no” у протилежному випадку. Регулярний вираз компілюється у розпізнавач шляхом побудови узагальненої діаграми переходів, що називається кінцевим автоматом. Автомат може бути детермінованим або не детермінованим (не детермінований може мати більш за один перехід з стану при одному і тому ж вхідному символі).” [1].

Існують кілька способів завдання кінцевого автомата Наприклад, кінцевий автомат можна завдати через 5 параметрів: \boldsymbol{M = (Q , \Sigma , \delta , q_0 , F)} (2.1)

де:

Q — кінцева множина станів автомата;

q0 — початковий стан автомату ( q_0 \in Q);

F — множина кінцевих станів, таких що F \subset Q;

Σ — Допустимий вхідний алфавіт, з якого формулюються рядки, що вважаються автоматом;

δ — задане відображення множини Q \times \Sigma у множину \mathcal {P} (Q) підмножин Q:

\delta : Q \times \Sigma \rightarrow \mathcal {P} (Q)(2.2)

(іноді δ називають функцією переходів автомата).

Автомат починає працювати зі стану q0, зчитує по одному символу вхідного рядка. Кожен наступний символ переводить автомат до стану Q згідно функції переходів. Якщо по завершенню потоку символів, автомат знаходиться у „допустимому” стані – слово приймається і відповідно належить мові даного автомату. Інакше, слово не приймається.

**Регулярні вирази** (англ. regular expressions, RegExp) — система синтаксичного розбору текстових фрагментів за формалізованим шаблоном, що базується на завданні запису-зразка для пошуку тексту. Регулярні вирази є розвитком символів-джокерів ( англ. wildcard characters).

Базис – словник, алфавіт вірних слів. Сигнатура – операції конструювання нових вірних слів з уже існуючих.

Для побудови мови регулярних висловів було використано три операції над подіями (дві бінарні і одна унарна):

1. *A* ∨ *B* — диз'юнкція або об'єднання (позначається також *A* ∪ *B*) — теоретико множинна операція: подія *A* ∨ *B* представляє собою звичайне об'єднання множин *A* та *B*;
2. *AB* — добуток (конкатенація), визначається через добуток слів. Добутком слів *p* та *q* називають слово *pq*, утворене в результаті дописування слова *q*справа до слова *p*. Подія *AB* складається із тих і тільки тих слів, які мають вигляд *pq*, де *p* належить *A*, а *q* належить *B*;
3. {*A*} — ітерація (позначається також *A*\*).

Ітерація визначається трохи складніше. Введемо позначення *A*m для добутку  

\begin{matrix} \underbrace{AA\dots A} \\ n \end{matrix}. Тоді ітерацію можна висловити через попередні дві операції наступним чином:

{*A*} = *A* ∨ *A*2 ∨ … ∨ *A*m ∨ …

Слово *q* тоді і тільки тоді належить {*A*}, коли *q* має вигляд *p*m, де *p* — належить *A*.

Треба зазначити, що в більшості сучасних мов програмування, що підтримують регулярні вирази, їх синтаксис не є однаковим, а отже має сенс при вивченні синтаксису регулярних виразів дивитися їх синтаксис безпосередньо у контексті конкретної мови програмування. Краще за всі інші регулярні вирази підтримують Perl, Python, Php, Javascript.

## Синтаксичний аналіз

**Синтаксичний аналіз** – процес співставлення лінійного потоку лексем із формальною граматикою мови, до якої належать вищезгадані лексеми. Завдання синтаксичного аналізатору – перевірити правильний порядок розташування лексем у вхідному потоці.

Зазвичай правила розташування лексем в деякій мові задаються за допомогою формальної граматики.

Під час парсингу текст оформлюється у [структуру даних](http://uk.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85), зазвичай — в [дерево](http://uk.wikipedia.org/wiki/%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_(%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85)), яке відображує синтаксичну структуру вхідної послідовності, і добре підходить для подальшої обробки. Зазвичай парсери працюють в два етапи: на першому ідентифікуються осмислені лексеми (виконується [лексичний аналіз](http://uk.wikipedia.org/wiki/%D0%9B%D0%B5%D0%BA%D1%81%D0%B8%D1%87%D0%BD%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D1%96%D0%B7)), на другому створюється дерево розбору. Проте можемо розробити парсер, що працює і в один проход. Це дуже зберігає час. Приклад дерева розбору зображено на рис. 2.

180px-Parsing-example

Рисунок 2. Приклад дерева розбора

**Формальна граматика, або просто граматика в теорії формальних мов** — спосіб опису формальної мови, тобто вилучення деякої підмножини із множин усіх слів деякого скінченого алфавіту. Існують породжувальні та розпізнавальні (або аналітичні) граматики — перші задають правила, за допомогою яких можна побудувати довільне слово із мови, а другі - за заданим словом визначити, чи входить воно до складу мови. Усі символи граматики можна умовно розділити на термінали і нетермінали.

**Формальною граматикою** називаємо

- термінальний алфавіт

Vн – нетермінальний алфавіт

*-* аксіома (початковий символ)

P ϵ {ui -> vi} - правила граматики

ui, vi – будь-яка комбінація термінальних і нетермінальних символів

„**Термінал (термінальний символ)** — об’єкт, що безпосередньо міститься в мові, яка відповідає даній граматиці, та має конкретне незмінне значення (загально «літери»). У формальних мовах в якості терминалів зазвичай беруть усі або частину стандартних символів — латинські літери, цифри та спеціальні символи.

**Нетермінал (нетермінальний символ)** — об’єкт, що означає деяку конструкцію мови (наприклад: формулу, арифметичний вираз, тощо) та не має конкретного незмінного значення. „ [4]. Зазвичай позначують грецькими літерами.

**Породжувальні граматики**

Словами мови, що задана за допомогою деякої граматики є послідовності терміналів, що виводяться з первинного нетерміналу за допомогою правил виводу.

**Виведення (вивід)**

Виводом називають послідовність рядків, що складаються із терміналів та нетерміналів, де першим міститься рядок, що складається з одного стартового нетерміналу, а кожний наступний рядок отримується із попереднього шляхом заміни деякого під рядка за допомогою використання одного з правил граматики. Кінцевим вважається рядок, що складений виключно з терміналів.

