Міністерство освіти і науки України.

Національний університет “Львівська політехніка”

ІКТА

Кафедра ЕОМ

Описание: Описание: Описание: Gerb

**Пояснювальна записка**

*до курсової роботи*

*з предмету:“Системне програмування”*

*на тему:*

**“Розробка системних програмних модулів та компонент систем програмування”**

*Індивідуальне завдання “Розробка транслятора з вхідної мови програмування”*

Варіант №23

**Анотація**

В курсовій роботі розроблено транслятор з вхідної мови програмування,відповідно до варіанту,в асемблерний файл,після чого реалізована його компіляція і створення виконавчого файлу. Дана програма здійснює перевірку вхідного файлу на помилки (синтаксичні, семантичні, лексичні). Було реалізовано висхідний тип, методом зсув - згортка.

А також транслятор був протестований на виявлення помилок і коректність роботи.

**Зміст**

Завдання на курсову роботу………………………………………………………….4

Вступ……………………………………………………………………………………5

1. Огляд методів та способів проектування трансляторів…………………………….6
2. Формальний опис вхідної мови програмування…………………………………….8
   1. *Деталізований опис вхідної мови в термінах розширеної нотації*

*Бекуса-Наура……………………………………………………………………………….8*

* 1. *Опис термінальних символів та ключових слів……………………………………..9*

1. Розробка транслятора вхідної мови програмування………………………………..11
   1. *Вибір технології програмування………………………………………………………11*
   2. *Проектування таблиць транслятора та вибір структур даних……………..12*
   3. *Розробка лексичного аналізатора…………………………………………………...16*
      1. *Розробка граф-схеми алгоритму…………………………………………………..16*
      2. *Опис програми реалізації лексичного аналізатора…………………………….18*
   4. *Розробка синтаксичного та семантичного аналізатора………………………19*
      1. *Розробка граф-схеми алгоритму…………………………………………………..21*
      2. *Опис програми реалізації синтаксичного та семантичного аналізатора.22*
   5. *Розробка генератора коду……………………………………………………………...23*
      1. *Розробка граф-схеми алгоритму……………………………………………………23*
      2. *Опис програми реалізації генератора коду……………………………………..25*
2. Опис інтерфейсу та інструкції користувача………………………………………….27
3. Відлагодження та тестування програми……………………………………………..29
   1. *Виявлення лексичних помилок……………………………………………………………29*
   2. *Виявлення синтаксичних помилок………………………………………………………29*
   3. *Виявлення семантичних помилок………………………………………………………..29*
   4. *Загальна перевірка коректності роботи транслятора…………………………..29*

Висновки……………………………………………………………………………31

Список літератури………………………………………………………………..32

Додатки …………………………………………………………………………….33

*А. Лістинг програм………………………………………………………………………33*

**Завдання на курсову роботу**

Темою даного курсового проекту є розробка системних програмних модулів та компонент систем програмування.

Варіант №22

Потрібно розробити транслятор вхідної мови програмування.

Опис вхідної мови:

Блок тіла програми : MAINPROGRAM DATA…; START – END

Оператор вводу :GET

Оператор виводу : PUT

Оператор присвоєння : :=

Оператор:FOR – DOWNTO - DO (Паскаль)

Регістр ключових слів : Up

Регістр ідентифікаторів : Up \_

Арифметичні операції : +; -; \*; DIV; MOD

Операції порівняння : ==; !=; LT; GT

Логічні операції : !!; AND; OR

Типи даних : INTEGER, BOOLEAN, рядкова константа

Коментар : %% ... %%

**ВСТУП**

Системне програмування - це вид [програмування](http://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F), який полягає в роботі з [системним програмним забезпеченням](http://uk.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%BD%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%B5_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F).

Знати мову програмування і мати комп'ютер — не достатньо, щоб реалізувати програму. Потрібно мати транслятор мови і, бажано, бібліотеки стандартних програм та засоби отримання підказок від комп'ютера, що робити у випадку допущення помилок. Сукупність таких засобів утворює систему програмування деякою мовою. Найважливішим елементом у такій системі є транслятор.

Транслятор — це спеціальна програма, яку використовують для перекладу програм користувача, написаних мовою програмування високого рівня, у так звані машинні коди, зрозумілі процесору. Транслятори створюють для кожної мови програмування. Отже, транслятори перекладають команди користувача в набір команд процесора. Це дає змогу використовувати програми, створені мовою низького рівня, на різних типах машин. Іншими словами, транслятор — це перекладач. Розрізняють два типи трансляторів:компілятори, інтерпретатори.

Компілятор - це програма, призначена для перекладу в машинні коди програми, що написана мовою високого рівня. Процес такого перекладання називається компіляцією. Кінцевим результатом роботи компілятора є програма в машинних кодах, яка потім виконується ЕОМ. Існує інший спосіб поєднання процесів трансляції та виконання програм. Він називається інтерпретацією.  
Інтерпретатор - це програма, що призначена для трансляції та виконання вихідної програми по командах. Такий процес називається інтерпретацією.

У процесі трансляції відбувається перевірка програми на відповідність до правил її написання. Якщо в програмі знайдені помилки, транслятор виводить повідомлення про них на екран монітора. Судячи з цього зручніше було б подавати текст вхідної мови у вигляді послідовності лексем. У випадку відсутності помилок повинен генеруватися код.

**1. ОГЛЯД МЕТОДІВ ТА СПОСОБІВ ПРОЕКТУВАННЯ ТРАНСЛЯТОРІВ**

Створення компіляторів відбувається в певних конкретних умовах: для різних мов, для різних цільових платформ, з різними вимогами для створення компіляторів. Є такі методи створення компіляторів:

1. Прямий метод - цільовою мовою і мовою реалізації є асемблер.

2. Метод розкрути - саме цей метод і використовується у даній курсовій роботі, тобто вибирається інструмент (в даній курсовій це мова асемблер), для якого вже існує компілятор.

3.Використання крос-трансляторів.

4.З використанням віртуальних машин – дає спосіб отримати переносимо програму.

5. Компіляція на ходу.

В даній курсовій роботі згідно із завданням для парних варіантів необхідно реалізувати висхідний метод граматичного розбору. Детермiнованi висхiднi розпiзнавачi, так само як i спаднi, можуть бути побудованi не для всякої КВ-граматики, а тiльки для визначених пiдкласiв таких граматик. Найбiльш широким пiдкласом КВ-граматик є LR(k)–граматики. Цi граматики забезпечують розпiзнавання ланцюжка при переглядi злiва направо, про що свiдчить буква L (/I>Left) у назвi граматики, i дозволяють виконати правобiчне згортання, про що свiдчить буква R (Right) у назвi. Параметр k свiдчить про те, що для визначення правила граматики, яке потрiбно застосувати для згортання ланцюжка, треба переглянути не бiльше нiж k ще не прочитаних символiв вхiдного ланцюжка.

У загальному випадку алгоритми побудови розпiзнавачiв для LR(k)–граматик виявляються досить складними i трудомiсткими, тому на практицi найчастiше використовують пiдкласи LR(k)–граматик: LR(0) або SLR(1) – простi (Simple), LR(1)–граматики, що дозволяють вiдносно просто виконувати побудову висхiдних розпiзнавачiв. При цьому для кожного пiдкласу LR(k)–граматик використовується свiй алгоритм побудови.

У загальному випадку процедуру побудови висхiдного розпiзнавача за заданою граматикоюможна описати в такий спосiб:

1. Визначити для даної граматики функцiї ВПЕРВ i ВПIСЛЯ.
2. Побудувати детермiновану таблицю переходiв, що має по одному стовпцю для кожного граматичного символу i по одному рядку для кожного граматичного входження i маркера дна.
3. Якщо таблиця, побудована на кроцi 2, виходить недетермiнованою (має бiльше одного стану), то потрiбно перетворити цю таблицю в детермiновану, розглядаючи її як недетермiновану таблицю переходiв кiнцевого автомата з початковим станом *h*0.
4. Стани, отриманi на кроцi 3 (крiм стану, що вiдповiдає порожнiй множинi), варто використовувати як магазиннi символи. Отримана таблиця переходiв може мiстити переходи в порожню множину. Такi елементи варто розумiти як забороненi i розглядати переходи в них як помилки.
5. Керуючу таблицю заповнюють рядок за рядком вiдповiдно до множини граматичних входжень, що позначають рядки.

Транслятор — це програма, що призначена для перекладу тексту програми з однієї мови програмування на іншу. Процес перекладання називається трансляцією. У процесі трансляції програми інтерпретатором відбувається перевірка програми на відповідність правилам її написання. Якщо в програмі знайдені помилки, транслятор виводить повідомлення про них на екран монітора.

**2. ФОРМАЛЬНИЙ ОПИС ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ**

*2.1. Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура.*

<program> ::= MAINPROGRAM DATA <VAR\_blok>START <code\_blok> END .

<VAR\_blok> ::= <declarations> [{;<declarations>} ] .

<declarations>::= INTEGER <declaration\_i> [{,<declaration\_i>} ] | BOOLEAN <declaration\_b> [{,<declaration\_b>} ].

<declaration\_i> ::= <ident> [:= <const\_i> ] .

<declaration\_b> ::= <ident> [:= <const\_b> ] .

<ident> ::= \_<h\_letter>[<h\_letter>|<number>].

<l\_or\_n> ::= <letter>|<number> .

<letter> ::= <l\_leter>|<h\_leter>

<h\_letter>::=A|B|C|D|E|F|G|H|I|J|K|L|N|M|O|P|Q|R|S|T|U|V|W|X|Y|Z.

<l\_leter>::= a|b|c|d|e|f|g|h|i|j|k|l|n|m|o|p|q|r|s|t|u|v|w|x|y|z.

<number> ::= 0|1|2|3|4|5|6|7|8|9 .

<const\_i> ::= [-]<number>[{number}] .

<const\_b>::=TRUE|FALSE.

<code\_blok> ::= <statement> [{<statement>}] .

<statement> ::= <equation>|<cycle>|<Input>|<Output> .

<equation> ::= <ident>:=<expression\_i>|<expression\_b>.

<expression\_i>::=<term>[{+<term>|-<term>}] .

<term>::=<ident>|<const\_i>|<factor>.

<factor>::=[{\*<term>|DIV<term>|MOD<term>|<brackets>}].

<brackets>::=(<expression\_i>).

<expression\_b>::= <term\_b>[{== <term\_b>|!= <term\_b> | LT <term\_b> | GT <term\_b>}].

<term\_b>::=<ident>|<const\_b>|<factor\_b>.

<factor\_b>::=<term\_b>[{OR<term\_b>|<and>|<not>|<brackets\_b>}].

<and>::=[{AND<term\_b>}].

<not>::=!!<factor\_b>.

<brackets\_b>::=(<expression\_b>).

<cycle> ::= FOR <equation> DOWNTO <const\_i> DO [START] <code\_blok> [END].

<Input> ::= GET(<ident>).

<Output> ::= PUT (<expression\_i>|<string>) .

<string> ::= “<l\_or\_n>[{<l\_or\_n>}]”.

*2.2. Термінальні символи та ключові слова.*

MAINPROGRAM – Початок програми

DATA – блок опису змінних

START – початок тіла програми

ENT – кінець тіла програми

GET – оператор вводу

PUT – оператор виводу

FOR – DOWNTO – DO – оператор циклу

\* – операція множення

DIV – операція ділення

MOD – остача від ділення

== – рівно

!= – не рівно

INTEGER – цілий тип даних

BOOLEAN – дійсний тип

:= - присвоєння

LT - менше

GT - більше

+ - операція додавання

- - операція віднімання

!! - інверсія

AND - кон`юнкція

OR - диз`юнкція

%%…%% – блочний коментар

“ – початок рядкової константи при операції виводу;   
” – кінець рядкової константи при операції виводу;   
– розділювач між аргументами;   
; – ознака кінця оператора;   
(– відкриваюча дужка;   
) – закриваюча дужка;   
Як термінальні символи використовуються також усі індійські цифри (0–9), латинські букви (a-z, A-Z), символи табуляції, символ переходу на нову стрічку та пробіл.

