НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Кафедра автоматизації проектування енергетичних процесів і систем

**КУРСОВА РОБОТА**

з дисципліни «Лінгвістичне забезпечення САПР-2»

(назва дисципліни)

на тему: «Розробка транслятора загального призначення»

Студента групи ТР-41

напряму підготовки ***6.050101 Комп’ютерні науки***

спеціальності ***7.05010102 Інформаційні технології проектування***

\_\_\_\_\_\_\_\_\_\_Войтовича С.В.\_\_\_\_\_\_\_\_

(прізвище та ініціали)

Керівник к.т.н., доцент Медведєва В.М.

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

Національна оцінка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кількість балів: \_\_\_\_\_\_\_\_\_\_Оцінка: ECTS \_\_\_\_\_

Члени комісії \_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

Київ - 2017 рік

**Національний технічний університет України**

**“Київський політехнічний інститут імені Ігоря Сікорського”**

Факультет ТЕПЛОЕНЕРГЕТИЧНИЙ ФАКУЛЬТЕТ

( повна назва)

Кафедра автоматизації проектування енергетичних процесів та систем

(повна назва)

Освітньо-кваліфікаційний рівень бакалавр

Напрям підготовки ***6.050101 Комп’ютерні науки***

(шифр і назва)

Спеціальність ***7.05010102 Інформаційні технології проектування***

(шифр і назва)

ЗАВДАННЯ

**НА КУРСОВУ РОБОТУ СТУДЕНТУ**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Войтовичу Сергію Вікторовичу\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

(прізвище, ім’я, по батькові)

1. Тема роботи «Розробка транслятора загального призначення»

керівник курсової роботи –

Медведєва Валентина Миколаївна, канд. техн. наук, доцент

( прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

2. Строк подання студентом роботи – 7 червня 2017 р.

3. Вихідні дані до роботи: Visual Studio 2015 та MS SQL Server 2012

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) – розробити транслятор загального призначення та пояснити логіку його роботи.

5. Дата видачі завдання – 15 лютого 2017 р.

**Студент \_\_\_\_\_\_\_\_\_ Войтович С.В.**

( підпис ) (прізвище та ініціали)

**Керівник курсової роботи \_\_\_\_\_\_\_\_\_ Медведєва В.М.**

( підпис ) (прізвище та ініціали)

**КАЛЕНДАРНИЙ ПЛАН**

**виконання курсової роботи**

**студентом Войтович С.В.**

(прізвище та ініціали)

|  |  |  |  |
| --- | --- | --- | --- |
| №  з/п | Назва етапів виконання курсової роботи | Строк виконання етапів роботи | Примітка |
| **1** | Затвердження теми роботи | 15.02.2017 |  |
| **2** | Вивчення та аналіз задачі | 16.02.2017 –  28.02.2017 |  |
| **3** | Розробка лексичного аналізатора | 29.02.2017 –  03.03.2017 |  |
| **4** | Розробка синтаксичного аналізатора | 14.03.2017 –  23.03.2017 |  |
| **5** | Переведення в ПОЛІЗ усієї програми за алгоритмом Дейкстри | 28.03.2017 –  08.04.2017 |  |
| **6** | Виконання ПОЛІЗ | 28.04.2017 –  5.05.2017 |  |
| **7** | Написання пояснювальної записки | 25.05.2017 |  |

**Студент \_\_\_\_\_\_\_\_\_ Войтович С.В.**

( підпис ) (прізвище та ініціали)

**Керівник курсової роботи \_\_\_\_\_\_\_\_\_ Медведєва В.М.**

( підпис ) (прізвище та ініціали)

**АНОТАЦІЯ**

В курсовій роботі розроблено транслятор для заданої мови. До його складу входять лексичний та синтаксичний аналізатори, блок побудови польського інверсного запису (ПОЛІЗ), а також виконання ПОЛІЗ.

Метою курсової роботи є створення граматики та розробки транслятора мови програмування високого рівня. Мова високого рівня стала основним засобом розробки програмного забезпечення, тому компілятори складають вагому частину програмного забезпечення.

Оскільки програми мови високого рівня мають мало сенсу без добре розробленого інтерфейсу, тому приділяється увага не лише розв’язанню задачі, а й зручному її перегляду для користувача.

Пояснювальна записка складається з 40 сторінок, 22 рисунків, 3 таблиць та 3 додатків.

**ANNOTATION**

In the course work a compiler for the given language is designed. It consists of scanner and parser analyzers, the Polish construction inverse record (climbed) and climbed performance.

The aim of the course work is grammar and high-level programming language translator development. High level language became the main mean of software development, became compilers are important of computer software.

As high level language program have little meaning without a well-designed interface that focuses not only solving the problem but also convenient for the user to view it.

The explanatory, memorandum consists of 40 pages, 22 figures, 3 tables and 3 annexes.

Зміст

[Вступ 7](#_Toc483134824)

[1 Постановка задачі 8](#_Toc483134825)

[2 Поняття транслятора і принципи розробки 9](#_Toc483134826)

[2.1 Поняття транслятора, його структура та основні функції 9](#_Toc483134827)

[2.2 Граматика мови 10](#_Toc483134828)

[2.2.1 Поняття граматики мови 10](#_Toc483134829)

[2.2.2 Граматика 11](#_Toc483134830)

[2.2.3 Приклад програми за даною граматикою: 12](#_Toc483134831)

[3 Лексичний аналізатор 13](#_Toc483134832)

[3.1 Функції лексичного аналізатора 13](#_Toc483134833)

[3.2 Реалізація лексичного аналізатора за допомогою скінченного автомату 14](#_Toc483134834)

[3.3 Приклади роботи лексичного аналізатора 18](#_Toc483134835)

[4 Синтаксичний аналізатор 18](#_Toc483134836)

[4.1 Принцип дії синтаксичного аналізатора 18](#_Toc483134837)

[4.2 Поняття синтаксичного аналізу на базі рекурсивного спуску 18](#_Toc483134838)

4.2.1 Граматика для рекурсивного спуску…………………..19

[4.3 Результат роботи синтаксичного аналізатора 20](#_Toc483134839)

[5 Генерація ПОЛІЗ та його виконання 21](#_Toc483134840)