При проведенні аналізу потоку лексем розрізняють висхідний та низхідний методи розбору. При низхідному - розбір проводиться з першої лексеми заданого нетерміналу та продовжується до отримання задовільної послідовності слів, що задовольняють деякому граматичному правилу мови. При висхідному – навпаки, розбір починається з останньої лексеми в деякому нетерміналі та продовжуємо розбір у східному напрямку.

Низхідний розбір можна розглядати як спробу побудови дерева розбору вхідного рядка, починаючи із кореню та створення вершин у прямому порядку. Розглянемо низхідний аналіз у загальному вигляді, а саме — аналіз методом рекурсивного спуска, котрий може використовувати відкати, тобто проводити повторне сканування вхідного потоку. Однак синтаксичні аналізатори з відкатом зустрічаються нечасто. Одна з причин цього полягає в тому, що для аналізу конструкцій реальних мов про­грамування відкат буде потрібен дуже рідко. В ситуаціях з аналізом природніх мов аналіз з відкатом є не дуже еффективним, а приоритетними є табличні методи, наприклад, метод динамічного програмування, тощо.

Нижче приведено приклад, в якому відкат все ж потрібен. Тут же пропонується спосіб дослідження вхідного потоку при відкаті.

Приклад:

Розглянемо граматику

S -> cAd

А —> ab | а

Та вхідний рядок ***w = cad.*** При низхідній побудові дерева розбору для цього рядка ми спочатку створюємо дерево, що складається з одного вузла, поміченого як ***S***. Покажчик входу вказує на ***с,*** перший символ рядка ***w.*** Тепер використаємо першу продукцію для ***S,*** щоб отримати дерево, що проілюстровано на рис. 3.

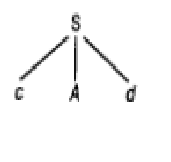


Рисунок 3. Дерево розбору при вхідному рядку w = cad

Крайній лівий лист, ***с***, відповідає першому символу ***w,*** перемістимо покажчик входу на ***а,*** *2* символ рядка ***w,*** та розглянемо наступний лист дерева, що помічений як ***А.*** Тепер можна застосувати для ***А*** першу альтернативу та отримати дере­во, що зображено на рис. 4.

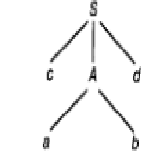


Рисунок 4. Дерево розбора при першій альтернативі

Тобто ми виявимо відповідність зчитаного символу ***а*** листу дерева. Перейдемо до наступного символу — ***d.*** Однак ***d*** не відповідає листу дерева ***b,*** а це означає, що ми повинні зробити відкат до ***А*** для того, щоб обрати нову альтернативу для роботи.

Повертаючись до ***А,*** ми повинні повернути покажчик входу на позицію 2, в якій він був, коли ми вперше прийшли до розкладу ***А.*** Це означає, що процедура для ***А*** (процедура для нетерміналів) повинна зберігати покажчик входу в локальній змінній. При розгляді другої альтернативи для ***А*** ми отримаємо дерево, показане на рис. 5.

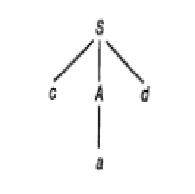


Рисунок 5. 2 альтернатива

Лист ***а*** відповідає другому символу ***w,*** а лист ***d —*** третьому. Оскільки в цей момент ми побудували дерево розбору для ***w***, ми припиняємо роботу та повідомляємо про успішне завершення розбору.

## Семантичний аналіз

Дерево розбору або інші результати синтаксичного блоку аналізу оброблюються з метою встановлення його семантики (змісту) — наприклад, прив’язка ідентифікаторів до їхніх декларацій, перевірка відповідності типів, визначення типів виразів та іншого. Результати зазвичай називають «проміжним представленням/кодом», та можуть бути доповнені деревом розбору, якимось новим деревом, абстрактним набором команд або чимось ще, що б було зрозуміло та зручно для подальшої розробки.

Для перевірки відповідності типів виразів, ідентифікаторів та констант використовується структура таблиця атрибутів.

Таблиця атрибутів – це таблиця, що містить дані про типи та декларації ідентифікаторів і відповідно при зустрічі в процесі аналізу деякого ідентифікатору, ми можемо звернутися до цієї таблиці, щоб перевірити наявність об’явленого ідентифікатора, відповідності типів, якщо цей ідентифікатор ми зустріли в деякому виразі.

Семантичний аналіз часто робиться за допомогою апарату семантичних дій. Це означає, що по тексту граматики проставляються позначення дій, які треба виконувати під час аналізу конкретного слова.

## Генерація машинних команд

Останньою стадією моделі компіляції є генератор коду, який одержує на вхід оброблене попередніми стадіями мовного процесору проміжне представлення вихідної програми й виводить еквівалентну цільову програму. Традиційно до генератора коду пред'являються жорсткі вимоги. Одержуваний код повинен бути коректним і високоякісним, що означає ефективне використання ресурсів цільової машини. Крім того, ефективно повинен працювати й сам генератор коду.  
Хоча дрібні деталі генератора коду залежать від цільової машини й операційної системи, питання на зразок керування пам'яттю, вибір інструкцій, розподіл регістрів і порядок обчислень, властиві практично всім завданням, пов'язаним з генерацією коду.

**Вхід генератора коду. [4].**

Вхідний потік генератора кода являє собою проміжне представлення вихідної програми, отримане на початковій стадії компіляції, разом з таблицею символів, яка використовується для визначення адрес, часу виконання об'єктів даних, позначуваних у проміжному представленні іменами.

Ми вважаємо, що перед генерацією коду початковою стадією компіляції було виконане сканування, розбір і трансляція вихідної програми в докладне проміжне представлення, так що значення імен у проміжній мові можуть бути представлені величинами, з якими цільова машина може працювати безпосередньо (біти, цілі й дійсні числа, тощо).