**3. РОЗРОБКА ТРАНСЛЯТОРА ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ**

*3.1. Вибір технології програмування*

Було обрано технологію об'єктно-орієнтованого програмування. Об'єктно-орієнтоване програмування (ООП) — одна з [парадигм](http://uk.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B8_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F) [програмування](http://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F), яка розглядає програму як множину «об'єктів», що взаємодіють між собою. В ній використано декілька технологій від попередніх парадигм, зокрема [успадкування](http://uk.wikipedia.org/wiki/%D0%A3%D1%81%D0%BF%D0%B0%D0%B4%D0%BA%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F%29), [модульність](http://uk.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D1%8C_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F%29), [поліморфізм](http://uk.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D1%96%D0%BC%D0%BE%D1%80%D1%84%D1%96%D0%B7%D0%BC_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F%29) та [інкапсуляцію](http://uk.wikipedia.org/wiki/%D0%86%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D1%96%D1%8F).

На відміну від традиційних поглядів, коли програму розглядали як набір [підпрограм](http://uk.wikipedia.org/wiki/%D0%9F%D1%96%D0%B4%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%B0), або як перелік [інструкцій](http://uk.wikipedia.org/wiki/%D0%86%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%86%D1%96%D1%8F) комп'ютеру, ООП програми можна вважати сукупністю об'єктів. Відповідно до парадигми об'єктно-орієнтованого програмування, кожний об'єкт здатний отримувати [повідомлення](http://uk.wikipedia.org/wiki/%D0%9E%D0%B1%D0%BC%D1%96%D0%BD_%D0%BF%D0%BE%D0%B2%D1%96%D0%B4%D0%BE%D0%BC%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F%D0%BC%D0%B8), обробляти дані, та надсилати повідомлення іншим об'єктам. Кожен об'єкт — своєрідний незалежний автомат з окремим призначенням та відповідальністю.

Об'єктно-орієнтований підхід полягає в наступному наборі основних принципів:

* Все є об'єктами.
* Всі дії та розрахунки виконуються шляхом взаємодії (обміну даними) між об'єктами, при якій один об'єкт потребує, щоб інший об'єкт виконав деяку дію. Об'єкти взаємодіють, надсилаючи і отримуючи повідомлення. Повідомлення — це запит на виконання дії, доповнений набором аргументів, які можуть знадобитися при виконанні дії.
* Кожен об'єкт має незалежну пам'ять, яка складається з інших об'єктів.
* Кожен об'єкт є представником (екземпляром, примірником) класу, який виражає загальні властивості об'єктів.
* У класі задається поведінка (функціональність) об'єкта. Таким чином усі об'єкти, які є екземплярами одного класу, можуть виконувати одні й ті ж самі дії.
* Класи організовані у єдину деревовидну структуру з загальним корінням, яка називається ієрархією [успадкування](http://uk.wikipedia.org/wiki/%D0%A3%D1%81%D0%BF%D0%B0%D0%B4%D0%BA%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F%29). Пам'ять та поведінка, зв'язані з екземплярами деякого класу, автоматично доступні будь-якому класу, розташованому нижче в ієрархічному дереві.

. NET Framework - програмна платформа компанії [Microsoft](http://translate.googleusercontent.com/translate_c?hl=uk&sl=ru&u=http://ru.wikipedia.org/wiki/Microsoft&prev=/search%3Fq%3D%25D0%259F%25D0%25BB%25D0%25B0%25D1%2582%25D1%2584%25D0%25BE%25D1%2580%25D0%25BC%25D0%25B0%2BNET%26hl%3Duk%26client%3Dfirefox-a%26hs%3D8xm%26rls%3Dorg.mozilla:uk:official%26prmd%3Divns&rurl=translate.google.com&usg=ALkJrhiFai22xg8kbmRmIaquCdqtZjeTVg) , призначена для створення звичайних програм та веб-додатків.

Головною ідеєю розробки. NET Framework було прагнення зробити платформонезалежну віртуальну машину для виконання одного і того ж коду в різних [ОС](http://translate.googleusercontent.com/translate_c?hl=uk&sl=ru&u=http://ru.wikipedia.org/wiki/%25D0%259E%25D0%25BF%25D0%25B5%25D1%2580%25D0%25B0%25D1%2586%25D0%25B8%25D0%25BE%25D0%25BD%25D0%25BD%25D0%25B0%25D1%258F_%25D1%2581%25D0%25B8%25D1%2581%25D1%2582%25D0%25B5%25D0%25BC%25D0%25B0&prev=/search%3Fq%3D%25D0%259F%25D0%25BB%25D0%25B0%25D1%2582%25D1%2584%25D0%25BE%25D1%2580%25D0%25BC%25D0%25B0%2BNET%26hl%3Duk%26client%3Dfirefox-a%26hs%3D8xm%26rls%3Dorg.mozilla:uk:official%26prmd%3Divns&rurl=translate.google.com&usg=ALkJrhj6Q20mWMxoU9MPs8mpllbw5E6Igg) без внесення змін на момент компіляції. Але з часом Microsoft обмежилася підтримкою тільки своїх операційних систем Windows. Підтримкою деяких інших платформ займаються незалежні розробники (проекти [Mono](http://translate.googleusercontent.com/translate_c?hl=uk&sl=ru&u=http://ru.wikipedia.org/wiki/Mono&prev=/search%3Fq%3D%25D0%259F%25D0%25BB%25D0%25B0%25D1%2582%25D1%2584%25D0%25BE%25D1%2580%25D0%25BC%25D0%25B0%2BNET%26hl%3Duk%26client%3Dfirefox-a%26hs%3D8xm%26rls%3Dorg.mozilla:uk:official%26prmd%3Divns&rurl=translate.google.com&usg=ALkJrhg2g2RzNA3ZZOSrpwdpw3hiO30Xyg) , [Portable.NET](http://translate.googleusercontent.com/translate_c?hl=uk&sl=ru&u=http://ru.wikipedia.org/wiki/Portable.NET&prev=/search%3Fq%3D%25D0%259F%25D0%25BB%25D0%25B0%25D1%2582%25D1%2584%25D0%25BE%25D1%2580%25D0%25BC%25D0%25B0%2BNET%26hl%3Duk%26client%3Dfirefox-a%26hs%3D8xm%26rls%3Dorg.mozilla:uk:official%26prmd%3Divns&rurl=translate.google.com&usg=ALkJrhg4rMFLF7gapPOVlT1uWF-2gVNTUw) ).

Однією з основних ідей Microsoft. NET є сумісність програмних частин, написаних на різних мовах. Наприклад, служба, написана на [C + +](http://translate.googleusercontent.com/translate_c?hl=uk&sl=ru&u=http://ru.wikipedia.org/wiki/C%252B%252B&prev=/search%3Fq%3D%25D0%259F%25D0%25BB%25D0%25B0%25D1%2582%25D1%2584%25D0%25BE%25D1%2580%25D0%25BC%25D0%25B0%2BNET%26hl%3Duk%26client%3Dfirefox-a%26hs%3D8xm%26rls%3Dorg.mozilla:uk:official%26prmd%3Divns&rurl=translate.google.com&usg=ALkJrhinALCCsjpR61tLYEcyl_4hLaLQQA) для Microsoft. NET, може звернутися до методу класу з бібліотеки, написаної на [Delphi](http://translate.googleusercontent.com/translate_c?hl=uk&sl=ru&u=http://ru.wikipedia.org/wiki/Delphi_%28%25D1%258F%25D0%25B7%25D1%258B%25D0%25BA_%25D0%25BF%25D1%2580%25D0%25BE%25D0%25B3%25D1%2580%25D0%25B0%25D0%25BC%25D0%25BC%25D0%25B8%25D1%2580%25D0%25BE%25D0%25B2%25D0%25B0%25D0%25BD%25D0%25B8%25D1%258F%29&prev=/search%3Fq%3D%25D0%259F%25D0%25BB%25D0%25B0%25D1%2582%25D1%2584%25D0%25BE%25D1%2580%25D0%25BC%25D0%25B0%2BNET%26hl%3Duk%26client%3Dfirefox-a%26hs%3D8xm%26rls%3Dorg.mozilla:uk:official%26prmd%3Divns&rurl=translate.google.com&usg=ALkJrhj0QYr61Z7-NDVZdvJAX3oHzQG_lg) ; на [C #](http://translate.googleusercontent.com/translate_c?hl=uk&sl=ru&u=http://ru.wikipedia.org/wiki/C_Sharp&prev=/search%3Fq%3D%25D0%259F%25D0%25BB%25D0%25B0%25D1%2582%25D1%2584%25D0%25BE%25D1%2580%25D0%25BC%25D0%25B0%2BNET%26hl%3Duk%26client%3Dfirefox-a%26hs%3D8xm%26rls%3Dorg.mozilla:uk:official%26prmd%3Divns&rurl=translate.google.com&usg=ALkJrhhpPOPGzfa8VBx6MRwAfgTLbWgnZQ) можна написати клас, успадкованих від класу, написаного на [Visual Basic. NET](http://translate.googleusercontent.com/translate_c?hl=uk&sl=ru&u=http://ru.wikipedia.org/wiki/Visual_Basic_.NET&prev=/search%3Fq%3D%25D0%259F%25D0%25BB%25D0%25B0%25D1%2582%25D1%2584%25D0%25BE%25D1%2580%25D0%25BC%25D0%25B0%2BNET%26hl%3Duk%26client%3Dfirefox-a%26hs%3D8xm%26rls%3Dorg.mozilla:uk:official%26prmd%3Divns&rurl=translate.google.com&usg=ALkJrhhJrmD25W3clsp8TsnsQe0Dq24Zlw) , а [виключення](http://translate.googleusercontent.com/translate_c?hl=uk&sl=ru&u=http://ru.wikipedia.org/w/index.php%3Ftitle%3D%25D0%2598%25D1%2581%25D0%25BA%25D0%25BB%25D1%258E%25D1%2587%25D0%25B5%25D0%25BD%25D0%25B8%25D1%258F%26action%3Dedit%26redlink%3D1&prev=/search%3Fq%3D%25D0%259F%25D0%25BB%25D0%25B0%25D1%2582%25D1%2584%25D0%25BE%25D1%2580%25D0%25BC%25D0%25B0%2BNET%26hl%3Duk%26client%3Dfirefox-a%26hs%3D8xm%26rls%3Dorg.mozilla:uk:official%26prmd%3Divns&rurl=translate.google.com&usg=ALkJrhgwPTkt-ee1Bv64pQN1bLI_YmKWvA) , створене методом, написаним на C # , може бути перехоплено і оброблено в Delphi. Кожна бібліотека (збірка) в. NET має свідчення про свою версію, що дозволяє усунути можливі конфлікти між різними версіями збірок.

. NET є патентованою технологією корпорації Microsoft.

*3.2. Проектування таблиць транслятора.*

Для того,щоб реалізувати лексичний аналіз створюємо таблицю, в якій розмістимо всі лексеми (ArrayList lexems = new ArrayList()), і таблицю (Hashtable ID = new Hashtable()) для ідентифікаторів, які будуть введені користувачем. Для реалізації таблиці лексем описаний наступний клас Token:

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **Lex** | **comment** | **type** |
| зберігання номера лексеми | зберігання лексеми | зберігання класу лексеми | зберігання коду лексеми |

public class Token

{

private int id=0;

private int type;

private string lex = "";

private string comment = "";

public int ID

{

get { return id; }

set { id = value; }

}

public int Type

{

get { return type; }

set { type = value; }

}

public string Lex

{

get { return lex; }

set { lex = value; }

}

public string Comment

{

get { return comment; }

set { comment = value; }

}

}

Атрибути лексем з використовуються при синтаксичному аналізі. Лексеми мають наступні коди:

-1) - Нерозпізнана лексема

0) - MAINPROGRAM

1) - DATA

2) – INTEGER

3) - BOOLEAN

4) - <ident>

5) - <num>

6) - :=

7) - TRUE

8) - FALSE

9) - ,

10) - ;

11) - START

12) - INPUT

13) - OUTPUT

14) - (

15) - )

16) - +

17) - -

18) - MUL

19) - DIV

20) - MOD

21) – ==

22) - !=

23) - LT

24) - GT

25) - !!