[5.1 Генерація ПОЛІЗ 21](#_Toc483134841)

[5.1.1 Оператор циклу 22](#_Toc483134842)

[5.1.2 Оператор умовного переходу 24](#_Toc483134843)

[5.1.3 Тернарний оператор 25](#_Toc483134844)

[5.1.4 Оператор вводу та виводу 25](#_Toc483134845)

[5.1.5 Особливості генерації ПОЛІЗ 26](#_Toc483134846)

[5.1.6 Результат побудови ПОЛІЗ 26](#_Toc483134847)

[5.2 Виконання ПОЛІЗ 27](#_Toc483134848)

[5.2.1 Правила виконання 27](#_Toc483134849)

[5.2.2 Результат виконання ПОЛІЗ 28](#_Toc483134850)

[6 Інструкція користувача по роботі з транслятором 29](#_Toc483134851)

[Висновки 32](#_Toc483134852)

[Список використаних джерел 3](#_Toc483134853)4

[Додаток 1 36](#_Toc483134854)

[Додаток 2 38](#_Toc483134855)

[Додаток 3 42](#_Toc483134856)

Вступ

Теорія і практика розробки трансляторів розвиваються вже кілька десятиліть, і за цей час були розроблені сотні трансляторів, тим не менш, цей напрямок програмної інженерії не втрачає своєї актуальності і сьогодні. Це пов'язано як з розвитком технології виробництва обчислювальних систем, так і з необхідністю вирішення все більш складних прикладних задач. Транслятори грають важливу роль в розробці програмного забезпечення, при передачі повідомлень по мережі, при організації B2B з'єднань. Тому вимоги до надійності трансляторів надзвичайно високі, так як від правильності їх роботи залежить правильність роботи систем, в які вони входять або при розробці яких вони використовувалися.

Необхідність нових трансляторів може бути обумовлена різними причинами, зокрема, функціональними обмеженнями, відсутністю локалізації, низькою ефективністю. Існує безліч прикладів того, що створення свого спеціалізованого транслятора буває актуальним і ефективним. Тому, основи теорії мов і формальних граматик, а також практичні методи розробки трансляторів лежать у фундаменті інженерної освіти.

1. Постановка задачі

Використовуючи теоретичний матеріал по курсу «Лінгвістичне забезпечення САПР», реалізувати на будь-якій мові програмування транслятор у відповідності до індивідуального завдання. Транслятор має складатися з 4-ох частин :

1) лексичний аналізатор;

2) синтаксичний аналізатор;

3) побудова польського інверсного запису (ПОЛІЗ);

4) виконання польського інверсного запису.

Мають бути наведені декілька тестових прикладів роботи транслятора. Необхідний мінімум : показати обробку помилок лексичного і синтаксичного аналізаторів, результати роботи відтрансльованої програми.

Граматика мови має такі особливості :

**Оператор циклу та умовного переходу:**

<цикл>::= do while (<логічний вираз>) <список операторів> enddo

<ум.перех>::= if <лог.вираз> then<оператор>

**Особливості арифметичного виразу:** +-\*/, унарний мінус, (), константи з фіксованою точкою

**Додаткові оператори:** Унарний мінус

**Роздільник:** ¶

**Алгоритм лексичного аналізу**: Розбір до роздільника.

**Алгоритм синтаксичного аналізу**: Висхідний розбір.

1. Поняття транслятора і принципи розробки
   1. Поняття транслятора, його структура та основні функції

Призначення транслятора мови високого рівня полягає в перетво­ренні початкового тексту програми у формат команд, котрі здатний сприймати комп'ютер. Наприклад для виконання простого оператора *Х=Y+Z* необхідно перетворити його на наступну послідовність команд:

звернення до області пам'яті, що містить Y;

звернення до області пам'яті, що містить Z

додавання змінних Y і *Z* ;

пересилка результату в область пам'яті, де знаходиться змінна X.

Таким чином, транслятор є програмою, котра здатна сприймати ланцюжок символів певного вигляду (тобто текст програми, написаний початковою мовою) і видавати інший рядок символів (програму на ма­шинній мові, так звану об’єктну або цільову програму). Особливістю транслятора проблемно-орієнтованої мови є те, що його виходом може бути текст на внутрішній мові САПР: певні структури даних, які фор­муються з початкового тексту; команди виконавчих механізмів і таке інше. Якщо цільова програма написана машинною мовою, користувач може потім запустити її для обробки деяких вхідних даних і отримання вихідних.[1]

Трансляторам властивий ряд загальних рис, що спрощує процес створення транслюючих програм. До складу будь-якого транслятора вхо­дять три основні компоненти [3] :

* + лексичний аналізатор;
  + синтаксичний аналізатор;
  + генератор кодів машинних команд.
  1. Граматика мови
     1. Поняття граматики мови

Більшість концепцій базується на теорії формальних граматик, основоположником якої є H. Хомський.

Граматика мови програмування – це формальний опис синтаксису або форми, в якій записані окремі речення програми або вся програма. Граматика не описує семантику, тобто зміст різноманітних речень. Інформація про семантику міститься в програмах генерації об'єктного коду. Неформально мову можна визначити як підмножину всіх правильних речень.

Алфавіт — це порожня скінченна множина елементів. Назвемо елементи алфавіту символами.

Будь-яка скінченна послідовність символів алфавіту **A** називається ланцюжком (рядком),

Наприклад, в алфавіті **A**={a,b,c} ланцюжки: аb, b, bacc, c та ін.

Існує порожній ланцюжок, який позначається символом ^. Також важливим є порядок символів у ланцюжку: **ab** не те саме, що **ba**.

Довжина ланцюжка **x** (записується **|x|** ) дорівнює числу символів у ланцюжку.

**|^|=0, |b|=1, |acb|=3**

Якщо **х** і **у** — ланцюжки, то **xy** є ланцюжок, отриманий шляхом дописування символів ланцюжка **у** слідом за символами ланцюжка **x**. Якщо **x=yx**, **y=zy**, тоді **xy=yxzy**.