Ми передбачаємо, що були виконані всі необхідні перевірки типів, всі необхідні оператори перетворення типів перебувають на своїх місцях і були виявлені всі очевидні семантичні помилки. Тобто стадія генерації коду може працювати в припущенні, що її вхід не містить помилок.

**Цільові програми [4].**

Виходом генератора коду є цільова програма. Подібно проміжному коду, вихід генератора коду може бути різних видів: абсолютна машинна мова, переміщувана машинна мова, або мова асемблера. Перевагою генерації абсолютної програми машинною мовою є те, що код розміщується у фіксоване місце в пам'яті й негайно виконується; невеликі програми при цьому швидко компілюються й виконуються. Генерація переміщуваної програми машинною мовою (об'єктного модуля) забезпечує можливість роздільної компіляції підпрограм. Безліч переміщуваних об'єктних модулів можуть бути потім зв'язані в єдине ціле й завантажені для виконання спеціальною програмою - єднальним завантажником. Додаткові витрати на зв'язування й завантаження компенсуються можливістю роздільної компіляції підпрограм і виклику інших, раніше скомпільованих підпрограм з об'єктних модулів. Якщо цільова машина не обробляє переміщення автоматично, компілятор повинен надати завантажнику явну інформацію про переміщення для зв'язування сегментів роздільно скомпільованих підпрограм.

# Реалізація

## Вибір і обґрунтування мови реалізації

Під час проектування майбутнього продукту було проведено детальний аналіз декількох популярних мов програмування з метою вибору кращого середовища реалізації. У полі зору автора опинились

* Java
* Perl
* Python
* Delphi
* C++

В процесі вивчення особливостей вищевказаних мов було прийнято рішення зупинити свій вибір на мові Python через цілий ряд переваг:

* Мова програмування високого рівня
* Об’єктно-орієнтованість
* Дуже зручні типи даних
* Велика швидкість роботи завдяки використанню бібліотек, реалізованих на чистому C
* Простий і зрозумілий синтаксис
* Крос-платформеність
* Прекрасні засоби роботи з YAML
* Вбудована підтримка регулярних виразів

YAML був обраний для зберігання інформації у зовнішніх конфігураційних файлах через свої виняткові переваги перед подібними форматами (XML і т. д.), в першу чергу читабельність і здатність до легкої модифікації даних користувачем.

## Лексичний аналізатор

### Механізм регулярних виразів

Лексичний аналізатор на вхід отримує код на мові Pascal, і виділяє з вхідного потоку лексеми, які потім подаються на вхід парсеру. Для розпізнавання цих лексем ми використовуємо механізм регулярних виразів. Вся інформація відносно ключових слів, символів-розділювачів, класів символів зберігається у конфігураційному файлі metadata.yml у форматі YAML.

Класи символів розрізняємо за допомогою регулярних виразів. Запишемо спочатку правила розрізнення класів лексем у вигляді алгебри регулярних подій.

**Ідентифікатор**

ID = letter{letter|digit}\*

letter = A|B|C|…|Z

digit = 0|1|2|…|9

**Ціла константа**

intConst = (+|-|ε)digit+

**Дійсна константа**

floatConst = (+|-|ε)digit+.digit+((ε)|(e(+|-)digit+))

**Строковий літерал**

stringConst = (letter|digit|\_|!|@|#|$|%|&|\*|<|>|/|)\*

Тепер перепишемо їх у вигляді регулярних виразів мови Python.

**Ідентифікатор**

ID = '^[A-Za-z][A-Za-z\_0-9]{0,255}$'

**Ціла константа**

intConst = '^[+-]?\d{1,10}$'

**Дійсна константа**

floatConst = '^([+-]?((\d+\.\d+)|(\d+\.\d+e[+-]\d+)))$'

**Строковий літерал**

stringConst = "^'.{0,65535}'$"

Ключові слова

AND ARRAY BEGIN FORWARD DIV DO ELSE END FOR FUNCTION IF MOD NOT OF OR PROCEDURE PROGRAM RECORD THEN TO TYPE VAR WHILE

Повністю приведено файл metadata.yml у додатку А.

### Алгоритм виділення лексем

Приведемо алгоритм розпізнавання лексем, який використовує вищезгаданий файл конфігурації.

def getClass(self, word):

"""

Effects: Returns word class

"""

c = None

if ( (word in self.KeyWords)

or (word in metadata["delimiters"])

or (word in metadata["double"])

or (word in metadata["conditional\_delimiters"])

or (word in metadata["multiplicative"])

or (word in metadata["additive"])

or (word in metadata["Relation"])

):

c = word

else:

for r in self.RegExp.keys():

if re.compile(r).match(word):

c = self.RegExp[r]

return c

Цей алгоритм перевіряє, з яким з регулярних виразів співпадає лексема. Якщо з жодним не співпадає, повертається None.

# Синтаксичний аналізатор

## Граматика мови

Використовувана граматика мови Паскаль є контекстно вільною граматикою. **Контекстно-вільною**, або **КВ-граматикою**, називається граматика, в якій ліві частини всіх продукцій є нетерміналами. Зміст терміну "контекстно-вільна" полягає в тім, що застосування продукції A® w до ланцюжка uAv не залежить, тобто є **вільним** від сусідніх з A символів, які утворюють **контекст** uv.

Зазначимо, що БНФ вигляду A::=w цілком аналогічна продукції A® w. Отже, сукупності БНФ є просто іншою формою КВ-граматик.

**Контекстно-вільною мовою** (**КВ-мовою**) називається мова, що може бути задана КВ-граматикою.

Крім того, граматика є лівоасоціативною, тобто при обробці виразу a + b + d спочатку підраховується сума a + b, а потім вже до цієї суми додаємо d.

## Формат граматики мови у РБНФ

Приведемо граматику реалізованої мови Паскаль у вигляді розширених Бекусо-Науровських форми.