26) - AND

27) - OR

28) - <string>

29) – FOR

30) – END

31) – DO

32) – DOWNTO

Атрибути використовуються генератором коду для формування відповідних процедур мовою асемблер.

Для зберігання таблиці ідентифікаторів використовується Hashtable ID = new Hashtable()

Полем таблиці є клас ID\_info ,до якого входять такі поля:

|  |  |  |
| --- | --- | --- |
| **Id** | **Var\_value** | **type** |
| зберігання ідентифікаторів | зберігання типу змінної | зберігання значення змінної |

public class ID\_info

{

private string id="";

private string type = "";

private string Var\_value = "";

public string ID

{

get { return id; }

set { id = value; }

}

public string Type

{

get { return type; }

set { type = value; }

}

public string Var\_Value

{

get { return Var\_value; }

set { Var\_value = value; }

}

}

*3.3. Розробка лексичного аналізатора.*

Лексичний аналіз – це перша фаза трансляції, яка застосовується для групування символів вхідного ланцюга в більші конструкції, що називаються лексемами. З кожною лексемою зв’язано два поняття:

Клас лексеми, що визначає загальну назву для категорії елементів, що мають спільні властивості (наприклад, ідентифікатор, ціле число, рядок символів і т. д.).

Значення лексеми, що визначає підрядок символів вхідного ланцюга, що відповідають розпізнаному класу лексеми. В залежності від класу, значення лексеми може бути перетворено у внутрішнє представлення вже на етапі лексичного аналізу. Так, наприклад, роблять з числами, перетворюючи їх в машинне двійкове представлення, що забезпечує більш компактне зберігання і перевірку правильності діапазону на ранній стадії аналізу.

*3.3.1. Розробка граф-схеми алгоритму*

Лексичний аналізатор був розроблений на основі скінченного автомата.

Скінченний автомат, є особливим видом [автомату](http://uk.wikipedia.org/wiki/%D0%90%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82) — абстракції, що використовується для описання шляху зміни стану об'єкта в залежності від досягнутого стану та інформації отриманої ззовні. Його особливістю є [скінченність множини](http://uk.wikipedia.org/wiki/%D0%A1%D0%BA%D1%96%D0%BD%D1%87%D0%B5%D0%BD%D0%BD%D0%B0_%D0%BC%D0%BD%D0%BE%D0%B6%D0%B8%D0%BD%D0%B0) станів автомату. Розбір виконується взалежності від кількості символів з яких складається лексема. Згідно даного алгоритму лексеми можуть бути одно символьні, двосимвольні та багатосимвольні. Для багатосимвольних лексем розроблено функцію, яка виділяє лексеми.



Блок 1 – *Початок роботи лексичного аналізатора*

Блок 2 – *Відкриття і порядкове читання вхідного файлу*

Блок 3 – *Перевірка чи кінець файлу*

Блок 4 – *Виділення першого символу нової лексеми*

Блок 5 – *Перевіряє чи перший символ A-Z*

Блок 6 – *Розпізнання ключових слів*

Блок 7 – *Перевіряє чи перший символ \_*

Блок 8 – *Розпізнання ідентифікаторів*

Блок 9 – *Перевіряє чи перший символ 0-9*

Блок 10 – *Розпізнання числових констант*

Блок 11 – *Перевіряє чи перший символ “*

Блок 12 – *Розпізнання рядкових констант*

Блок 13 – *Перевіряє чи символ є оператор*

Блок 14 – *Зберігання до таблиці лексем*

Блок 15 – *Перевіряє чи перший символ є %*

Блок 16 *– Розпізнання коментарів*

Блок 17 *– Перевіряє чи символ є розділювач*

Блок 18 – *Зберігання до таблиці лексем*

Блок 19 – *Перевірка чи лексема*

Блок 20 – Зберігання нерозпізнаної лексеми

Блок 21 – *Завершення роботи*

*3.3.2 Опис програми реалізації лексичного аналізатора*

Програма по одному символу читає із вхідного тексту , котрий дописується до поточної лексеми, якщо зустрічається пробіл чи символ переходу на новий рядок, то вони пропускаються. Після цього відбувається перевірка на наявність лексеми в таблиці ключових слів та таблиці ідентифікаторів. Якщо лексема не виявлена, то формується повідомлення про помилку. Під час цього встановлюються прапорці при знаходженні коментарів чи рядкових констант для виводу, тоді перевірка не відбувається, оскільки в такому випадку лексеми можуть бути якими-завгодно. Прочитаний рядок передає як параметр функції Lexical\_Analyzer . В лексичному аналізаторі для розпізнання використовуються наступні функції:

Lexical\_Analyzer () тут відбувається аналіз першого символу нової лексеми:

A-Z **=>** запуск *Parse*Keys()

\_ ***=>*** запуск *Parse*ID()

0-9 або - **=>** запуск *Parse*Const()

==,!=,!!,;, , ,(,)  **=>** збереження лексеми

% **=>**  запуск *Parse*Com ()

“ **=>** запускається функція *Parse*Str()

не є жодним з вищезгаданих **=>** запуск RozbirNoTok()

*ParseID()-*розбираються символи, які можуть містити ідентифікатори і потім відбуваються його збереження.

*ParseKeys()-* розбираються символи, які можуть міститись в ключових словах і здійснюється перевірка чи ці символи є ключовим словом і якщо це так то зберігає його, як лексему

*ParseCom() -*перевіряє чи 2 символ \*,якщо це так то цей рядок є коментарем.

*ParseConst()-* розбираються символи, які можуть містити числові константи і зберігає лексему

*ParseNoLex()-* пошук розділювача,рахується що всі символи, які були виділені є нерозпізнаною лексемою

*ParseStr() -* відбувається пошук символа завершення рядка,всі символи,що містяться між початком і кінцем вважається рядковою константою.

**3.4.Розробка синтаксичного аналізатора**

Першим етапом синтаксичного аналізу є синтаксичний розбір. На етапі його виконання підтверджується то, що вхідний ланцюжок символів є програмою, а окремі підланцюжки складають синтаксично правильні програмні об'єкти. Після чого здійснюється аналіз їх семантичної коректності.

Розбір призначений для перевірки того,чи ланцюжок відповідаю якомусь правилу з сукупності правил визначеної граматикою. Синтаксичний розбір виконується розпізнавачами. Якщо на будь – якому рівні ланцюжок не відповідає ніякому правилу,то ми дістанемо загальну відмову.

Аналізатор вибирає лексему з сукупності лексем аналізуючи її визначає чи вона може стояти в цьому місці, якщо так, то заштовхує її в стек. Після чого знов вибирає лексему і перевіряє чи вона може стояти в цьому місці,якщо так то її також заштовхує в стек. Ця процедура виконується доти, доки ми не дійдемо до лексеми,яка завершує певну конструкцію. Тоді ця конструкція виштовхується зі стеку,згортається і в згорненому вигляді назад заштовхується в стек. Цей метод називається методом синтаксичного аналізу на базі магазинного автомата, аналіз проводиться від листків до кореня.

Основним завданням семантичного аналізатора є перевірка типів. Сама програма перевірки типів базується на інформації про синтаксичні конструкції мови, представлення типів і правилах присвоєння типів конструкціям мови.

*3.4.1 Розробка граф-схеми алгоритму*



*Рис 2.Граф-схема роботи синтаксичного аналізатора*

Блок 1 – Початок синтаксичного аналізатора

Блок 2 – Перевірка чи не кінець

Блок 3 – Виділення пешого символа лексеми

Блок 4 – Перевірка чи наступний символ GET

Блок 5 – Зсув до кінця конструкції

Блок 6 – Перевірка чи конструкція розпізнана

Блок 7 – Згортка конструкції GET

Блок 8 - Перевіряє чи наступний символ PUT

Блок 9 – Зсув до кінця конструкції

Блок 10 – Перевірка чи конструкція розпізнана

Блок 11 – Згортка конструкції PUT

Блок 12 - Перевіряє чи наступний символ FOR

Блок 13 – Зсув до кінця конструкції

Блок 14 - Перевірка чи конструкція розпізнана

Блок 15 - Згортка конструкції *FOR*

Блок 16 - Перевіряє чи наступний символ вираз

Блок 17 - Зсув до кінця конструкції

Блок 18 - Перевірка чи конструкція розпізнана

Блок 19 – Згортка виразу

Блок 20 – Конструкція нерозпізнана

Блок 21 - Перевіряє чи наступний символ START

Блок 22 – Зсув до кінця конструкції

Блок 23 - Перевірка чи конструкція розпізнана

Блок 24 – Згортка блоку коду

Блок 25 – Перевіряє чи наступний символ тип даних

Блок 26 – Зсув до кінця конструкції

Блок 27 - Перевірка чи конструкція розпізнана

Блок 28 – Згортка оголошення змінних

Блок 29 – Перевіряє чи наступний символ DATA

Блок 30 – Зсув до кінця конструкції

Блок 31 - Перевірка чи конструкція розпізнана

Блок 32 – Згортка блоку даних

Блок 33 – Згортка всієї програми

Блок 34 – Кінець

Блок 35 – Нерозпізнана конструкція

Блок 36 – Кінець

*3.4.2 Опис програми реалізації синтаксичного та семантичного аналізатора*

На вхід синтаксичному аналізатору подається таблиця лексем. Програма почерзі заштовхує всі лексеми в магазин (стек) поки не дійде до «листків дерева». Зустрівши конструкції які є листками синтаксичний аналізатор починай перевірку відповідності конструкцій з правилами вхідної граматики. Конструкція згортається і стає «листком дерева». Аналіз триває доти поки програма не згорне всі конструкції і не дійде до «кореня дерева», яким є термінал Program.

В програмі використовуються 2 функції:

* Фуккція зсуву – шукає кінець конструкції і перевіряє чи наступний символ відповідає допустимий в даному контексті програми
* Функція згортки – витягує з стеку розпізнану конструкцію, згортає її і заштовхує в стек у згорнутому вигляді.

В аналізаторі присутні функції перетворення виразів з інфіксної в постфіксну форми і перевірки на коректність використання у виразах відповідних змінних.

Всі помилки виявлені на етапі синтаксичного аналізу заносяться в таблицю помилок. Конструкції, які не згортаються не зберігаються в магазині, це забезпечує продовження роботи транслятора при виявленні помилки.

**3.5.Розробка генератора коду**

*3.5.1 Розробка граф-схеми алгоритму*

Генерація коду є тісно пов\*язаним з синтаксичним аналізом,зразу після розпізнавання генерується код.