Якщо **z=xy** — ланцюжок, тоді **x** — голова, **у** — хвіст ланцюжка **z; x** — правильна голова, якщо **у** — не порожній ланцюжок; **у** — правильний хвіст, якщо **x**— не порожній ланцюжок. [1]

Граматика, за H. Хомським — це алгоритм, що здатний визначити, чи належить даний ланцюжок до даної мови.

Породжуюча граматика — це упорядкована четвірка:

G=<VT, VN, 𝜎, P>,

де VT — скінченна множина, що називається термінальним словником;

VN — скінченна множина, що називається не термінальним словником;

𝜎 — аксіома граматики, s Є VN;

P — множина правил виведення спеціального вигляду:

P = { p: α→β }, α,β є (VTVN)\*.



Приклад подання граматики:

G=<{a,b}, {A,B}, A, {A→aBb, B→a, B→b} >. [1]

* + 1. **Граматика**

В процесі роботи над курсовою роботою була розроблена граматика, що відповідає всім поставленим вимогам та написаний приклад програми (Рисунок 2.1):

<прогр>::=prog IDN var <спис\_огол1> { <спис\_опер1> }

<спис\_огол1>::=<спис\_огол>

<спис\_огол>::=<спис\_ід1> ¶|<спис\_ід1> ¶ <спис\_огол>

<спис\_ід1>::=<спис\_ід>

<спис\_ід>::=<тип> IDN|<спис\_ід> , IDN

<спис\_опер1>::=<спис\_опер>

<спис\_опер>::=<опер1> ¶|<опер1> ¶ <спис\_опер>

<тип>::=int|double

<опер1>::=<опер>

<опер>::=<вивід>|<ввід>|<присв>|<цикл>|<ум\_опер>

<присв>::=IDN = <вираз1>

<вивід>::=cout <опер\_виводу1>

<опер\_виводу1>::=<опер\_виводу>

<опер\_виводу>::=<< IDN|<< CST|<опер\_виводу> << IDN|<опер\_виводу> << CST

<ввід>::=cin <опер\_вводу1>

<опер\_вводу1>::=<опер\_вводу>

<опер\_вводу>::=>> IDN|<опер\_вводу> >> IDN

<цикл>::=do while ( <лог\_вираз1> ) <спис\_опер1> enddo

<ум\_опер>::=if ( <лог\_вираз1> ) then <опер\_блок>|if ( <лог\_вираз1> ) then <опер>

<опер\_блок>::={ <спис\_опер1> }

<лог\_вираз1>::=<лог\_вираз>

<лог\_вираз>::=<лог\_терм1>|<лог\_вираз> or <лог\_терм1>

<лог\_терм1>::=<лог\_терм>

<лог\_терм>::=<лог\_мн1>|<лог\_терм> and <лог\_мн1>

<лог\_мн1>::=<лог\_мн>

<лог\_мн>::=<відн>|[ <лог\_вираз1> ]|not <лог\_мн>

<відн>::=<вираз1> < <вираз1>|<вираз1> > <вираз1>|<вираз1> <= <вираз1>|<вираз1> >= <вираз1>|<вираз1> == <вираз1>|<вираз1> != <вираз1>

<вираз1>::=<вираз>

<вираз2>::=<вираз1>

<вираз>::=<терм1>|<вираз> + <терм1>|<вираз> - <терм1>

<терм1>::=<терм>

<терм>::=<мн1>|<терм> \* <мн1>|<терм> / <мн1>

<мн1>::=<мн>

<мн>::=IDN|CST|( <вираз2> )

* + 1. Приклад програми за даною граматикою:

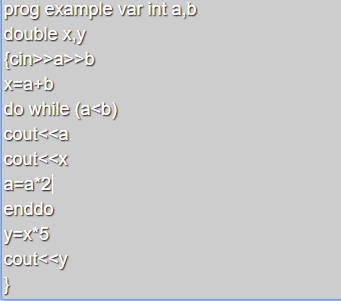


Рисунок 2.1 – Приклад програми

# Лексичний аналізатор

* 1. Функції лексичного аналізатора

**Основна задача лексичного аналізатора (ЛА)** - розбір вхідного рядка і символів на лексичні одиниці. **Лексична одиниця (ЛО)** - це підрядок вхідного рядка. Вона може містити лише термінальні символи (елементи) і не може містити інших лексем. Для синтаксичного аналізу лексема є найменшою одиницею мови, а з термінальними символами працює виключно лексичний аналізатор - сканер.

Лексеми описуються **двома ознаками:** клас і номер елемента даного класу. Розпізнають 3 **класи лексем:** термінальні (TRM), включають символи,службові слова та роздільники, ідентифікатори (IDN) та константи (CON).

**Задачі сканера**

* + Виокремлення лексем в програмі.
  + Побудова таблиць.

**Вхідні дані**

* + Таблиця лексем мови (ключові слова, оператори, роздільники).
  + Початкова програма.

**Вихідні дані**

* + Таблиця констант.
  + Таблиця ідентифікаторів.
  + Вихідна таблиця лексем.

Для кожної мови будується одна таблиця лексем. Вона не залежить від початкової програми, і будується за граматикою мови за такими принципами:

1. таблиця лексем повинна містити усі термінали, що зустрічаються у правилах граматики;
2. в таблиці лексем кожний запис відповідає тільки одному терміналу;
3. білі роздільники не входять в таблицю лексем;
4. окремими елементами таблиці лексем є ідентифікатори та констан­ти, складові цих класів лексем не входять до таблиці лексем.[1]

У ході виконання курсової роботи було побудовано таблицю лексем (Таблиця 3.1) для розробленої раніше мови, наведеної в прикладі (Рисунок 2.1), лексеми закодовані послідовними числами.