*программа* ::= **Program** **ID ;** *Блок* **.**

*Блок*::= [ *Определение\_типов* ]

[ *Определение\_констант* ]

[ *Объявление\_переменных* ]

[ *Объявление\_процедур\_и\_функций* ]

*Составное\_действие*

*Определение\_типов* ::= **Type**  *Определение\_типа* **;**

{ *Определение\_типа* **;** }

*Определение\_констант* ::=**Const** *Определение\_константы* **;**

{ *Определение\_константы* **;** }

*Объявление\_переменных* ::= **Var** *Объявление\_переменной* **;**

*{ Объявление\_переменной* **;***}*

*Объявление\_процедур\_и\_функций* ::=

{ (*Объявление\_процедуры* **|** *Объявление\_функции*) **;**}

*Определение\_типа* ::= **ID** **=** *Тип*

*Определение\_константы* ::= **ID** **=** *Константа*

*Объявление\_переменной* ::= *Список\_идентификаторов* **:** *Тип*

*Объявление\_процедуры* ::=

**Procedure** **ID** **(** *Список\_формальных\_параметров* **)** **;**

( *Блок* ***|* Forward** )

*Объявление\_функции* ::=

**Function** **ID** **(** *Список\_формальных\_параметров* **)** **:**

*Тип\_результата* **;** ( *Блок* ***|* Forward** )

*Список\_формальных\_параметров* ::=

[ *Список\_идентификаторов* **:** **ID**

{**;** *Список\_идентификаторов* **:** **ID**} ]

*Составное\_действие* ::= **begin**

*Последовательность\_действий*

**end**

*Последовательность\_действий* ::=

*Действие* { **;** *Действие }*

*Действие* ::= *Простое\_действие | Сложное\_действие*

*Простое\_действие* ::= [ **ID** (*Действие\_присваивания |*

*Действие\_вызова\_процедуры*) ]

*Действие\_присваивания* ::= *Переменная* **:=** *Выражение*

*Действие\_вызова\_процедуры* ::=

**(** *Список\_фактических\_параметров* **)**

*Сложное\_действие*::= *Составное\_действие |*

**IF** *Выражение* **THEN** *Действие* [ **ELSE** *Действие* ] |

**WHILE** *Выражение* **DO** *Действие* |

**FOR** **ID** **:=** *Выражение* **TO** *Выражение* **DO** *Действие*

*Тип* ::= **ID** |

**ARRAY** **[** *Диапазон\_индекса* **]** **OF** *Тип* |

**RECORD**  *Список\_полей* **END** |

*Константа* ***..*** *Константа*

*Диапазон\_индекса* ::=**ID** |

*Константа* ***..*** *Константа*

*Тип\_результата*::= **ID**

*Список\_полей* ::= [ *Список\_идентификаторов* **:** *Тип*

*{* ***;*** *Список\_идентификаторов* **:** *Тип* } ]

*Константа* ::= [ *Знак* ] **INT** | **STRCONST**

*Выражение*::=

*Простое\_выражение* [ *Знак\_отношения Простое\_выражение* ]

*Знак\_отношения* ::= **<** | **<=** | **>** | **>=** | **<>** | **=**

*Простое\_выражение* ::= [ *Знак* ] *Слагаемое*

*{ Аддитивная\_операция Слагаемое }*

*Аддитивная\_операция* ::= + | - | **or**

*Слагаемое*::= *Множитель*

*{ Мультипликативная\_операция Множитель }*

*Мультипликативная\_операция* ::= **\*** | div | **mod** | **and**

*Множитель*::= **INT** | **STRCONST** | *Переменная |*

*Вызов\_функции |* **not** *Множитель* |

**(** *Выражение* **)**

*Вызов\_функции* ::= /\*ID\*/ **(** *Список\_фактических\_параметров* **)**

*Переменная* ::= /\*ID\*/[ **.** **ID** Переменная

**[** *Выражение* **]** Переменная]

*Список\_фактических\_параметров* ::=

[ *Выражение* { **,** *Выражение* } ]

*Список\_идентификаторов* ::= **ID** { **,** **ID** }

*Знак* ::= + | -

### Формат зберігання РБНФ у конфігураційному файлі

Ми вирішили винести всю граматику мови у зовнішній файл у вигляді метаданих. Отримали параметризований мовний процесор, тобто граматику мови можна модифікувати без втручання у внутрішній код програмного продукту. Автором був розроблений метод запису РБНФ у форматі YAML. YAML був вибраний через виняткову читабельність і здатність до модифікації, і, звичайно, переносу на іншу платформу. Розглянемо, як записати правило граматики у форматі РБНФ до конфігураційного файлу.

* **Вигляд правила у форматі РБНФ**

программа ::= Program ID ; Блок .

* **Вигляд правила у конфігураційному файлі**

programme:

- [program, id, ;, Block, "."]

Повністю файл з правилами граматики Rules.yml приведемо у додатку В.

### Алгоритм зчитування і аналізу граматики з конфігураційного файлу

Для зчитування граматики мови з конфігураційного файлу був розроблений алгоритм. Він зчитує лексему зі входу і звіряє її з підходящим правилом, взятим із граматики мови.

def checkAccordingToRule(self, inRuleItem):

"""

Effects: Checks programme on syntax correctness (one RuleItem)

"""

RuleItem = self.subSequence(inRuleItem, 0, len(inRuleItem)-1)

i = 0

while i < len(RuleItem):

self.ruleSymbol = RuleItem[i]

if self.ruleSymbol == self.inputSymbol: self.addToDictionary()

if self.ruleSymbol == "[[":

index = self.findElement(RuleItem, "]]", i)

if self.isDerivedFrom(RuleItem[i+1]):

RuleItem.pop(i)

RuleItem.pop(index-1)

i -= 1

else:

i= index

elif self.ruleSymbol == "{":

index = self.findElement(RuleItem, "}", i)

self.RepeatableSymbols = self.subSequence(RuleItem, i+1, index-1)

if self.isDerivedFrom(RuleItem[i+1]):

# Insert RepeatableSymbols into RuleItem with index i

\_i = i

for reSymbol in self.RepeatableSymbols:

RuleItem.insert(\_i, reSymbol)

\_i = \_i + 1

i -= 1

else:

i = index

else:

try:

self.treatSymbol()

except E.ENotFoundRule:

return None

i += 1

Цей алгоритм зчитує символ з вхідного потоку, одночасно з цим бере наступне слово з правила граматики і звіряє вхідний поток з правилом. При співпадінні вхідного потоку з правилом алгоритм зчитує наступний символ.