Блок 1 – Початок генератора коду

Блок 2 – Перевірка чи не кінець

Блок 3 – Виділення пешого символа лексеми

Блок 4 – Перевірка чи наступний символ GET

Блок 5 – Зсув до кінця конструкції

Блок 6 – Перевірка чи конструкція розпізнана

Блок 7 - Виконання функції GenInput()

Блок 8 - Перевіряє чи наступний символ PUT

Блок 9 – Зсув до кінця конструкції

Блок 10 – Перевірка чи конструкція розпізнана

Блок 11 – Виконання функції GenOutput()

Блок 12 - Перевіряє чи наступний символ FOR

Блок 13 – Зсув до кінця конструкції

Блок 14 - Перевірка чи конструкція розпізнана

Блок 15 - Виконання функції GenFor()

Блок 16 - Перевіряє чи наступний символ вираз

Блок 17 - Зсув до кінця конструкції

Блок 18 - Перевірка чи конструкція розпізнана

Блок 19 - Генерація виразів

Блок 20 – Конструкція нерозпізнана

Блок 21 - Перевіряє чи наступний символ START

Блок 22 – Зсув до кінця конструкції

Блок 23 - Перевірка чи конструкція розпізнана

Блок 24 – Виконання функції GenStart()

Блок 25 – Перевіряє чи наступний символ тип даних

Блок 26 – Зсув до кінця конструкції

Блок 27 - Перевірка чи конструкція розпізнана

Блок 28 – Виконання функції GenType()

Блок 29 – Перевіряє чи наступний символ DATA

Блок 30 – Зсув до кінця конструкції

Блок 31 - Перевірка чи конструкція розпізнана

Блок 32 – Виконання функції GenData()

Блок 33 – Виконання функції GenMainProg()

Блок 34 – Кінець

Блок 35 – Нерозпізнана конструкція

Блок 36 – Кінець

*3.5.2 Опис програми реалізації генератора коду*

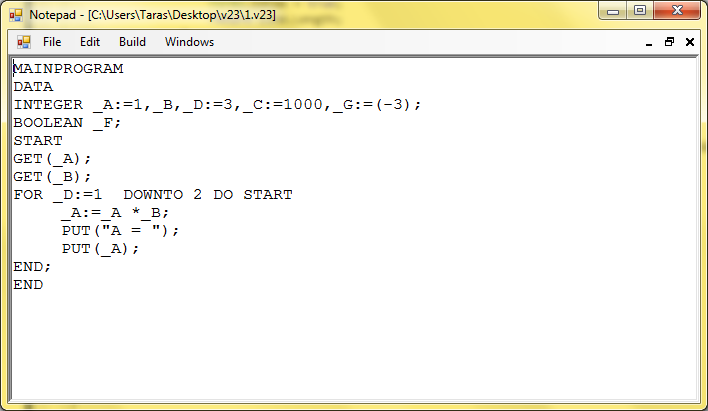
Програма перебирає по черзі лексеми із таблиці лексем. В залежності від коду знайденої лексеми в проміжному представленні програми буде вставлено відповідний еквівалентний асемблерний код. Для прикладу, коли зустрінеться ключове слово Data, вставиться опис сегменту даних,а при зустрічі ключового слова Input, то згенерується виклик функції Input.

Програма розроблена таким чином, що в асемблер пустий код генеруватися не буде. Для виключення можливості помилок імена ідентифікаторів будуть генеруватися в асемблерний код зі змінами.

В розробленому трансляторі генератор коду послідовно викликає функції,які записують у вихідний файл частини коду. Створюється відповідний вихідний код взалежності від лексем.

**4. Опис інтерфейсу та інструкції користувачеві**

Програму Notepad можна запустити з виконавчого файлу G23.exe. G23 є програмою, яка працює на платформі .NET і є візуальною програмою(повинна бути встановлена платформа net framework 2.0). Notepad містить редактор тексту і транслятор. Редактор тексту дозволяє відкривати файли з розширенням \*.v23.



*Рис 4.Вікно редактора тексту*

В розробленому редакторі реалізовані такі можливості, як:введення і редагування тексту,відкриття нового файл,збереження файлу,закриття файлу,копіювання тексту.

Тепер потрібно ввести код програми або відкрити готовий для того щоб його транслювати. Для того щоб транслювати програму потрібно в меню вибрати пункт Build або натиснути клавішу F5.Після чого ми отримаємо вікно транслятора,яке складається з 5 вкладок:

* Таблиця лексем
* Таблиця ідентифікаторів
* Таблиця рядкових констант
* Таблиця помилок
* Файл асемблерного коду



*Рис5.Вікно транслятора*

**5. ВІДЛАГОДЖЕННЯ ТА ТЕСТУВАННЯ ПРОГРАМИ**

Відлагодження програми відбувається та тестування програми відбувається в покроковому режимі в середовищі Visual Studio 2010.З використанням точок зупинки, що дає можливість знайти місце логічної помилки. В покроковому режимі у нас є можливість прослідкувати за значеннями змінних.

**5.1. Виявлення лексичних помилок.**

Нерозпізнана лексема – це єдина помилка яка може виникнути на етапі лексичного аналізу. Коли ця помилка все таки виникла,то на етапі синтаксичного аналізу буде згенерована помилка.

**5.2. Виявлення синтаксичних помилок.**

На етапі синтаксичного аналізу виявляється основна кількість помилок. Ці помилки пов’язані з невірними записами конструкцій вхідної мови. Всі помилки виявленні на етапі синтаксичного аналізу заносяться в таблицю помилок.

**5.3. Виявлення семантичних помилок.**

Виявлення семантичних помилок – це перевірка числових констант на відповідність типу, перевірка на коректність використання змінних різних типів у цілочисельних і логічних виразах.

**5.4. Загальна перевірка коректності роботи транслятора.**

Загальна перевірка коректності роботи транслятора полягає в транслюванні коректної вхідної програми з використанням всіх можливостей мови,в асемблерний код та перевірці на правильність виконання програми, яку потрібно попередньо скомпілювати і странслювати за допомогою ml.exe

Текст програми:

MAINPROGRAM

DATA

INTEGER \_A:=1,\_B,\_D:=3,\_C:=1000,\_G:=(-3);

BOOLEAN \_F;

START

GET(\_A);

GET(\_B);

FOR \_D:=1 DOWNTO 2 DO START

\_A:=\_A \*\_B;

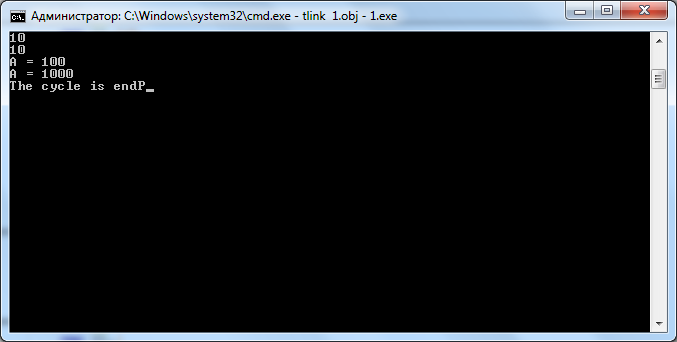
PUT("A = ");

PUT(\_A);

END;

END

Результат роботи програми:



**Висновки**

Підчас виконання курсової роботи:

1. Складено формальний опис мови програмування v23 у формі розширеної нотації Бекуса-Наура, дано опис усіх символів та ключових слів.

2. Створено транслятор мови програмування v23, а саме:

2.1.1. Розроблено лексичний аналізатор, здатний розпізнавати лексеми, що є описані в формальному описі мови програмування.

2.1.2. Розроблено синтаксичний аналізатор на основі методу зсув-згортка.

2.1.3. Розроблено генератор коду, який починає свою роботу під час коректного виявлення кожної конструкції і генерує код який відповідає кожній конструкції вхідної мови.

3. Проведене тестування транслятора за допомогою тестових програм за наступними пунктами: Виявлення лексичних помилок. Виявлення синтаксичних помилок. Виявлення семантичних помилок. Загальна перевірка роботи компілятора.

**СПИСОК ЛІТЕРАТУРИ**

1. Шильдт Г. С#. – Санкт-Петербург: BXV, 2007. – 688 с.
2. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. – М.: Мир, 1978. – т.1, 612 с.
3. А.Ю. Молчанов «Системное програмное обеспечение» Лабораторный практикум 2005р.
4. Компаниец Р.И., Маньков Е.В., Филатов Н.Е. Системное программирование. Основы построения трансляторов. – СПб.: КОРОНА принт, 2004. – 256 с.
5. Ваймгартен Ф. Трансляция языков программирования. – М.: Мир, 1977.

*Додатки*

*Додаток А*

*Лістинг програми*

using System;

using System.Collections.Generic;

using System.Text;

using System.Windows.Forms;

using System.Collections;

namespace Notepad

{

public class Const\_Strings

{

private string str\_value = "";

private int num\_lex;

public string Value

{

get { return str\_value; }

set { str\_value = value; }

}

public int Number

{

get { return num\_lex; }

set { num\_lex = value; }

}

}

public class Iden\_info

{

private string id="";

private string type = "";

private string Id\_value = "";

public string Id

{

get { return id; }

set { id = value; }

}

public string Type

{

get { return type; }

set { type = value; }

}

public string Value

{

get { return Id\_value; }

set { Id\_value = value; }

}

}

public class Lexem

{

private int id=0;

private int type = -1;

private string lex = "";

private string comments = "";

public int Id

{

get { return id; }

set { id = value; }

}

public int Type

{

get { return type; }

set { type = value; }

}

public string Lex

{

get { return lex; }

set { lex = value; }

}

public string Comments

{

get { return comments; }

set { comments = value; }

}

}

static class Parser

{

public static int num = 0;

public static bool isStart = false;

public static string Comments;

public static Hashtable identy = new Hashtable();

public static Hashtable labels = new Hashtable();

public static Hashtable const\_str = new Hashtable();

public static Hashtable const\_int = new Hashtable();

public static ArrayList lexems = new ArrayList();

public static ArrayList lines = new ArrayList();

static Iden\_info id\_info;

static Const\_Strings strs;

static Lexem lexem;

public static void Parse(string line)

{

line = line + '\n';

for (int i = 0; i < line.Length;i++)

{

if (line.Length == 1)

continue;

if (Form1.isCom)

i = ParseComments(line, i) - 1;

if (line[i] >= 'A' && line[i] <= 'Z')

{

i = ParseKeyWords(line, i) - 1;

continue;

}

if (line[i] == '\_')

{

i = ParseIdent(line, i) - 1;

continue;

}

if ((line[i] >= '0' && line[i] <= '9'))

{

i = ParseNum(line, i)-1;

continue;

}

switch (line[i])

{

case '+':

lexem = new Lexem();

lexem.Id = num;

num++;

lexem.Lex = "+";

lexem.Type = 16;

lexem.Comments = "Арифметичний оператор";

lexems.Add(lexem);

break;

case '-':

lexem = new Lexem();

lexem.Id = num;

num++;

lexem.Lex = "-";

lexem.Type = 17;

lexem.Comments = "Арифметичний оператор";

lexems.Add(lexem);

break;

case '\*':

lexem = new Lexem();

lexem.Id = num;

num++;

lexem.Lex = "\*";

lexem.Type = 18;

lexem.Comments = "Арифметичний оператор";

lexems.Add(lexem);

break;

case '!':

if (line[i + 1] == '!')

{

lexem = new Lexem();

lexem.Id = num;

num++;

lexem.Lex = "!!";

lexem.Type = 25;

lexem.Comments = "Оператор NOT";

lexems.Add(lexem);

i++;

}

else

{

if (line[i + 1] == '=')

{

lexem = new Lexem();

lexem.Id = num;

num++;

lexem.Lex = "!=";

lexem.Type = 22;

lexem.Comments = "Оператор не рівно";

lexems.Add(lexem);

i++;

}

else

{

lexem = new Lexem();

lexem.Id = num;

num++;

lexem.Lex = "!";

lexem.Type = -1;

lexem.Comments = "Нерозпізнана лексема";

lexems.Add(lexem);

}

}

break;

case '=':

if (line[i + 1] == '=')

{

lexem = new Lexem();

lexem.Id = num;

num++;

lexem.Lex = "==";

lexem.Type = 23;

lexem.Comments = "Оператор рівно";

lexems.Add(lexem);

i++;

}

else

{

lexem = new Lexem();

lexem.Id = num;

num++;

lexem.Lex = "=";

lexem.Type = -1;

lexem.Comments = "Нерозпізнана лексема";

lexems.Add(lexem);

}

break;

case ':':

if (line[i + 1] == '=')

{

lexem = new Lexem();

lexem.Id = num;

num++;

lexem.Lex = ":=";

lexem.Type = 6;

lexem.Comments = "Оператор присвоєння";

lexems.Add(lexem);

i++;

}

else

{

lexem = new Lexem();

lexem.Id = num;

num++;

lexem.Lex = ":";

lexem.Type = -1;

lexem.Comments = "Нерозпізнана лексема";

lexems.Add(lexem);