Таблиця 3.1

|  |  |  |  |
| --- | --- | --- | --- |
| Код | Лексема | Код | Лексема |
| 1 | prog | 19 | / |
| 2 | var | 20 | ( |
| 3 | int | 21 | ) |
| 4 | double | 22 | = |
| 5 | cout | 23 | << |
| 6 | cin | 24 | >> |
| 7 | if | 25 | < |
| 8 | then | 26 | > |
| 9 | do | 27 | <= |
| 10 | while | 28 | >= |
| 11 | enddo | 29 | == |
| 12 | { | 30 | != |
| 13 | } | 31 | and |
| 14 | , | 32 | or |
| 15 | ¶ | 33 | not |
| 16 | + | 34 | idn |
| 17 | - | 35 | const |
| 18 | \* | 36 | [ |
|  |  | 37 | ] |

* 1. Реалізація лексичного аналізатора. Перегляд до роздільника

Застосовуючи перегляд до роздільника, реалізація сканера буде наступною: коли зчитується наступний символ вхідної програми, з’ясовують, чи є він роздільником. Між двома роздільниками знаходиться лексична одиниця [3]. Спочатку всі лексичні одиниці порівнюються з лексемами типу TRM, при збіжності вони заносяться до вихідної таблиці лексем [7]. При незбіжності класифікуються як можливий ідентифікатор або константа. Мітка є ідентичною до ідентифікатора, адже розцінюється як ідентифікатор окремого типу (label) і записується до таблиці ідентифікаторів. Якщо лексична одиниця не підходить до жодної з цих категорій, видається сигнал про помилку [3].

Для того, щоб усі помилки супроводжувалися номером рядка, у якому вони зустрілися, було збережено для кожної лексеми номер рядка.

* 1. Приклади роботи лексичного аналізатора

Після лексичного аналізу введеного коду програми (Рисунок 2.1) можна знайти таблиці лексем, ідентифікаторів та констант (Рисунок 3.2) . Якщо програма виконується з лексичною помилкою, буде виведена відповідна інформація (Рисунок 3.3).

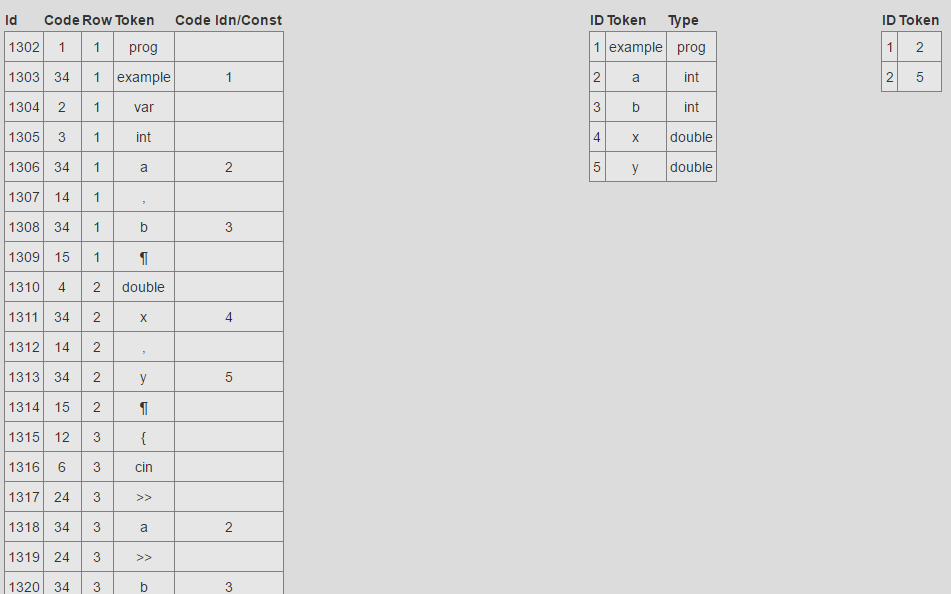


Рисунок 3.2 – Приклад роботи лексичного аналізатора без помилок в програмі



Рисунок 3.3 – Приклад виводу помилки при роботі лексичного аналізатора з помилкою в програмі

# Синтаксичний аналізатор

* 1. Принцип дії синтаксичного аналізатора

Мета синтаксичного розбору - визначення у вхідному тексті мови конструкцій, що описуються граматикою.

На вході синтаксичний аналізатор (СА) отримує побудовані лексичним аналізатором вихідні таблиці лексем. В процесі своєї роботи СА будує дерево виводу для отриманої послідовності лексем, часто цей процес здійснюється неявно, але на ньому базуються основні методи синтаксичного розбору. На виході СА повинен винести вердикт, чи є отримана послідовність лексем правильним реченням даної мови.

Розрізняють дві категорії алгоритмів синтаксичного розбору: низхідний (зверху-вниз) та висхідний (знизу-вверх). Ці терміни відповідають способу побудови дерев виводу. При низхідному розборі дерево будується від кореня (початкового символу, тобто аксіоми) до кінцевих вузлів (термінальних символів). Метод висхідного розбору полягає в тому, що заданий термінальний ланцюжок намагаються привести до аксіоми граматики.[1]

* 1. Поняття синтаксичного аналізу на базі рекурсивного спуску

Перетворена відповідно до правил граматика може використовуватися в низхідному методі граматичного розбору без повернень, котрий називається методом рекурсивного спуску.

Процесор граматичного розбору, що базується на цьому методі, складається з окремих процедур для кожного нетерміналу, визначеного граматикою. Кожна така процедура відображає праву частину відповідного правила і намагається знайти у вхідному потоці підрядок, що починається з поточної лексеми і може бути інтерпретований як нетермінальний символ, пов’язаний з даною процедурою. В процесі своєї роботи вона може викликати інші подібні процедури або навіть саму себе (рекурсивно) для пошуку інших нетерміналів, якщо вони зустрічаються в правій частині правила.

Якщо ця процедура знаходить (тобто визначає) відповідний їй нетермінал, вона завершує свою роботу, передає керуючій програмі ознаку успішного завершення і встановлює вказівник поточної лексеми на першу лексему після розпізнаного підрядка. Якщо ж процедурі не вдається знайти підрядок, котрий може бути інтерпретований як відповідний нетермінальний символ, вона завершується з ознакою невдачі і викликає процедуру видачі діагностичного повідомлення та процедуру відновлення.

Процедура видачі діагностичних повідомлень повинна повідомляти про місце помилки (наприклад, рядок, символ) і можливо, про зміст помилки.