## Семантичний аналізатор

### Опис семантичних дій

При реалізації семантичного аналізу ми вирішили використати підхід семантичних дій. По тексту граматики поряд з потрібним правилом ми ставимо позначку семантичної дії, в якій робимо певну роботу по перевірці семантичної коректності програми і підготовлюємо інформацію для подальшої генерації коду. Всі семантичні дії можна розділити на 2 класи: накоплюючі дії і обробні дії. Перші зазвичай лише збирають інформацію для подальшого використання, звичайно в якості структур даних використовують стек. 2 група дій бере цю інформацію зі стека і використовує її для перевірки правильності програми. Для зручності ми ввели 4 семантичних стеки в якості структур даних, потрібних при цій стадії. Розглянемо призначення кожного.

* Для зберігання даних
* Довжина даних
* Перелік дій одного рівня
* Перехід між рівнями

### Алгоритм зчитування правил граматики з обробкою семантичних дій

При реалізації семантичного аналізатора ми проставили у файлі семантичні дії поряд з потрібними частинами правил і модифікували алгоритм обробки граматики з ціллю враховувати семантичні дії. Ось як виглядає модифікований алгоритм.

def checkAccordingToRule(self, inRuleItem):

"""

Effects: Checks programme on syntax correctness (one RuleItem)

"""

RuleItem = self.subSequence(inRuleItem, 0, len(inRuleItem)-1)

i = 0

while i < len(RuleItem):

self.ruleSymbol = RuleItem[i]

self.nextRuleSymbol = self.getRuleSymbol(RuleItem, i)

if self.isSemanticAction(self.ruleSymbol):

i += 1

continue

self.doSemanticAction(RuleItem, i)

if self.nextRuleSymbol == "0":

i = i + 1

continue

if self.ruleSymbol == self.inputSymbol: self.addToDictionary() if self.ruleSymbol == "[[":

index = self.findElement(RuleItem, "]]", i)

if self.isDerivedFrom(RuleItem[i+1]):

RuleItem.pop(i)

RuleItem.pop(index-1)

i -= 1

else:

i= index

elif self.ruleSymbol == "{":

index = self.findElement(RuleItem, "}", i)

self.RepeatableSymbols = self.subSequence(RuleItem, i+1, index-1)

if self.isDerivedFrom(RuleItem[i+1]):

# Insert RepeatableSymbols into RuleItem with index i

\_i = i

for reSymbol in self.RepeatableSymbols:

RuleItem.insert(\_i, reSymbol)

\_i = \_i + 1

i -= 1

else:

i = index

else:

try:

self.treatSymbol()

except E.ENotFoundRule:

return None

i += 1

Крім описаного вище, цей алгоритм враховує семантичні дії. Якщо символу приписується семантична дія, алгоритм виконує її.

Модифікований файл у форматі YAML можна переглянути у додатку Б.

# Внутрішні структури

## Класи елементів мови

Задачею бакалаврської роботи є в тому числі і підготовка до завершальної стадії компілятора – генерації коду. Для цього у процесі роботи мовного процесора під час попередніх стадій створюються об’єкти для подальшого їх використання на кінцевій стадії. Як приклад можна привести об’єкт класу

AttrFor. Цикл for в Паскалі виглядає як

For parameter := start\_value to end\_value do

begin

// Do everything

End

При обробці цього коду створюється об’єкт класу AttrFor.

class AttrFor(AttrObject):

def \_\_init\_\_(self):

AttrObject.\_\_init\_\_(self, "AttrFor", "")

self.parameter = None

self.first = None

self.last = None

self.step = None

self.body = None

І заповнюються всі вказані параметри.

Приведемо повний перелік усіх класів.

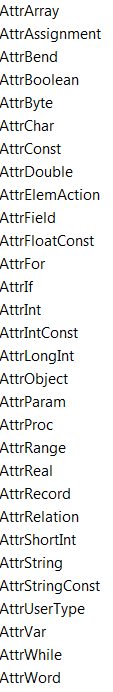
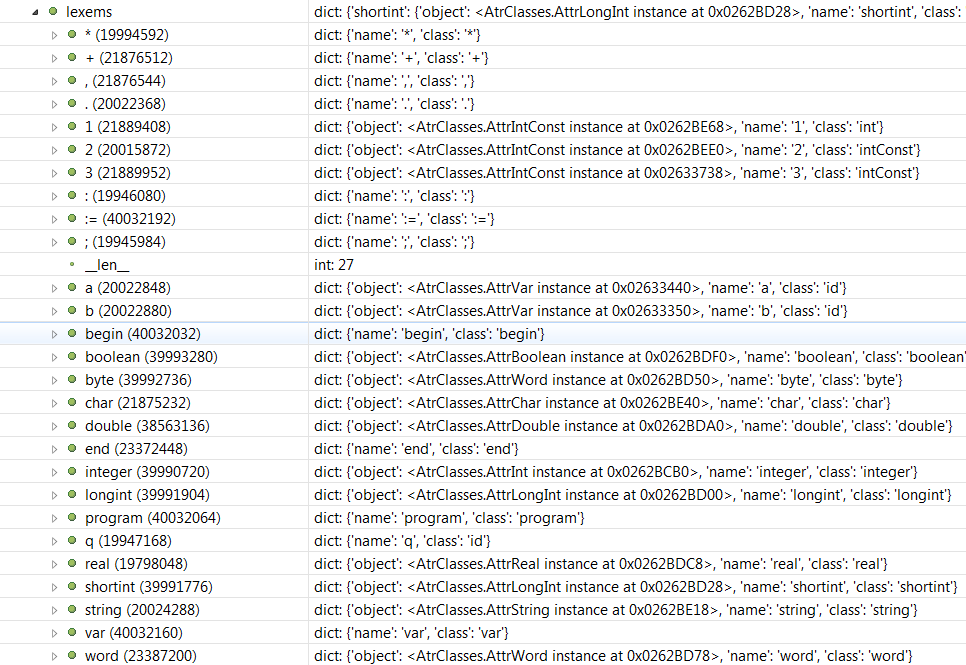


Рисунок 6. Перелік усіх атрибутних класів

## Таблиця атрибутів

Таблиця атрибутів є однією з найважливіших структур даних у мовному процесорі. Вона зберігає інформацію щодо усіх лексем, їх атрибутів, вона використовується впродовж роботи усіх стадій програмного продукту.