}

break;

case '\"':

i = ParseString(line, i)-1;

break;

case '%':

i = ParseComments(line, i) - 1;

break;

case ';':

lexem = new Lexem();

lexem.Id = num;

num++;

lexem.Lex = ";";

lexem.Type = 10;

lexem.Comments = "Розділювач";

lexems.Add(lexem);

break;

case ',':

lexem = new Lexem();

lexem.Id = num;

num++;

lexem.Lex = ",";

lexem.Type = 9;

lexem.Comments = "Розділювач";

lexems.Add(lexem);

break;

case ')':

lexem = new Lexem();

lexem.Id = num;

num++;

lexem.Lex = ")";

lexem.Type = 15;

lexem.Comments = "Розділювач";

lexems.Add(lexem);

break;

case '(':

lexem = new Lexem();

lexem.Id = num;

num++;

lexem.Lex = "(";

lexem.Type = 14;

lexem.Comments = "Розділювач";

lexems.Add(lexem);

break;

case' ':

break;

case '\t':

break;

case '\n':

lines.Add(num);

break;

default:

i = ParseNoLex(line, i) - 1;

break;

}

}

}

private static int ParseIdent(string line, int position)

{

int i = position + 1;

string id;

while (line[i] >= 'A' && line[i] <= 'Z' || line[i] >= '0' && line[i] <= '9')

{

i++;

}

id = line.Remove(0, position);

id = id.Remove(i - position);

lexem = new Lexem();

lexem.Id = num;

num++;

lexem.Lex = id;

lexem.Type = 4;

lexem.Comments = "Ідентифікатор";

lexems.Add(lexem);

return i;

}

private static int ParseKeyWords(string line,int position)

{

string Key\_Word;

int i = position+1;

while (line[i] >= 'A' && line[i] <= 'Z')

{

i++;

}

Key\_Word = line.Remove(0, position);

Key\_Word = Key\_Word.Remove(i-position);

string[] KeyWords = { "MAINPROGRAM", "DATA", "START", "END", "GET", "PUT", "FOR", "DO", "AND", "DIV", "MOD", "LT", "GT", "INTEGER", "BOOLEAN", "TRUE", "FALSE", "OR", "DOWNTO" };

foreach (string st in KeyWords)

{

if (st.Equals(Key\_Word))

{

lexem = new Lexem();

lexem.Id = num;

num++;

switch (Key\_Word)

{

case "MAINPROGRAM":

lexem.Type = 0;

break;

case "DATA":

lexem.Type = 1;

break;

case "START":

isStart = true;

lexem.Type = 11;

break;

case "END":

lexem.Type = 30;

break;

case "GET":

lexem.Type = 12;

break;

case "PUT":

lexem.Type = 13;

break;

case "FOR":

lexem.Type = 31;

break;

case "AND":

lexem.Type = 26;

break;

case "DIV":

lexem.Type = 19;

break;

case "BOOLEAN":

lexem.Type = 3;

break;

case "INTEGER":

lexem.Type = 2;

break;

case "TRUE":

lexem.Type = 7;

break;

case "FALSE":

lexem.Type = 8;

break;

case "MOD":

lexem.Type = 20;

break;

case "LT":

lexem.Type = 23;

break;

case "GT":

lexem.Type = 24;

break;

case "DO":

lexem.Type = 32;

break;

case "DOWNTO":

lexem.Type = 29;

break;

case "OR":

lexem.Type = 27;

break;

}

lexem.Lex = Key\_Word;

lexem.Comments = "Ключове слово";

lexems.Add(lexem);

return i;

}

}

lexem = new Lexem();

lexem.Id = num;

num++;

lexem.Lex = Key\_Word;

lexem.Type = -1;

lexem.Comments = "Нерозпізна лексема";

lexems.Add(lexem);

return i;

}

private static int ParseNum (string line,int position)

{

int i = position;

string Num;

i++;

while (line[i] >= '0' && line[i] <= '9')

{

i++;

}

Num = line.Remove(0, position);

Num = Num.Remove(i - position);

lexem = new Lexem();

lexem.Id = num;

num++;

lexem.Lex = Num;

try

{

if (isStart)

const\_int.Add(Num, Num);

}

catch { }

lexem.Type = 5;

lexem.Comments = "Числова константа";

lexems.Add(lexem);

return i;

}

private static int ParseString(string line,int position)

{

int i=position+1;

string st;

while (i < line.Length)

{

if (line[i] == '\"')

{

i++;

break;

}

i++;

}

if (i == line.Length)

{

lexem = new Lexem();

lexem.Id = num;

num++;

lexem.Lex = "\"";

lexem.Type = -1;

lexem.Comments = "Нерозпізна лексема";

lexems.Add(lexem);

return position + 1;

}

else

{

st = line.Remove(0, position);

st = st.Remove(i - position);

lexem = new Lexem();

lexem.Id = num;

num++;

lexem.Lex = st;

strs = new Const\_Strings();

strs.Value = st;

strs.Number = num;

try

{

const\_str.Add(st, strs);

}

catch { };

lexem.Type = 28;

lexem.Comments = "Рядкова константа";

lexems.Add(lexem);

return i;

}

}

private static int ParseComments(string line, int position)

{

int i = position + 1;

string com;

if (line[i] != '%' && !Form1.isCom)

{

lexem = new Lexem();

lexem.Id = num;

num++;

lexem.Lex = "%";

lexem.Comments = "Нерозпізна лексема";

lexems.Add(lexem);

return position + 1;

}

else

{

if (line.LastIndexOf("%%") != -1)

{

com = line.Remove(0, position);

com = com.Remove(line.LastIndexOf("%%") + 2 - position);

Comments = Comments + com;

lexem = new Lexem();

lexem.Id = num;

num++;

lexem.Lex = Comments;

Comments = "";

lexem.Comments = "Коментар";

//lexems.Add(lexem);

Form1.isCom = false;

return line.LastIndexOf("%%") + 3;

}

{

com = line.Remove(0, position);

Comments = Comments + com;

Form1.isCom = true;

return line.Length;

}

}

}

private static int ParseNoLex(string line, int position)

{

int i = position;

string NoLex;

while (line[i] != ' ' && line[i] != '\t' && line[i] != '\n' && line[i] != ')' && line[i] != '(' && line[i] != ',' && line[i] != ';')

{

i++;

}

NoLex = line.Remove(0, position);

NoLex = NoLex.Remove(i - position);

lexem = new Lexem();

lexem.Id = num;

num++;

lexem.Lex = NoLex;

lexem.Type = -1;

lexem.Comments = "Нерозпізна лексема";

lexems.Add(lexem);

return i;

}

}

}

using System;

using System.Collections.Generic;

using System.Text;

using System.Collections;

using System.Windows.Forms;

namespace Notepad

{

public class Error

{

int NumLex;

string TextErr;

public int Num\_Lex

{

get { return NumLex; }

set { NumLex = value; }

}

public string Error\_Text

{

get { return TextErr; }

set { TextErr = value; }

}

}

class Syntax

{

static Error err;

static PreTabel pt = new PreTabel();

static ArrayList Row;

public static ArrayList errors = new ArrayList();

static Stack magazin = new Stack();

static Stack construct = new Stack();

static Lexem Pred, Next, Temp;

public static int i = 0;

static string state = "";

static bool isEqu=false;

static int Exit = 0,Start,Finish;

static int j = 0;

static Iden\_info id\_info;

static string Type\_var = "";

static string ident = "";

static string Value = "";

static bool isMinus = false;

static bool isGet = false,isPut = false,isWhile=false;

static int Put\_Start=0, Put\_Finish=0;

static bool isString = false;

public static void Analiz(ArrayList lexems)

{

Pred = (Lexem)lexems[i++];

if (Pred.Type == 0)

{

construct.Push(Pred);

magazin.Push(Pred);

Generation.GenerateBegin();

}

else

{

err = new Error();

err.Error\_Text = "Програма повинна розпочинатись з ключового слова Program";

err.Num\_Lex = i - 1;

errors.Add(err);

return;

}

Next = (Lexem)lexems[i++];

while (i < lexems.Count)

{

try

{

Temp = (Lexem)construct.Peek();

if (Next.Type == 11 && Temp.Type == 1 || Temp.Type == 0 && Next.Type == 11)

{

Pred = Temp;

}

else

if (Next.Type == 1 || Next.Type == 2 || Next.Type == 3 || Next.Type == 11 || Next.Type == 14 ||

(Next.Type == 4 && Pred.Type == 9) || (Next.Type == 4 && Pred.Type == 2) || (Next.Type == 4 && Pred.Type == 3))

construct.Push(Next);

}

catch { }

if (Next.Type == 2)

Type\_var = "INTEGER";

if (Next.Type == 3)

Type\_var = "BOOLEAN";

if (Next.Type == 4)

ident = Next.Lex;

if (Next.Type == 17)

isMinus = true;

if (Next.Type == 5)

if (!isMinus)

Value = Next.Lex;

else

{

Value = "-" + Next.Lex;

isMinus = false;

}

if (Next.Type == 7)

Value = "1";

if (Next.Type == 8)

Value = "0";

if (Next.Type == 9 || Next.Type == 10)

{

id\_info = new Iden\_info();

id\_info.Id = ident;

id\_info.Type = Type\_var;

id\_info.Value = Value;

try

{

if (ident != "")

Parser.identy.Add(ident, id\_info);

}

catch {

err = new Error();

err.Error\_Text = "Змінна вже оголошена";

err.Num\_Lex = i - 1;

errors.Add(err);

return;

}

ident = "";

Value = "";

}

try

{

Row = (ArrayList)PreTabel.preTable[Pred.Type];

state = (string)Row[Next.Type];

}

catch {

err = new Error();

err.Error\_Text = "Нерозпізнана лексема";

err.Num\_Lex = i - 1;

errors.Add(err);

return;

}

switch (state)

{

case "<":

magazin.Push(Next);

Pred = Next;

Next = (Lexem)lexems[i++];

break;

case ">":

//try

//{

Temp = (Lexem)construct.Pop();

while (Temp != (Lexem)magazin.Pop()) { }

if (Pred.Type == 1 && Next.Type == 11 || Pred.Type == 0 && Next.Type == 11)

{

construct.Push(Next);

magazin.Push(Next);

Generation.GenereteVar();

Generation.GenereteDerlaration();

AnalizBegin(lexems, i - 1);

return;

}

if (!(Temp.Type == 4 && Next.Type == 10))

{

Pred = Next;

Next = (Lexem)lexems[i++];

}

//}

//catch {

// Pred = Next;

// Next = (Lexem)lexems[i++];

//}

break;

case "=":

Pred = Next;

Next = (Lexem)lexems[i++];

break;

case "-":

err = new Error();

err.Error\_Text = "Помилка в оголошенні змінних";

err.Num\_Lex = i - 1;

errors.Add(err);

return;

}

}

}

private static void AnalizBegin(ArrayList lexems, int i)

{

bool Type = true ;

Iden\_info id;

Pred = (Lexem)lexems[i++];

Next = (Lexem)lexems[i++];

while (i <= lexems.Count)

{

if (Next.Type == 12)

isGet = true;

if (Next.Type == 13)

isPut = true;

if (Next.Type == 12 || Next.Type == 13 || Next.Type == 31)

{

Type = true;

}

if (Next.Type == 31)

{

isWhile = true;

}

if (Next.Type == 4 && Pred.Type == 11 || Next.Type == 4 && Pred.Type == 10 || Next.Type == 4 && Pred.Type == 31 || Next.Type == 4 && Pred.Type == 32)

{

isEqu = true;

Exit = i - 1;

j = 0;

if (Next.Type == 4)

{

if (Parser.identy.ContainsKey(Next.Lex))

{

id = (Iden\_info)Parser.identy[Next.Lex];

Type = (id.Type == "INTEGER") ? true : false;

}

}

}

if (!isWhile && Next.Type == 4)

isDeclaredVariable(Next, Type, i - 1);

if (Next.Type == 32)

Generation.RepeatStart();

if (Next.Type == 10 && Pred.Type == 30)

Generation.GenerateUntil();

if (Next.Type == 5 && Pred.Type == 29)

Generation.GenereteInicialCycle(Next);

if (isGet && Pred.Type == 4 && Next.Type != 15)

{

err = new Error();

err.Error\_Text = "Невірний запис конструкції Get";

err.Num\_Lex = i;

errors.Add(err);

return;

}

if (isGet && Pred.Type == 4 && Next.Type == 15)

{

Lexem lx = (Lexem)lexems[i-2];

Generation.GenerateReadVar(lx.Lex);