* + 1. **Граматика для рекурсивного спуску**

<програма>::=prog IDN var <список оголошень>{<список операторів>}

<список оголошень>::=<тип><список ідентифікаторів>{;<тип><список ідентифікаторів>}

<тип>::=int|double

<список ідентифікаторів>::=IDN{,IDN}

<список операторів>::=<оператор>¶{<оператор>¶}

<оператор>::=<ввід>|<вивід>|IDN=<E>|if(<відношення>){<список операторів>}|do while (<відношення>)<список операторів>¶enddo

<ввід>::=cin>>IDN(>>IDN)

<вивід>::cout<<IDN|cout<<CST(<<IDN|<<CST)

<Е>::=<T>{+<T>|-<T>}

<Т>::=<C>{\*<C>|/<C>}

<C>::=CON|IDN|(<E>)|-<O>

<відношення>::=<E>(<|>|==|<=|>=|!=)<E>

* 1. Результат роботи синтаксичного аналізатора

В результаті роботи синтаксичного аналізатора для заданої програми (Рисунок 2.1) виводиться повідомлення про успішне завершення синтаксичного аналізу. При виявленні помилки в програмі, виводиться відповідне повідомлення (Рисунок 4.3)



Рисунок 4.2 – Результат роботи програми, яка не містить синтаксичних помилок



Рисунок 4.3 – Приклад виводу помилки при роботі програми, яка містить синтаксичні помилки

# Генерація ПОЛІЗ та його виконання

* 1. Генерація ПОЛІЗ

Для генерації об’єктного коду вихідної програми її перетворюють в деяку проміжну форму більш зручну для наступної обробки. В більшості внутрішніх форм представленні операції розміщуються в тому порядку, в якому вони мають виконуватися. Це полегшує наступну генерацію об’єктного коду. Однією із внутрішніх форм представлення є польський інверсний запис – ПОЛІЗ.[2]

Щоб перетворити вихідну програму в ПОЛІЗ скористуємося алгоритмом Дейкстри. Цей алгоритм полягає в наступному:

* ідентифікатори і константи проходять від входу прямо на вихід;
* операції попадають на вихід тільки через стек. Якщо пріоритет операції, що знаходиться в стеку більший або рівний пріоритету поточної операції, то операція із стека передається на вихід і п.2 повторюється. В противному випадку поточна операція заноситься в стек;
* якщо стек пустий, то поточна операція заноситься в стек;
* якщо вхідний ланцюжок пустий, то всі операції виштовхуються на вихід;[4]

Для наведеної граматику було побудовано таблицю пріоритетів (Таблиця 5.1).

Таблиця 5.1. Пріоритети операцій

|  |  |
| --- | --- |
| Операція або оператор | Пріоритет |
| {, if, do, ( | 0 |
| }, ) ,while, cout, cin, >>, <<, ¶ | 1 |
| then, enddo | 2 |
| or, = | 3 |
| and | 4 |
| not | 5 |
| >,<,>=,<=,==,!= | 6 |
| +,- | 7 |
| \*,/,@ |  |

* + 1. Оператор циклу

do while (a<b)

a = a\*b

enddo

Для наведеного вище оператора циклу було побудовано блок-схему (рисунок 5.1).

**ПОЛІЗ** буде мати наступний вигляд

mi+1 УПЛ a a b \* =mi БП

while; ¶ enddo

Наведений ПОЛІЗ можна отримати за умови наступного навантаження на службові слова:

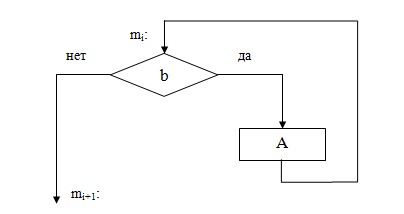


Рисунок 5.1 – Блок-схема оператору цикла

* оператор **do**  має пріоритет 1 та генерує 2 робочі мітки **mi, mi+1** та записується в стек
* оператор **while** генерує мітку **mi:**та записує її позицію
* символ **¶** діє по своєму пріоритету, але якщо ознака циклу **true** то виягує зі стека мітку **mi+1**  та генерує **mi+1 УПЛ**
* у інших випадках ¶являється закриваючою дужкою для попереднього виразу, має пріоритет 1, виштовхує з стеку все до **do** таким чином генерує поліз для операторів в циклі
* оператор **enddo** очищує стек до **while**, генерує на вихід **mi БП mi+1 :** , видаляє запис **while** із стеку та записує позицію **mi+1**
  + 1. Оператор умовного переходу

if (<лог. вираз>) {<сп. операторів>}

Для наведеного вище оператора умовного переходу було побудовано блок-схему (Рисунок 5.2).

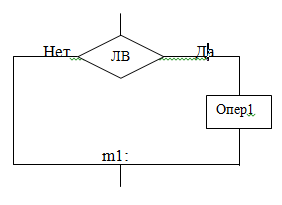


Рисунок 5.2 – Блок-схема оператора умовного переходу

**ПОЛІЗ** буде мати наступний вигляд:

<лог вираз> miУПЛ <спис.опер> mi:

}

{

Наведений ПОЛІЗ можна отримати за умови наступного навантаження на службові слова:

* оператор **if** грає роль відкриваючої дужки та має пріоритет 0;
* по **{** генерується мітка **mi** та **УПЛ**;
* }записує адресу мітки **mi**.
  + 1. Оператор вводу та виводу

cout<<IDN|cout<<CST(<<IDN|<<CST)

cin>>IDN(>>IDN)

По **cout / cin** в стек заноситься cout / cin. По **<</>>**  в ПОЛІЗ генерується  
<< / >>. По **¶** зі стека виштовхується cout / cin.

* + 1. Особливості генерації ПОЛІЗ

Деякі особливості виникають з відкриваючою і закриваючою дужкою і оператором присвоєння. Відкриваюча дужка записується у стек але ніколи не заноситься в ПОЛІЗ, а закриваюча дужка взагалі нікуди не записується, лише переноситься зі стеку в ПОЛІЗ все до закриваючої дужки включно і видаляє зі стеку закриваючу дужку.

Оператор присвоєння застосовується до двох верхніх комірок стеку, але зі стеку видаляються обидві комірки, а значення першої комірки заноситься по відповідному адресу ідентифікатора другої комірки стеку.

Для міток створюється окрема таблиця міток в якій для кожної мітки відповідає деяка комірка ПОЛІЗ.