Рисунок 7. Таблиця атрибутів

# Аналіз отриманих результатів. Тестування

Проведемо тестування програмного продукту на всіх стадіях його роботи. Спочатку перевіримо правильність роботи лексичного аналізатора. Подаємо на вхід файл з кодом

program q;

var

a@, b: integer;

d: real;

begin

d:=a;

for a:= 1 to 2+2 do

begin

b:=1;

a:=2;

end;

d:=a;

end.

Мовний процесор видає інформацію про помилку, вказує помилкове слово і рядок у програмі, де воно зустрілося.



Рисунок 8. Лексичний аналізатор, аналіз помилок

Наступною стадією перевірки є перевірка синтаксису. Синтаксичний аналізатор видає помилку, якщо лексеми стоять у неправильному порядку. Подамо файл з пропущеною «;»

program q;

var

a, b: integer;

d: real;

begin

d:=a;

for a:= 1 to 2+2 do

begin

b:=1;

a:=2;

end

d:=a;

end.

Результат виконання мовного процесору виглядає наступним чином:



Рисунок 9. Вихід мовного процесора при помилці синтаксису

program q;

var

a, b: integer;

d: real;

begin

d:=a;

for a:= 1 2+2 do

begin

b:=1;

a:=2;

end;

d:=a;

end.



Рисунок 10. Вихід мовного процесора при помилці

в оформленні циклу for

Далі виконується перевірка семантики, перевірка типів, об’явлених змінних, умов у циклах, подвійного об’явлення типів.

program q;

type

my = record

a1: integer;

a2: double;

end;

var

a, b: my2;

d: real;

begin

for a:= 1 to 2+2 do

begin

b:=1;

a:=2;

end;

d:=a;

end.



Рисунок 11. Помилка при об’явленні типів

program q;

type

my = record

a1: integer;

a2: double;

end;

var

a, b: my;

d: real;

begin

for a:= 1 to 2+2 do

begin

b:=1;

a:=2;

end;

d:=a;

end.



Рисунок 12. Помилка при використанні типів

program q;

type

my = record

a1: integer;

a2: double;

end;

var

a, b: integer;

d: real;

begin

for a:= 9 to 2+2 do

begin

b:=1;

a:=2;

end;

d:=a;

end.



Рисунок 13. Помилка в циклі for

program q;

type

my = record

a1: integer;

a2: double;

end;

var

a, b: integer;

d: real;

begin

if 2+2 then a:=2;

for a:= 1 to 2+2 do

begin

b:=1;

a:=2;

end;

d:=a;

end.



Рисунок 14. Помилка в умові

program q;

type

my = record

a1: integer;

a2: double;

end;

my = record

a1: integer;

a2: double;

end;

var

a, b: integer;

i: integer;

d: real;

begin

d := 4;

for i:= 1 to (2+2\*2)\*2 do

begin

b:=1;

a:=2;

end;

d:=a;

end.



Рисунок 15. Подвійне об’явлення типів

program q;

type

my = record

a1: integer;

a2: double;

end;

var

a, b: integer;

i: integer;

d: real;

begin

d := 4;

for i:= 1 to (2+2\*2)\*2 do

begin

b:=1;

a:=2;

end;

d:=a;

end.

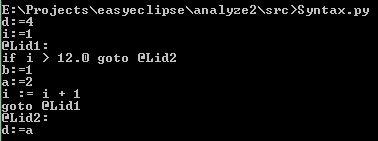


Рисунок 16. Вихі мовного процесора при коректному файлі, проміжний код

# Висновки

В ході роботи було реалізовано лексичний аналізатор, в якому використовуються регулярні вирази, синтаксичний аналізатор методом рекурсивного спуску з використанням граматики у формі РБНФ у конфігураційному файлі, семантичний аналізатор з використанням методу семантичних дій, проставлених біля кожного правила в граматиці. Підготовлено всі дані для генерації машинного коду. Важливо відмітити, що мовний процесор є параметризований граматикою мови, тобто можна модифікувати граматику без модифікації вихідного коду програмного продукту. Мовний процесор є однопрохідним, використана схема синтаксично керованого компілятора. В якості мови реалізації автором був вибраний Python через цілий ряд переваг, в якості формату конфігураційних файлів використовується формат YAML через виняткову читабельність і легку модифікацію. Отримані під час тестування результати доводять правильність роботи програмного продукту. Результати роботи будуть використані при вивченні курсу «Алгоритмічні мови та мовні процесори».

# Література

1. Ахо А, Сеті Р., Ульман Д. Компілятори. Принципи, технології, інструменти. – Москва, Санки Петербург, „Вільямс” Київ 2003.
2. Андреев А.М.,Березкин Д.В.,Брик А.В.,Смирнов Ю.М. Про один з методів написання синтаксичного аналізатора текстів на природній мові // „Інтелтек видавництво”. – відомини ВНЗ – „Приладобудівництво” 1995, №5.
3. Карпова Г.Д., Пирогова Ю.К., Кобзарева Т.Ю., Микаэлян Є.В. Комп’ютерний синтаксичний аналіз: опис моделей та направлень розробок. // Підсумки науки і техніки (серія "Обчислювальні науки"). Т.6. -М.: ВИНИТИ, 1991.
4. Кас’янов В.Н. Лекції з теорії формальних мов, автоматів та складності обчислювань. - Новосибірськ: НГУ. - 1995. - 112 с.
5. Конспект лекцій «Алгоритмічні мови і мовні процесори»
6. Лутц, Марк. Программирование на Python: Перевод с английского (+CD). — СПб.: Символ-Плюс, 2002. — 1136 с
7. Mark Lutz, David Ascher. Learning Python. O’Reilly, 1999
8. http://compilers.da.ru/
9. http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80
10. http://wiki.cs-mapping.com.ua/index.php/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80%D1%8B
11. http://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9\_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7
12. http://ru.wikipedia.org/wiki/%D0%9B%D0%B5%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9\_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7
13. http://ru.wikipedia.org/wiki/YAML
14. http://ru.wikipedia.org/wiki/Regexp
15. www.python.org
16. http://ru.wikipedia.org/wiki/Python

# Додатки

# Додаток А. Файл metadata.yml

programme:

- [program, id, ;, Block, "."]