Infix\_to\_Postfix(lexems, i - 2, i - 1, false);

}

if (isPut && Pred.Type == 14 && Next.Type == 4)

Put\_Start = i - 1;

if (isPut && Pred.Type == 14 && Next.Type == 28)

isString = true;

if (isPut && Pred.Type == 15 && Next.Type == 10)

{

if (!isString)

{

Put\_Finish = i - 3;

if (Put\_Start != 0 && Put\_Finish != 0 && Put\_Start != lexems.Count)

{

Infix\_to\_Postfix(lexems, Put\_Start, Put\_Finish, false);

Generation.GenerateWrite(true);

Put\_Start = 0;

Put\_Finish = 0;

}

}

}

if (isPut && isString)

{

Lexem lx = (Lexem)lexems[i - 1];

Generation.GenerateStr(lx.Lex);

Generation.GenerateWrite(false);

isString = false;

}

if (Next.Type == 11 || Next.Type == 12 || Next.Type == 13 || Next.Type == 14 || Next.Type == 31 ||

(Next.Type == 4 && Pred.Type == 10) || (Next.Type == 4 && Pred.Type == 11) || (Next.Type == 4 && Pred.Type == 31) || (Next.Type == 4 && Pred.Type == 32))

{

construct.Push(Next);

}

if (Next.Type == 14)

j++;

try

{

Row = (ArrayList)PreTabel.preTable[Pred.Type];

state = (string)Row[Next.Type];

}

catch

{

err = new Error();

err.Error\_Text = "Нерозпізнана лексема";

err.Num\_Lex = i - 1;

errors.Add(err);

return;

}

switch (state)

{

case "<":

magazin.Push(Next);

Pred = Next;

Next = (Lexem)lexems[i++];

break;

case ">":

if (Next.Type == 10 && isGet)

isGet = false;

if (Next.Type == 10 && isPut)

isPut = false;

if (Next.Type == 10 || Next.Type == 29)

Finish = i - 1;

if (Next.Type == 30 && i == lexems.Count)

{

Temp = (Lexem)construct.Pop();

if (Temp.Type != 11)

{

err = new Error();

err.Error\_Text = "Неправельне завершення програми, конструкції не завершені.";

err.Num\_Lex = lexems.Count - 1;

errors.Add(err);

return;

}

else

Generation.GenerateEndProgram();

return;

}

Temp = (Lexem)construct.Peek();

if (Next.Type == 30 && Temp.Type != 11)

{

Pred = Next;

Next = (Lexem)lexems[i++];

continue;

}

if (isEqu)

Start--;

Temp = (Lexem)construct.Pop();

if (Temp.Type == 4)

{

if (Type && Exit != 0)

{

if (Next.Type == 29)

Infix\_to\_Postfix(lexems, Exit, Finish, true);

else

Infix\_to\_Postfix(lexems, Exit, Finish, false);

}

else

{

Infix\_to\_Postfix\_b(lexems, Exit, Finish);

}

}

if (!isEqu)

while (Temp != (Lexem)magazin.Pop()) { }

else

{

construct.Push(Temp);

Temp = (Lexem)construct.Peek();

if (Temp.Type == 4)

{

Start = Start + i - j;

while (Exit < Start)

{

magazin.Pop();

Start--;

}

Temp = (Lexem)construct.Peek();

if (Temp.Type == 4)

{

construct.Pop();

}

Temp = (Lexem)magazin.Peek();

if (Temp.Type == 4)

{

magazin.Pop();

}

Start = 0;

Finish = 0;

Exit = 0;

isEqu = false;

}

else

{

while (Temp != (Lexem)magazin.Pop()) { Start--; }

construct.Pop();

//j--;

}

}

if (i == lexems.Count)

{

i++;

break;

}

else

{

Pred = Next;

Next = (Lexem)lexems[i++];

}

break;

case "=":

Pred = Next;

Next = (Lexem)lexems[i++];

break;

case "-":

err = new Error();

err.Error\_Text = "Задана лексема не може використовуватись в цій конструкції";

err.Num\_Lex = i - 1;

errors.Add(err);

return;

}

}

}

private static void isDeclaredVariable(Lexem lex, bool Type, int num)

{

if (!Parser.identy.ContainsKey(lex.Lex))

{

err = new Error();

err.Num\_Lex = num;

err.Error\_Text = "Змінна не оголошена";

errors.Add(err);

}

else

{

Iden\_info id = (Iden\_info)Parser.identy[lex.Lex];

if (Type)

{

if (id.Type != "INTEGER")

{

err = new Error();

err.Num\_Lex = num;

err.Error\_Text = "Змінна повинна бути типу INTEGER";

errors.Add(err);

}

}

else

{

if (id.Type != "BOOLEAN")

{

err = new Error();

err.Num\_Lex = num;

err.Error\_Text = "Змінна повинна бути типу BOOLEAN";

errors.Add(err);

}

}

}

}

static Stack stack;

static ArrayList postfix;

public static void Infix\_to\_Postfix(ArrayList lexems, int start, int finish, bool b)

{

stack = new System.Collections.Stack();

postfix = new ArrayList();

Lexem lexem, lex, Zero = new Lexem();

Lexem lexReturn = new Lexem();

lexReturn.Id = -1;

Zero.Id = 0;

Zero.Lex = "0";

Zero.Type = 35;

Lexem type;

lex = (Lexem)lexems[start + 1];

if (lex.Type == 6)

{

lexReturn = (Lexem)lexems[start];

start = start + 2;

}

for (int j = start; j <= finish; j++)

{

lexem = (Lexem)lexems[j];

lex = (Lexem)lexems[j + 1];

if (lexem.Type == 14 && lex.Type == 17)

postfix.Add(Zero);

switch (lexem.Type)

{

case 4:

postfix.Add(lexem);

break;

case 5:

postfix.Add(lexem);

break;

case 14:

stack.Push(lexem);

break;

case 15:

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

while (type.Type != 14)

{

postfix.Add(type);

if (stack.Count != 0)

type = (Lexem)stack.Pop();

else

{

err = new Error();

err.Num\_Lex = j;

err.Error\_Text = "Дизбаланс дужок";

errors.Add(err);

break;

}

}

}

break;

case 16:

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

while (stack.Count != 0 && type.Type != 14)

{

postfix.Add(type);

type = (Lexem)stack.Pop();

if (stack.Count == 0)

postfix.Add(type);

}

if (type.Type == 14)

stack.Push(type);

stack.Push(lexem);

}

else

stack.Push(lexem);

break;

case 17:

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

while (stack.Count != 0 && type.Type != 14)

{

postfix.Add(type);

type = (Lexem)stack.Pop();

if (stack.Count == 0)

postfix.Add(type);

}

if (type.Type == 14)

stack.Push(type);

stack.Push(lexem);

}

else

stack.Push(lexem);

break;

case 18:

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

switch (type.Type)

{

case 16:

stack.Push(type);

stack.Push(lexem);

break;

case 17:

stack.Push(type);

stack.Push(lexem);

break;

case 18:

postfix.Add(type);

stack.Push(lexem);

break;

case 19:

postfix.Add(type);

stack.Push(lexem);

break;

case 20:

postfix.Add(type);

stack.Push(lexem);

break;

default:

if (type.Type == 14)

{

stack.Push(type);

stack.Push(lexem);

}

break;

}

}

else

{

stack.Push(lexem);

}

break;

case 19:

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

switch (type.Type)

{

case 16:

stack.Push(type);

stack.Push(lexem);

break;

case 17:

stack.Push(type);

stack.Push(lexem);

break;

case 18:

postfix.Add(type);

stack.Push(lexem);

break;

case 19:

postfix.Add(type);

stack.Push(lexem);

break;

case 20:

postfix.Add(type);

stack.Push(lexem);

break;

default:

if (type.Type == 14)

{

stack.Push(type);

stack.Push(lexem);

}

break;

}

}

else

{

stack.Push(lexem);

}

break;

case 20:

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

switch (type.Type)

{

case 16:

stack.Push(type);

stack.Push(lexem);

break;

case 17:

stack.Push(type);

stack.Push(lexem);

break;

case 18:

postfix.Add(type);

stack.Push(lexem);

break;

case 19:

postfix.Add(type);

stack.Push(lexem);

break;

case 20:

postfix.Add(type);

stack.Push(lexem);

break;

default:

if (type.Type == 14)

{

stack.Push(type);

stack.Push(lexem);

}

break;

}

}

else

{

stack.Push(lexem);

}

break;

}

}

while (stack.Count != 0)

{

type = (Lexem)stack.Pop();

if (type.Type != 14)

postfix.Add(type);

else

{

err = new Error();

err.Num\_Lex = finish;

err.Error\_Text = "Дизбаланс дужок";

errors.Add(err);

break;

}

}

//string Mess="";

//foreach (Lexem lx in postfix)

//{

// Mess = Mess + lx.Lex;

//}

//MessageBox.Show(Mess);

Generation.GenereteExp\_i(postfix, lexReturn, isPut, b);

lexReturn.Id = -1;

}

public static void Infix\_to\_Postfix\_b(ArrayList lexems, int start, int finish)

{

stack = new System.Collections.Stack();

postfix = new ArrayList();

Lexem lexem,lex;

Lexem type, lexReturn\_b = new Lexem();

lexReturn\_b.Id = - 1;

lex = (Lexem)lexems[start + 1];

if (lex.Type == 6)

{

lexReturn\_b = (Lexem)lexems[start];

start = start + 2;

}

for (int j = start; j <= finish; j++)

{

lexem = (Lexem)lexems[j];

switch (lexem.Type)

{

case 4:

postfix.Add(lexem);

break;

case 5:

postfix.Add(lexem);

break;

case 14:

stack.Push(lexem);

break;

case 15:

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

while (type.Type != 14)

{

postfix.Add(type);

if (stack.Count != 0)

type = (Lexem)stack.Pop();

else

{

err = new Error();

err.Num\_Lex = j;

err.Error\_Text = "Дизбаланс дужок";

errors.Add(err);

break;

}

}

}

break;

case 25:

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

if (type.Type == 25)

{

postfix.Add(type);

}

else

{

stack.Push(type);

}

stack.Push(lexem);

}

else

{

stack.Push(lexem);

}

break;

case 26:

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

switch (type.Type)

{

case 21:

stack.Push(type);

stack.Push(lexem);

break;

case 14:

stack.Push(type);

stack.Push(lexem);

break;

case 25:

postfix.Add(type);

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

if (type.Type == 26)

postfix.Add(type);

else

stack.Push(type);

stack.Push(lexem);

}

else

{

stack.Push(lexem);

}

break;

case 26:

postfix.Add(type);

stack.Push(lexem);

break;

case 27:

stack.Push(type);

stack.Push(lexem);

break;

case 22:

stack.Push(type);

stack.Push(lexem);

break;

}

}

else

{

stack.Push(lexem);

}

break;

case 27:

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

switch (type.Type)

{

case 21:

stack.Push(type);

stack.Push(lexem);

break;

case 14:

stack.Push(type);

stack.Push(lexem);

break;

case 25:

postfix.Add(type);

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

while ((type.Type == 26 || type.Type == 27) && stack.Count != 0)

{

isWhile = true;

postfix.Add(type);

type = (Lexem)stack.Pop();

}

if (stack.Count == 0)

if (type.Type == 26 || type.Type == 27)

postfix.Add(type);

else

stack.Push(type);

if (isWhile)

{

stack.Push(type);

isWhile = false;

}

stack.Push(lexem);

}

else

{

stack.Push(lexem);

}

break;

case 26:

postfix.Add(type);

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

if (type.Type == 27)

postfix.Add(type);

else

stack.Push(type);

stack.Push(lexem);

}

else

{

stack.Push(lexem);

}

break;

case 27:

postfix.Add(type);

stack.Push(lexem);

break;

case 22:

stack.Push(type);

stack.Push(lexem);

break;

}

}

else

{

stack.Push(lexem);

}

break;

case 21:

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

switch (type.Type)

{

case 21:

postfix.Add(type);

stack.Push(lexem);

break;

case 14:

stack.Push(type);

stack.Push(lexem);

break;

case 25:

postfix.Add(type);