* + 1. Результат побудови ПОЛІЗ

Після проходження лексичного та синтаксичного аналізу консолі можна переглянути ланцюжок ПОЛІЗ (Рисунок 5.4).

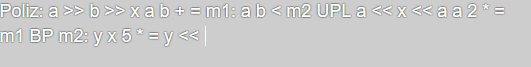


Рисунок 5.4 – Динаміка будування та ПОЛІЗ

* 1. Виконання ПОЛІЗ
     1. Правила виконання

ПОЛІЗ виконується з використанням стека відповідно до наступних правил:

1. якщо поточний символ є ідентифікатором, міткою або константою, то він заноситься у стек і здійснюється перехід до наступного символу;
2. якщо поточний символ є арифметичною дією або операцією порівняння (n-місною), то ця дія виконується над значеннями n елементів стеку починаючи із вершини. Результат цієї операції заноситься в стек замість n елементів стеку починаючи із вершини. Здійснюється перехід до наступного символу;
3. якщо поточний символ дія ( n-місна) і вона не є ні арифметичною, ні операцією порівняння, то ця дія виконується над n елементами стеку починаючи із вершини і видаляє їх із стека. Здійснюється перехід до наступного символу.[5]

Особливості виконання деякий команд:

* Оператори вводу і виводу виконуються над попереднім ідентифікатором;
* Оператор присвоєння виконується наступним чином: значення першого аргументу змінюється на значення другого;
* Результатом виконання операторів порівняння є або TRUE, або FALSE;
* Перехід на мітку при виконанні операцій умовного і безумовного переходів здійснюється наступним чином : курсор в ПОЛІЗ переводиться на те місце, адреса якого записана в таблиці міток.
  + 1. Результат виконання ПОЛІЗ

Після генерації ПОЛІЗ транслятор виконує програму та виводить результат на консоль (Рисунок 5.5).

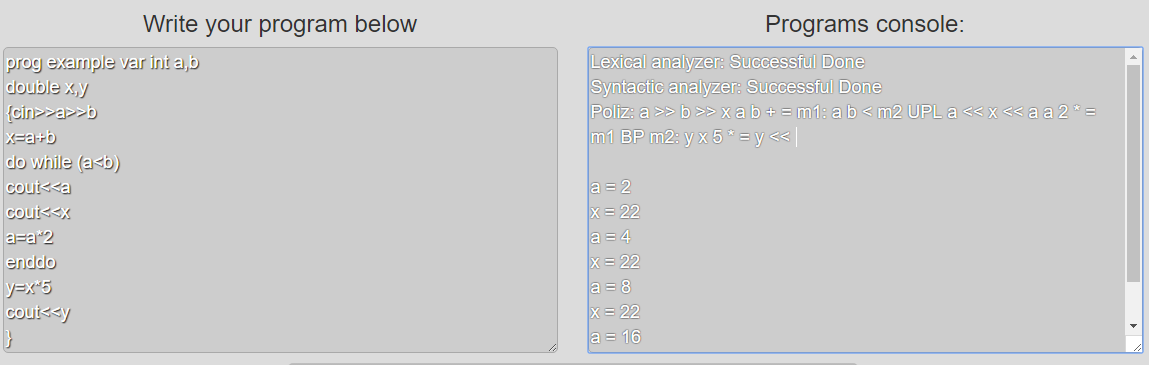


Рисунок 5.5 – Результат виконання ПОЛІЗ

# Інструкція користувача по роботі з транслятором

Коли користувач завантажує сторінку, він бачить граматику для мови (Рисунок 6.1), яке можна переглядати але не змінювати

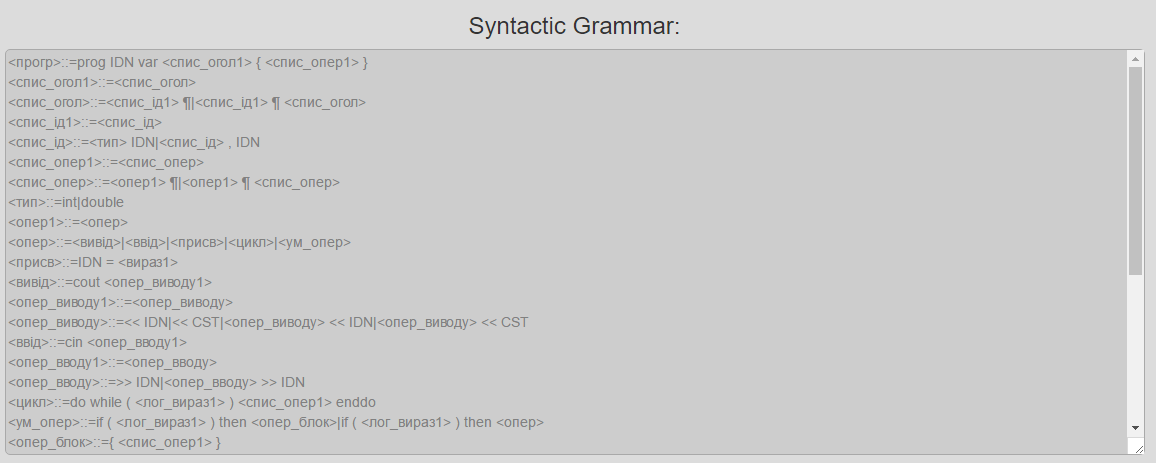


Рисунок 6.1 – Головне вікно програми

Для вводу коду програми (Рисунок 6.2), можна потрібно скористатися спеціальним полем для введення коду.

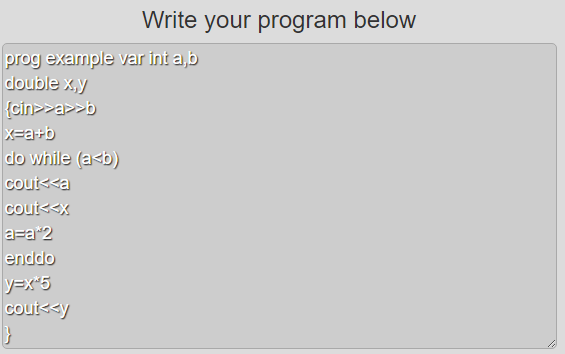


Рисунок 6.2 – Ввід коду

Після введення коду, потрібно натиснути на кнопку «RUN» (Рисунок 6.3).