Block:

- ["[[", types\_definition, "]]", "[[", constants\_definition, "]]", "[[", vars\_definition, "]]", "[[", procedures\_and\_f\_definition, "]]", complex\_action]

types\_definition:

- [type, type\_definition, ;, "{", type\_definition, ;, "}" ]

constants\_definition:

- [const, const\_definition, ;, "{", const\_definition, ;, "}"]

vars\_definition:

- [var, var\_definition, ;, "{", var\_definition, ;, '}']

procedures\_and\_f\_definition:

- ["{", procedure\_definition, ;, '}']

- ['{', f\_definition, ;, "}"]

type\_definition:

- [id, "#8000", "=", \_type]

const\_definition:

- ["id", "=", constant]

var\_definition:

- [id\_list, ":", \_type]

procedure\_definition:

- [procedure, id, (, formal\_list, ), ;, Block]

- [procedure, id, (, formal\_list, ), ;, forward]

f\_definition:

- [function, id, (, formal\_list, ), ":", id, ;, Block]

- [function, id, (, formal\_list, ), ":", id, ;, forward]

formal\_list:

- ["[[", id\_list, ":", id, "{", ;, id\_list, ":", id, "}", "]]"]

- ["[[", var, id\_list, ":", id, "{", ;, id\_list, ":", id, "}", "]]"]

complex\_action:

- [begin, action\_list, end]

action\_list:

- [action, ;, "{", action, ;, "}"]

action:

- [easy\_action]

- [hard\_action]

easy\_action:

- ["[[", id, action1, "]]"]

action1:

- [asnmt]

- [procedure\_call]

asnmt:

- [ "[[", component\_choice, "]]", ":=", expr]

procedure\_call:

- [(, fact\_list, )]

hard\_action:

- [complex\_action]

- [if, expr, then, action, "[[", else, action, "]]"]

- [while, expr, do, action, "#310,280"]

- [for, id, ":=", expr, to, expr, do, action, "#310,280"]

\_type:

- [id]

- [array, "[", index\_diapason, "]", of, id]

- [record, field\_list, end]

- [intConst, "..", intConst]

index\_diapason:

- [id]

- [intConst, "..", intConst]

res\_type:

- [id]

field\_list:

- ["[[", id\_list, ":", \_type, "{", ;, id\_list, ":", \_type, "}", "]]"]

constant:

- ["[[", sign, "]]", "intConst"]

- ["[[", sign, "]]", "StringConst"]

expr:

- [easy\_expr, "[[", Relation, easy\_expr, "]]"]

easy\_expr:

- ["[[", sign, "]]", add\_item, "{", additive, add\_item,"}"]

add\_item:

- [m\_item, "{", Multiplicative, m\_item, '0',"}"]

m\_item:

- [intConst]

- [floatConst]

- [StringConst]

- [variable]

- [f\_call]

- [not, m\_item]

- [(, expr, )]

f\_call:

- [id, (, fact\_list, )]

variable:

- [id, component\_choice]

component\_choice:

- ["[[", ., id, component\_choice, "]]"]

- ["[[", "[", expr, "]", component\_choice, "]]"]

fact\_list:

- ['[[', expr, "{", ",", expr, "}", "]]"]

id\_list:

- [id, '{', ",", id, "}"]

sign:

- [+]

- [-]

Relation:

- ["<"]

- ["<="]

- [">"]

- [">="]

- ["<>"]

- ["="]

additive:

- ["+"]

- ["-"]

- ["or"]

- ["xor"]

Multiplicative:

- [div]

- ["\*"]

- [mod]

- [and]

# Додаток Б. Файл Rules.yml

programme:

- [program, id, ;, Block, "."]

Block:

- ["[[", types\_definition, "]]", "[[", constants\_definition, "]]", "[[", vars\_definition, "]]", "[[", procedures\_and\_f\_definition, "]]", complex\_action]

types\_definition:

- [type, type\_definition, ;, "#7000", "{", type\_definition, ;, "#7000", "}" ]

constants\_definition:

- [const, const\_definition, ;, "#4000", "{", const\_definition, ;, "#4000", "}"]

vars\_definition:

- [var, var\_definition, ;, "#6000", "{", var\_definition, ;, "#6000", '}']

procedures\_and\_f\_definition:

- ["{", procedure\_definition, ;, '}']

- ['{', f\_definition, ;, "}"]

type\_definition:

- [id, "#8000", "=", \_type]

const\_definition:

- ["id", "#1000", "=", constant]

var\_definition:

- [id\_list, ":", \_type]

procedure\_definition:

- [procedure, "#16", id, "#18", (, formal\_list, ), "#23", ;, Block]

- [procedure, id, (, formal\_list, ), ;, forward]

f\_definition:

- [function, "#17", id, "#18", (, formal\_list, ), ":", id, "#22", ;, Block]

- [function, id, (, formal\_list, ), ":", id, ;, forward]

formal\_list:

- ["[[", id\_list, ":", id, "#21", "{", ;, id\_list, ":", id, "#21", "}", "]]"]

- ["[[", var, id\_list, ":", id, "#21", "{", ;, id\_list, ":", id, "#21", "}", "]]"]

complex\_action:

- [begin, "#200", action\_list, end, "#220"]

action\_list:

- [action, "0", "#210", ;, "{", action, ;, "}"]

action:

- [easy\_action]

- [hard\_action]

easy\_action:

- ["[[", id, "#30", action1, "]]"]

action1:

- [asnmt]

- [procedure\_call]

asnmt:

- [ "[[", component\_choice, "]]", ":=", expr, "0", "#31"]

procedure\_call:

- [(, fact\_list, )]

hard\_action:

- [complex\_action]

- [if, "#230", expr, "0", "#240", then, "#250", action, "0", "#260", "[[", else, "#250", action, "0", "#270", "]]"]

- [while, "#290", expr, "0", "#300", do, action, "0", "#310,280"]

- [for, "#311", id, "#330", ":=", expr, "0", "#340", to, "#350", expr, "0", "#360", do, "#250", action, "0", "#310,280"]

\_type:

- [id, "#9000"]

- [array, "#15000", "[", index\_diapason, "]", "#16000", of, id, "#17000"]

- [record, "#18000", field\_list, end, "#19000"]

- [intConst, "#12000", "..", intConst, "#10000"]

index\_diapason:

- [id, "#11000"]

- [intConst, "#12000", "..", intConst, "#14000"]

res\_type:

- [id]

field\_list:

- ["[[", id\_list, ":", \_type, "{", ;, id\_list, ":", \_type, "}", "]]"]

constant:

- ["[[", sign, "]]", "intConst", "#2000"]

- ["[[", sign, "]]", "StringConst", "#3000"]

expr:

- [easy\_expr, "[[", Relation, easy\_expr, "0", "#2999", "]]"]

easy\_expr:

- ["[[", sign, "]]", add\_item, "{", additive, add\_item, "0", "#29","}"]

add\_item:

- [m\_item, "{", Multiplicative, m\_item, '0', "#29","}"]

m\_item:

- [intConst, "#26"]

- [floatConst, "#26"]

- [StringConst, "#26"]

- [variable]

- [f\_call]

- [not, m\_item]

- [(, expr, )]

f\_call:

- [id, (, fact\_list, )]

variable:

- [id, "#26", component\_choice]

component\_choice:

- ["[[", ., id, component\_choice, "]]"]

- ["[[", "[", expr, "]", component\_choice, "]]"]

fact\_list:

- ['[[', expr, "{", ",", expr, "}", "]]"]

id\_list:

- [id, "#55000", '{', ",", id, "#5000", "}"]

sign:

- [+]

- [-]

Relation:

- ["<", '#2992']

- ["<=", '#2992']

- [">", '#2992']

- [">=", '#2992']

- ["<>", '#2992']

- ["=", '#2992']

additive:

- ["+", "#28"]

- ["-", "#28"]

- ["or", "#28"]

- ["xor", "#28"]

Multiplicative:

- [div, "#28"]

- ["\*", "#28"]

- [mod, "#28"]

- [and, "#28"]

# Додаток В. Вигляд правил граматики без семантичних дій

programme:

- [program, id, ;, Block, "."]

Block:

- ["[[", types\_definition, "]]", "[[", constants\_definition, "]]", "[[", vars\_definition, "]]", "[[", procedures\_and\_f\_definition, "]]", complex\_action]

types\_definition:

- [type, type\_definition, ;, "{", type\_definition, ;, "}" ]

constants\_definition:

- [const, const\_definition, ;, "{", const\_definition, ;, "}"]

vars\_definition:

- [var, var\_definition, ;, "{", var\_definition, ;, '}']

procedures\_and\_f\_definition:

- ["{", procedure\_definition, ;, '}']

- ['{', f\_definition, ;, "}"]

type\_definition:

- [id, "#8000", "=", \_type]

const\_definition:

- ["id", "=", constant]

var\_definition:

- [id\_list, ":", \_type]

procedure\_definition:

- [procedure, id, (, formal\_list, ), ;, Block]

- [procedure, id, (, formal\_list, ), ;, forward]

f\_definition:

- [function, id, (, formal\_list, ), ":", id, ;, Block]

- [function, id, (, formal\_list, ), ":", id, ;, forward]

formal\_list:

- ["[[", id\_list, ":", id, "{", ;, id\_list, ":", id, "}", "]]"]

- ["[[", var, id\_list, ":", id, "{", ;, id\_list, ":", id, "}", "]]"]

complex\_action:

- [begin, action\_list, end]

action\_list:

- [action, ;, "{", action, ;, "}"]

action:

- [easy\_action]

- [hard\_action]

easy\_action:

- ["[[", id, action1, "]]"]

action1:

- [asnmt]

- [procedure\_call]

asnmt:

- [ "[[", component\_choice, "]]", ":=", expr]

procedure\_call:

- [(, fact\_list, )]

hard\_action:

- [complex\_action]

- [if, expr, then, action, "[[", else, action, "]]"]

- [while, expr, do, action]

- [for, id, ":=", expr, to, expr, do, action]

\_type:

- [id]

- [array, "[", index\_diapason, "]", of, id]

- [record, field\_list, end]

- [intConst, "..", intConst]

index\_diapason:

- [id]

- [intConst, "..", intConst]

res\_type:

- [id]

field\_list:

- ["[[", id\_list, ":", \_type, "{", ;, id\_list, ":", \_type, "}", "]]"]

constant:

- ["[[", sign, "]]", "intConst"]

- ["[[", sign, "]]", "StringConst"]

expr:

- [easy\_expr, "[[", Relation, easy\_expr, "]]"]

easy\_expr:

- ["[[", sign, "]]", add\_item, "{", additive, add\_item,"}"]

add\_item:

- [m\_item, "{", Multiplicative, m\_item,"}"]

m\_item:

- [intConst]

- [floatConst]

- [StringConst]

- [variable]

- [f\_call]

- [not, m\_item]

- [(, expr, )]

f\_call:

- [id, (, fact\_list, )]

variable:

- [id, component\_choice]

component\_choice:

- ["[[", ., id, component\_choice, "]]"]

- ["[[", "[", expr, "]", component\_choice, "]]"]

fact\_list:

- ['[[', expr, "{", ",", expr, "}", "]]"]

id\_list:

- [id, '{', ",", id, "}"]

sign:

- [+]

- [-]

Relation:

- ["<"]

- ["<="]

- [">"]

- [">="]

- ["<>"]

- ["="]

additive:

- ["+"]

- ["-"]

- ["or"]

- ["xor"]

Multiplicative:

- [div]

- ["\*"]

- [mod]

- [and]