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

while ((type.Type == 26 || type.Type == 27 || type.Type == 21 || type.Type == 22 || type.Type == 23 || type.Type == 24) && stack.Count != 0)

{

isWhile = true;

postfix.Add(type);

type = (Lexem)stack.Pop();

}

if (stack.Count == 0)

if (type.Type == 26 || type.Type == 27 || type.Type == 21 || type.Type == 22 || type.Type == 23 || type.Type == 24)

postfix.Add(type);

else

stack.Push(type);

if (isWhile)

{

stack.Push(type);

isWhile = false;

}

stack.Push(lexem);

}

else

{

stack.Push(lexem);

}

break;

case 26:

postfix.Add(type);

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

while ((type.Type == 27 || type.Type == 21 || type.Type == 22 || type.Type == 23 || type.Type == 24) && stack.Count != 0)

{

isWhile = true;

postfix.Add(type);

type = (Lexem)stack.Pop();

}

if (stack.Count == 0)

stack.Push(type);

if (isWhile)

{

stack.Push(type);

isWhile = false;

}

stack.Push(lexem);

}

else

{

stack.Push(lexem);

}

break;

case 27:

postfix.Add(type);

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

if (type.Type == 21 || type.Type == 22 || type.Type == 23 || type.Type == 24)

postfix.Add(type);

else

stack.Push(type);

stack.Push(lexem);

}

else

{

stack.Push(lexem);

}

break;

case 22:

postfix.Add(type);

stack.Push(lexem);

break;

default:

if (type.Type == 23 || type.Type == 24)

{

postfix.Add(type);

stack.Push(lexem);

}

break;

}

}

else

{

stack.Push(lexem);

}

break;

case 22:

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

switch (type.Type)

{

case 21:

postfix.Add(type);

stack.Push(lexem);

break;

case 14:

stack.Push(type);

stack.Push(lexem);

break;

case 25:

postfix.Add(type);

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

while ((type.Type == 26 || type.Type == 27 || type.Type == 21 || type.Type == 22 || type.Type == 23 || type.Type == 24) && stack.Count != 0)

{

isWhile = true;

postfix.Add(type);

type = (Lexem)stack.Pop();

}

if (stack.Count == 0)

if (type.Type == 26 || type.Type == 27 || type.Type == 21 || type.Type == 22 || type.Type == 23 || type.Type == 24)

postfix.Add(type);

else

stack.Push(type);

if (isWhile)

{

stack.Push(type);

isWhile = false;

}

stack.Push(lexem);

}

else

{

stack.Push(lexem);

}

break;

case 26:

postfix.Add(type);

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

while ((type.Type == 27 || type.Type == 21 || type.Type == 22 || type.Type == 23 || type.Type == 24) && stack.Count != 0)

{

isWhile = true;

postfix.Add(type);

type = (Lexem)stack.Pop();

}

if (stack.Count == 0)

stack.Push(type);

if (isWhile)

{

stack.Push(type);

isWhile = false;

}

stack.Push(lexem);

}

else

{

stack.Push(lexem);

}

break;

case 27:

postfix.Add(type);

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

if (type.Type == 21 || type.Type == 22 || type.Type == 23 || type.Type == 24)

postfix.Add(type);

else

stack.Push(type);

stack.Push(lexem);

}

else

{

stack.Push(lexem);

}

break;

case 22:

postfix.Add(type);

stack.Push(lexem);

break;

default:

if (type.Type == 23 || type.Type == 24)

{

postfix.Add(type);

stack.Push(lexem);

}

break;

}

}

else

{

stack.Push(lexem);

}

break;

default:

switch (lexem.Type)

{

case 7:

postfix.Add(lexem);

break;

case 8:

postfix.Add(lexem);

break;

case 23:

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

switch (type.Type)

{

case 21:

postfix.Add(type);

stack.Push(lexem);

break;

case 14:

stack.Push(type);

stack.Push(lexem);

break;

case 25:

postfix.Add(type);

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

while ((type.Type == 26 || type.Type == 27 || type.Type == 21 || type.Type == 22 || type.Type == 23 || type.Type == 24) && stack.Count != 0)

{

isWhile = true;

postfix.Add(type);

type = (Lexem)stack.Pop();

}

if (stack.Count == 0)

if (type.Type == 26 || type.Type == 27 || type.Type == 21 || type.Type == 22 || type.Type == 23 || type.Type == 24)

postfix.Add(type);

else

stack.Push(type);

if (isWhile)

{

stack.Push(type);

isWhile = false;

}

stack.Push(lexem);

}

else

{

stack.Push(lexem);

}

break;

case 26:

postfix.Add(type);

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

while ((type.Type == 27 || type.Type == 21 || type.Type == 22 || type.Type == 23 || type.Type == 24) && stack.Count != 0)

{

isWhile = true;

postfix.Add(type);

type = (Lexem)stack.Pop();

}

if (stack.Count == 0)

stack.Push(type);

if (isWhile)

{

stack.Push(type);

isWhile = false;

}

stack.Push(lexem);

}

else

{

stack.Push(lexem);

}

break;

case 27:

postfix.Add(type);

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

if (type.Type == 21 || type.Type == 22 || type.Type == 23 || type.Type == 24)

postfix.Add(type);

else

stack.Push(type);

stack.Push(lexem);

}

else

{

stack.Push(lexem);

}

break;

case 22:

postfix.Add(type);

stack.Push(lexem);

break;

default:

if (type.Type == 23 || type.Type == 24)

{

postfix.Add(type);

stack.Push(lexem);

}

break;

}

}

else

{

stack.Push(lexem);

}

break;

case 24:

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

switch (type.Type)

{

case 21:

postfix.Add(type);

stack.Push(lexem);

break;

case 14:

stack.Push(type);

stack.Push(lexem);

break;

case 25:

postfix.Add(type);

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

while ((type.Type == 26 || type.Type == 27 || type.Type == 21 || type.Type == 22 || type.Type == 23 || type.Type == 24) && stack.Count != 0)

{

isWhile = true;

postfix.Add(type);

type = (Lexem)stack.Pop();

}

if (stack.Count == 0)

if (type.Type == 26 || type.Type == 27 || type.Type == 21 || type.Type == 22 || type.Type == 23 || type.Type == 24)

postfix.Add(type);

else

stack.Push(type);

if (isWhile)

{

stack.Push(type);

isWhile = false;

}

stack.Push(lexem);

}

else

{

stack.Push(lexem);

}

break;

case 26:

postfix.Add(type);

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

while ((type.Type == 27 || type.Type == 21 || type.Type == 22 || type.Type == 23 || type.Type == 24) && stack.Count != 0)

{

isWhile = true;

postfix.Add(type);

type = (Lexem)stack.Pop();

}

if (stack.Count == 0)

stack.Push(type);

if (isWhile)

{

stack.Push(type);

isWhile = false;

}

stack.Push(lexem);

}

else

{

stack.Push(lexem);

}

break;

case 27:

postfix.Add(type);

if (stack.Count != 0)

{

type = (Lexem)stack.Pop();

if (type.Type == 21 || type.Type == 22 || type.Type == 23 || type.Type == 24)

postfix.Add(type);

else

stack.Push(type);

stack.Push(lexem);

}

else

{

stack.Push(lexem);

}

break;

case 22:

postfix.Add(type);

stack.Push(lexem);

break;

default:

if (type.Type == 23 || type.Type == 24)

{

postfix.Add(type);

stack.Push(lexem);

}

break;

}

}

else

{

stack.Push(lexem);

}

break;

}

break;

}

}

while (stack.Count != 0)

{

type = (Lexem)stack.Pop();

if (type.Type != 14)

postfix.Add(type);

else

{

err = new Error();

err.Num\_Lex = finish;

err.Error\_Text = "Дизбаланс дужок";

errors.Add(err);

break;

}

}

Generation.GenereteExp\_b(postfix, lexReturn\_b, isWhile);

lexReturn\_b.Id = -1;

}

}

}

using System;

using System.Collections.Generic;

using System.Text;

using System.Collections;

using System.Windows.Forms;

namespace Notepad

{

class Const\_str

{

string name;

string size;

public string Name

{

get { return name; }

set { name = value; }

}

public string Size

{

get { return size; }

set { size = value; }

}

}

class Generation

{

public static string Code = "";

public static int numLabels = 0;

public static int numWhileLbls = 0;

public static int numEndWhileLbls = 0;

public static bool GenVar = false;

public static bool GenDeclar = false;

public static Hashtable const\_int = new Hashtable();

public static Hashtable const\_str = new Hashtable();

public static Queue labels = new Queue();

public static Stack whilelabels = new Stack();

public static Stack EndWhilelabels = new Stack();

public static Const\_str c\_str;

public static void GenerateBegin()

{

Code = Code + ".model small" + "\n";

Code = Code + ".386" + "\n";

Code = Code + ".stack 100h" + "\n\r";

}

public static void GenereteVar()

{

if (!GenVar)

{

Code = Code + ".DATA" + "\n";

Code = Code + "buff db 6,7 Dup(?)" + "\n";

Code = Code + "NewLine db 10,13,'$'\n";

Const\_Strings str\_1;

int i = 1;

foreach (DictionaryEntry str in Parser.const\_str)

{

str\_1 = (Const\_Strings)str.Value;

Code = Code + "str\_" + i.ToString() + " DB " + str\_1.Value + ",\'$\'\n";

Code = Code + "size\_" + i.ToString() + " DW " + "($ - str\_" + i.ToString() + "-1)\n";

c\_str = new Const\_str();

c\_str.Name = "str\_" + i.ToString();

c\_str.Size = "size\_" + i.ToString();

try{

const\_str.Add(str\_1.Value, c\_str);

}

catch { };

i++;

}

GenVar = true;

}

}

public static void GenereteDerlaration()

{

if (!GenDeclar)

{

string str\_1;

int i = 1;

foreach (DictionaryEntry str in Parser.const\_int)

{

str\_1 = (string)str.Value;

Code = Code + "num\_" + i.ToString() + " DW " + str\_1 + "\n";

try

{

const\_int.Add(str\_1, "num\_" + i.ToString());

}

catch { };

i++;

}

Code = Code + "b\_0" + " DB " + "0" + "\n";

Code = Code + "b\_1" + " DB " + "1" + "\n";

Iden\_info id;

string val = "";

foreach (DictionaryEntry de in Parser.identy)

{

id = (Iden\_info)de.Value;

if (id.Value == "")

val = "?";

else

val = id.Value;

if (id.Type == "INTEGER")

Code = Code + "\_" + id.Id + " DW " + val + "\n";

else

Code = Code + "\_" + id.Id + " DB " + val + "\n";

}

Code = Code + "Temp\_1" + " DW " + "?" + "\n";

Code = Code + "Temp\_2" + " DW " + "?" + "\n";

Code = Code + "Temp\_3" + " DW " + "?" + "\n";

Code = Code + "\r";

Code = Code + ".CODE" + "\n";

Code = Code + "Start\_:" + "\n";

Code = Code + ".startup" + "\n";

GenDeclar = true;

}

}

public static void GenereteExp\_i(ArrayList arr, Lexem lexReturn,bool isWrite, bool b)

{

int i=0;

Lexem lexem;

string num="";

if (arr.Count != 1)

{

foreach (Lexem lex in arr)

{

if (lex.Type == 4 || lex.Type == 5)

if (lex.Type == 4)

Code = Code + "fild " + "\_" + lex.Lex + "\n";

else

{

foreach (DictionaryEntry de in const\_int)

{

if ((string)de.Key == lex.Lex)

num = (string)de.Value;

}

Code = Code + "fild " + num + "\n";

}

else

if (lex.Type == 35)

Code = Code + "fldz" + "\n";

switch (lex.Type)

{

case 16:

Code = Code + "fadd" + "\n";

break;

case 17:

Code = Code + "fsub" + "\n";

break;

case 18:

Code = Code + "fmul" + "\n";

break;

case 19:

Code = Code + "fdiv" + "\n";

break;

case 20:

Code = Code + "fistp Temp\_2" + "\n";