Яка підсвічується при наведенні



Рисунок 6.3 – Кнопка виконання програми

Якщо на стадії лексичного аналізу виявлено лексичні помилки, буде видана відповідна помилка на консоль (Рисунок 6.4).

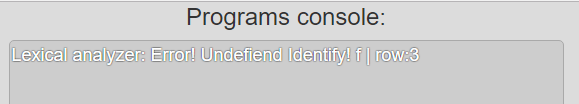


Рисунок 6.4 – Помилка на стадії лексичного аналізу

Якщо лексичний аналіз завершився без помилок, автоматично починає роботу синтаксичний аналіз. Якщо на стадії синтаксичного аналізу немає помилок, видається відповідне повідомлення(Рисунок 6.5). Якщо на стадії синтаксичного аналізу виявлено помилку, видається відповідний надпис (Рисунок 6.7).

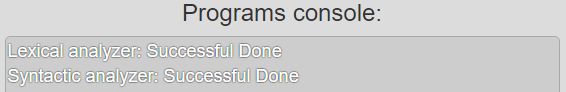


Рисунок 6.5 – Немає помилок на стадії синтаксичного аналізу

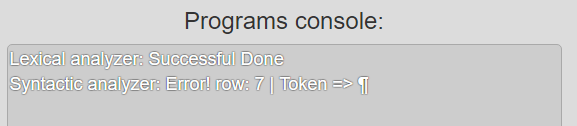


Рисунок 6.6 – Помилка на стадії синтаксичного аналізу

Після проходження лексичного та синтаксичного аналізу в консолі можна побачити згенерований ПОЛІЗ (Рисунок 6.7)

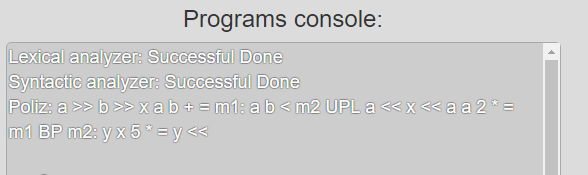


Рисунок 6.7 – Генерація ПОЛІЗ

Якщо в коді програми є оператори введення, з’явиться відповідне вікно введення значення змінної (Рисунок 6.8). Якщо в програмі є оператори виводу, то виконаний ПОЛІЗ буде відображений на головній вкладці (Рисунок 6.9). Якщо змінна не ініціалізована буде виведена відповідна помилка (Рисунок 6.10).

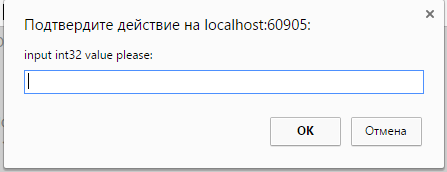


Рисунок 6.8 – Введення значення змінної

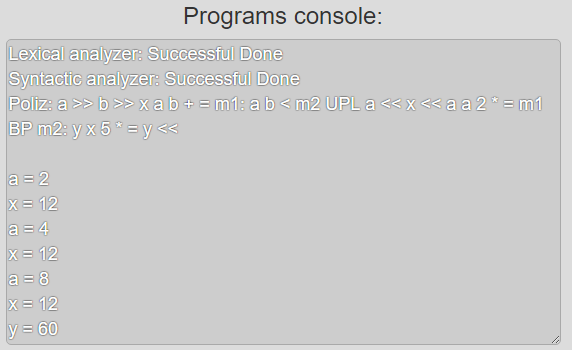


Рисунок 6.9 – Виконаний ПОЛІЗ

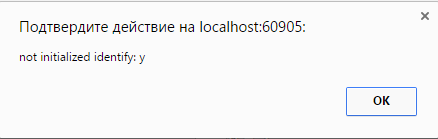


Рисунок 6.10 – Помилка при виконанні ПОЛІЗ

Також, після виконання програми користувач може переглянути необхідні таблички просто прокручуючи сторінку вверх вниз (Рисунок 6.12).

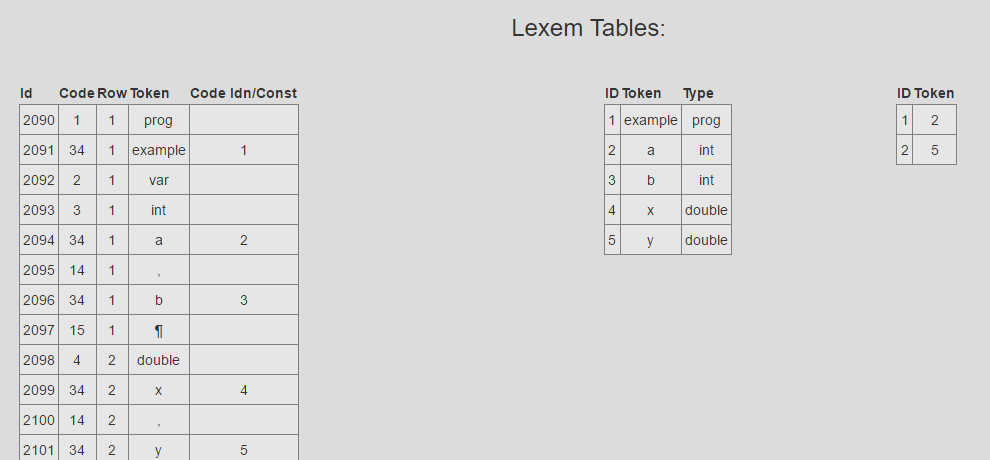


Рисунок 6.12 – Перегляд таблиць

Висновки

Під час виконання курсової роботи було реалізовано основі блоки побудови компіляторів. В результаті створено граматику спрощеної мови програмування. На базі граматики створено лексичний аналізатор, що реалізує скінченний автомат. На основі граматики побудовано рекурсивний спуск, відповідно до якого реалізовано синтаксичний аналізатор. Також розроблено блок побудови ПОЛІЗу за алгоритмом Дейкстри, який інтерпретується в окремому програмному модулі. Всі модулі були об’єднані в єдиний програмний продукт, що являє собою транслятор. Транслятор розроблено мовою програмування C# в середовищі Visual Studio 2015.