Code = Code + "fistp Temp\_3" + "\n";

Code = Code + "fild Temp\_2" + "\n";

Code = Code + "fild Temp\_3" + "\n";

Code = Code + "fprem" + "\n";

break;

}

i++;

}

if (lexReturn.Id != -1)

{

Code = Code + "fistp " + "\_" + lexReturn.Lex + "\n\r";

}

if (isWrite)

{

Code = Code + "fistp Temp\_1" + "\n";

Code = Code + "mov ax,Temp\_1" + "\n";

}

}

else {

lexem = (Lexem)arr[0];

if (lexReturn.Id != -1)

{

if (lexem.Type == 4)

{

Code = Code + "mov ax," + "\_" + lexem.Lex + "\n";

Code = Code + "mov " + "\_" + lexReturn.Lex + ", ax \n\r";

}

if(lexem.Type == 5)

Code = Code + "mov " + "\_" + lexReturn.Lex + ", " + lexem.Lex + "\n\r";

}

if (isWrite)

{

if (lexem.Type == 4)

Code = Code + "mov ax," + "\_" + lexem.Lex + "\n";

if(lexem.Type == 5)

Code = Code + "mov ax," + lexem.Lex + "\n";

}

if (b)

Code = Code + "fild " + "\_" + lexReturn.Lex + " \r";

}

}

public static void GenereteExp\_b(ArrayList arr, Lexem lexReturn, bool isWhile)

{

Lexem lex,lex1;

bool isEax = false;

bool isAl = false;

bool isDl = false;

bool TypeVar;

Iden\_info id;

if (arr.Count != 1)

{

Code = Code + "xor ax,ax \n";

Code = Code + "xor dx,dx \n";

for (int i = 0; i < arr.Count; i++)

{

try

{

lex = (Lexem)arr[i];

switch (lex.Type)

{

case 4:

id = (Iden\_info)Parser.identy[lex.Lex];

TypeVar = (id.Type == "INTEGER") ? true : false;

if (TypeVar)

if (!isEax)

{

Code = Code + "mov ax," + "\_" + id.Id + "\n";

isEax = true;

}

else

{

Code = Code + "mov dx," + "\_" + id.Id + "\n";

}

else

if (!isAl)

{

Code = Code + "mov al," + "\_" + id.Id + "\n";

isAl = true;

}

else

{

Code = Code + "mov dl," + "\_" + id.Id + "\n";

isDl = true;

}

break;

case 5:

if (!isEax)

{

Code = Code + "mov ax," + (string)const\_int[lex.Lex] + "\n";

isEax = true;

}

else

{

Code = Code + "mov dx," + (string)const\_int[lex.Lex] + "\n";

}

break;

case 21:

Code = Code + "cmp ax,dx" + "\n";

Code = Code + "jne Label\_" + numLabels.ToString() + "\n";

labels.Enqueue("Label\_" + numLabels.ToString());

numLabels++;

Code = Code + "mov ax,1" + "\n";

Code = Code + "jmp Label\_" + numLabels.ToString() + "\n";

labels.Enqueue("Label\_" + numLabels.ToString());

numLabels++;

Code = Code + (string)labels.Dequeue() + ": mov ax,0" + "\n";

Code = Code + (string)labels.Dequeue() + ": push ax" + "\n";

isEax = false;

break;

case 25:

lex1 = (Lexem)arr[i - 1];

switch (lex1.Type)

{

case 4:

if (isDl)

{

id = (Iden\_info)Parser.identy[lex1];

if (id.Value == "1")

Code = Code + "mov dl,b\_0 \n";

else

Code = Code + "mov dl,b\_1 \n";

}

else

{

id = (Iden\_info)Parser.identy[lex1.Lex];

if (id.Value == "1")

Code = Code + "mov al,b\_0 \n";

else

Code = Code + "mov al,b\_1 \n";

}

break;

default:

break;

}

break;

case 26:

if (isAl && isDl)

{

Code = Code + "and al,dl \n";

Code = Code + "push ax" + "\n";

isAl = false;

isDl = false;

}

else

{

if (isAl)

{

Code = Code + "pop dx" + "\n";

Code = Code + "and al,dl \n";

Code = Code + "push ax" + "\n";

}

else

{

Code = Code + "pop dx" + "\n";

Code = Code + "pop ax" + "\n";

Code = Code + "and al,dl \n";

Code = Code + "push ax" + "\n";

}

}

break;

case 27:

if (isAl && isDl)

{

Code = Code + "or al,dl \n";

Code = Code + "push ax" + "\n";

isAl = false;

isDl = false;

}

else

{

if (isAl)

{

Code = Code + "pop dx" + "\n";

Code = Code + "or al,dl \n";

Code = Code + "push ax" + "\n";

}

else

{

Code = Code + "pop dx" + "\n";

Code = Code + "pop ax" + "\n";

Code = Code + "or al,dl \n";

Code = Code + "push ax" + "\n";

}

}

break;

case 22:

Code = Code + "cmp ax,dx" + "\n";

Code = Code + "je Label\_" + numLabels.ToString() + "\n";

labels.Enqueue("Label\_" + numLabels.ToString());

numLabels++;

Code = Code + "mov ax,1" + "\n";

Code = Code + "jmp Label\_" + numLabels.ToString() + "\n";

labels.Enqueue("Label\_" + numLabels.ToString());

numLabels++;

Code = Code + (string)labels.Dequeue() + ": mov ax,0" + "\n";

Code = Code + (string)labels.Dequeue() + ": push ax" + "\n";

isEax = false;

break;

default:

switch (lex.Type)

{

case 7:

if (!isAl)

{

Code = Code + "mov al," + "\_" + lex.Lex + "\n";

isAl = true;

}

else

{

Code = Code + "mov dl," + "\_" + lex.Lex + "\n";

isDl = true;

}

break;

case 8:

if (!isAl)

{

Code = Code + "mov al," + "\_" + lex.Lex + "\n";

isAl = true;

}

else

{

Code = Code + "mov dl," + "\_" + lex.Lex + "\n";

isDl = true;

}

break;

case 23:

Code = Code + "cmp ax,dx" + "\n";

Code = Code + "jng Label\_" + numLabels.ToString() + "\n";

labels.Enqueue("Label\_" + numLabels.ToString());

numLabels++;

Code = Code + "mov ax,1" + "\n";

Code = Code + "jmp Label\_" + numLabels.ToString() + "\n";

labels.Enqueue("Label\_" + numLabels.ToString());

numLabels++;

Code = Code + (string)labels.Dequeue() + ": mov ax,0" + "\n";

Code = Code + (string)labels.Dequeue() + ": push ax" + "\n";

break;

isEax = false;

case 24:

Code = Code + "cmp ax,dx" + "\n";

Code = Code + "jnl Label\_" + numLabels.ToString() + "\n";

labels.Enqueue("Label\_" + numLabels.ToString());

numLabels++;

Code = Code + "mov ax,1" + "\n";

Code = Code + "jmp Label\_" + numLabels.ToString() + "\n";

labels.Enqueue("Label\_" + numLabels.ToString());

numLabels++;

Code = Code + (string)labels.Dequeue() + ": mov ax,0" + "\n";

Code = Code + (string)labels.Dequeue() + ": push ax" + "\n";

isEax = false;

break;

}

break;

}

}

catch { };

}

if (lexReturn.Id != -1)

{

Code = Code + "pop ax" + "\n";

Code = Code + "mov " + "\_" + lexReturn.Lex + ",al" + "\n\r";

}

}

else {

if (lexReturn.Id != -1)

{

lex = (Lexem)arr[0];

if (lex.Type != 7 && lex.Type != 8)

{

Code = Code + "mov al," + "\_" + lex.Lex + "\n";

}

else

{

if (lex.Type == 7)

Code = Code + "mov al,b\_1" + "\n";

else

Code = Code + "mov al,b\_0" + "\n";

}

Code = Code + "mov " + "\_" + lexReturn.Lex + ",al" + "\n\r";

}

else

{

lex = (Lexem)arr[0];

if (lex.Type != 7 && lex.Type != 8)

{

Code = Code + "mov al," + "\_" + lex.Lex + "\n";

}

else

{

if (lex.Type == 7)

Code = Code + "mov al,b\_1" + "\n";

else

Code = Code + "mov al,b\_0" + "\n";

}

Code = Code + "push ax" + "\n";

}

}

}

public static void GenerateReadVar(string str)

{

Code = Code + "call InputInt" + "\n";

Code = Code + "mov " + "\_" + str + ",ax\n\r";

}

public static void GenerateStr(string str)

{

Const\_str cs = (Const\_str)const\_str[str];

Code = Code + "mov ax, offset "+ cs.Name + "\n";

Code = Code + "mov dx," + cs.Size + "\n";

}

public static void GenerateWrite(bool type)

{

if (type)

{

Code = Code + "call OutInt" + "\n\r";

Code = Code + "mov dx,offset NewLine \n" + "mov ah,09\n" + "int 21h\n";

}

else

{

Code = Code + "call OutStr" + "\n\r";

}

}

public static void GenereteInicialCycle(Lexem lx)

{

Code = Code + "fild " + (string)const\_int[lx.Lex] + "\n";

Code = Code + "fsub \n";

Code = Code + "fistp Temp\_1 \n";

Code = Code + "mov cx,Temp\_1 \n";

}

public static void RepeatStart()

{

Code = Code + "Next\_" + numWhileLbls.ToString() + ":\n";

whilelabels.Push("Next\_" + numWhileLbls.ToString());

numWhileLbls++;

Code = Code + "push cx \n";

}

public static void GenerateUntil()

{

Code = Code + "pop cx \n";

Code = Code + "loop " + (string)whilelabels.Pop() + "\n";

}

public static void GenerateEndProgram()

{

Code = Code + "mov ah, 7" + "\n";

Code = Code + "int 21h" + "\n";

Code = Code + "ret" + "\n";

Code = Code + "OutInt proc \n" +

" test ax, ax\n" +

" jns oi1\n" +

" mov cx, ax\n" +

" mov ah, 02h\n" +

" mov dl, '-'\n" +

" int 21h\n" +

" mov ax, cx\n" +

" neg ax\n" +

"oi1: \n" +

" xor cx, cx\n" +

" mov bx, 10 \n" +

"oi2:\n" +

" xor dx,dx\n" +

" div bx\n" +

" push dx\n" +

" inc cx\n" +

" test ax, ax\n" +

" jnz oi2\n" +

" mov ah, 02h \n" +

"oi3:\n" +

" pop dx\n" +

" add dl, '0'\n" +

" int 21h\n" +

" loop oi3 \n" +

" ret\n" +

"OutInt endp \n" +

"InputInt proc \n" +

" mov ah,0ah \n" +

" xor di,di \n" +

" mov dx,offset buff \n" +

" int 21h \n" +

" mov dl,0ah\n" +

" mov ah,02\n" +

" int 21h \n" +

" mov si,offset buff+2 \n" +

" cmp byte ptr [si],\"-\" \n" +

" jnz ii1\n" +

" mov di,1 \n" +

" inc si \n" +

"ii1:\n" +

" xor ax,ax\n" +

" mov bx,10 \n" +

"ii2:\n" +

" mov cl,[si] \n" +

" cmp cl,0dh \n" +

" jz endin\n" +

" cmp cl,'0' \n" +

" jl er\n" +

" cmp cl,'9' \n" +

" ja er\n" +

" sub cl,'0' \n" +

" mul bx \n" +

" add ax,cx \n" +

" inc si \n" +

" jmp ii2 \n" +

"er: \n" +

" mov dx, offset error\n" +

" mov ah,09\n" +

" int 21h\n" +

" int 20h\n" +

"endin:\n" +

" cmp di,1 \n" +

" jnz ii3\n" +

" neg ax \n" +

"ii3:\n" +

" ret\n" +

"error db \"incorrect number$\" \n" +

"InputInt endp\n" +

"OutStr proc \n" +

"mov dx,ax \n" +

"mov ah,09\n" +

"int 21h\nret\n" +

"OutStr endp\n";

Code = Code + "end Start\_" + "\n\r";

}

}

}