Створений транслятор не може конкурувати з уже існуючими мовами програмування високого рівня, проте він здатний виконувати найпростіші арифметичні обчислення, виконувати цикли та оператори умовного переходу, а також введення/виведення даних. Цей компілятор є моделлю компіляторів мов високого рівня.

Список використаних джерел

1. Медведєва В.М. Основи побудови трансляторів. Навчальний посібник../ Медведєва В.М., Третяк В.А. -К.:ІЗМН, 2004.-63 с.
2. Медведєва В.М. Основи побудови трансляторів. Навчальний посібник../ Сліпченко В.Г., Медведєва В.М. -К.:ІЗМН, 2004.-85 с.
3. Карпов Ю.Г.«Теория и технология программирования. Основы построения трансляторов». / Карпов Ю.Г -М.;Изд-во «БХВ-Петербург»,2005. -372с.
4. А.Ахо «Теория синтаксического анализа, перевода и компиляции», том1. / А.Ахо, Дж.Ульман -М.; Изд-во «МИР», 1978. -289 с.
5. А.Ахо «Теория синтаксического анализа, перевода и компиляции», том2./ А.Ахо, Дж.Ульман -М.; Изд-во «МИР», 1978. -151 с.

Додаток 1

Розробка транслятора загального призначення

Специфікація

УКР.НТУУ”КПІ”.ТР-4166\_3К

Листів 1

Таблиця з описом документів

|  |  |  |
| --- | --- | --- |
| Позначення | Найменування | Примітки |
| Документація | | |
| УКР.НТУУ”КПІ”\_ТЕФ\_АПEПС\_ТР-4166\_6КР\_81-1 | Пояснювальна записка |  |
| Вхідні програми | | |
| УКР.НТУУ”КПІ”\_ТЕФ\_АПEПС\_ТР-4166\_6КР\_12-1 | Текст програми «Лексичний аналізатор» |  |
| УКР.НТУУ”КПІ”\_ТЕФ\_АПEПС\_ТР-4166\_6КР\_13-1 | Опис лексичного аналізатора |  |

Додаток 2

Розробка транслятора загального призначення

Текст програми

УКР.НТУУ”КПІ”\_ТЕФ\_АПEПС\_ТР-4166\_3К 12-1

Листів 3

private bool LexemParser(string codeRow, int rowNumber, out string analyzeResult)

{

string lastType = ""; //зміння для зберігання останнього об'явленого типу даних

analyzeResult = "";

string[] Tokens = codeRow.Split(' ');//ділимо по пробілу

foreach(string token in Tokens)

{

if (token!="")

{

var obj = \_inputLexem.Find(o => o.Token == token);

if (obj!=null)

{

\_outTables.\_OutLexem.Add(new OutLexem(\_outTables.\_OutLexem.Count() + 1,

rowNumber, token, obj.Id, null));

if (obj.Id==3 || obj.Id==4 || obj.Id==1)

{

lastType = obj.Token;

}

}

else if ((token[0]>=65 && token[0]<=90) || ((token[0]>=97 && token[0]<=122) ||

(token[0]==95))) // A-Z..a-z..\_

{

if (IsIdentify(token))

{

if (\_outTables.\_OutLexem.FindIndex(o => o.Token == "{") == -1)//якщо ми ще в розділі оголошення змінних

{

if (\_outTables.\_IdentifyTable.FindIndex(i=>i.Token==token)!=-1)//якщо таблиця ідентифікаторів вже містить даний токен

{

analyzeResult += "Error! Duplicate identify! " + token + " | row:"

+ rowNumber;

}

else

{

\_outTables.\_IdentifyTable.Add(new

Identify(\_outTables.\_IdentifyTable.Count() + 1, token, lastType));

\_outTables.\_OutLexem.Add(new OutLexem(\_outTables.\_OutLexem.Count() + 1,

rowNumber, token, 34, \_outTables.\_IdentifyTable.Count()));

}

}

else

{

int index = \_outTables.\_IdentifyTable.FindIndex(o => o.Token ==

token);//індекс ідентифікатора в таблиці ідентифікаторів

if (index==-1)

{

analyzeResult+= "Error! Undefiend Identify! " + token + " | row:" +

rowNumber;

return false;

}

else

{

\_outTables.\_OutLexem.Add(new OutLexem(\_outTables.\_OutLexem.Count() +

1, rowNumber, token, 34, index + 1));

}

}

}

}

else if (IsConst(token))

{

int index = \_outTables.\_ConstTable.FindIndex(o => o.Token == token);//індекс константи

if (index==-1)

{

\_outTables.\_ConstTable.Add(new Const(\_outTables.\_ConstTable.Count() + 1, token));

\_outTables.\_OutLexem.Add(new OutLexem(\_outTables.\_OutLexem.Count() + 1,

rowNumber, token, 35, \_outTables.\_ConstTable.Count()));

}

else

{

\_outTables.\_OutLexem.Add(new OutLexem(\_outTables.\_OutLexem.Count() + 1,

rowNumber, token, 35, index+1));

}

}

else

{

analyzeResult += "Error! Undefiend token " + token + " |row: " + rowNumber;

}

}

}

return true;

}

Додаток 3

Розробка транслятора загального призначення

Опис програми

УКР.НТУУ”КПІ”.ТР-4166\_3К 13-1

Листів 2

Опис синтаксичного аналізатора

Для реалізації лексичного аналізатора за допомогою скінченного автомату, для кожного стану було створено

методи:

* private string InsertWhiteSpace(string row)*–* метод, для додавання пробілів між такими лексемами як +, - ,\*, /, (, ), {, }, <<, >>, <, > =, ==
* public void DoAnalyze(out string analyzeResult)*–* метод що розбиває вхідний текст на рядки та виділяє лексеми з кожного рядка
* private bool IsIdentify(string token)*–* метод, для перевірки на ідентифікатор
* private bool IsConst(string token) *–* метод, перевірки на константу

змінні:

* string \_sourceText; - вхідний текст
* List<Lexem> \_inputLexem;– змінна для зберігання вхідних лексем
* OutTables \_outTables; *- змінна для зберігання вихідних лексем*