

## ЗМІСТ

Вступ.....	5
1 Граматика мови .....	6
1.1 Опис розробленої мови .....	6
1.2 Приклад коду програми .....	6
1.3 Граматика мови.....	6
2 Структура транслятора .....	8
2.1 Структура транслятора.....	8
3 Лексичний аналізатор.....	9
3.1 Таблиця лексем .....	12
3.2 Приклад роботи лексичного аналізатора .....	12
4 Синтаксичний аналізатор .....	15
4.1 Опис синтаксичного аналізатора.....	15
4.2 Діаграма станів синтаксичного аналізатора.....	15
4.3 Приклад роботи лексичного аналізатора .....	20
5 Генерація та виконання проміжної форми представлення .....	23
5.1 Опис генератору коду.....	23
5.1 Реалізація побудови полізу .....	24
6 Модуль виконання полізу.....	28
7 Опис проекту.....	31
7.1 Інтерфейс програми.....	32
Висновки .....	33
Список використаних джерел .....	34
Додаток.....	35

					УКР.НТУУ «КПІ ім. І. Сікорського»_ ТЕФ_АПЕПС_ТР-52			
Змн.	Лист	№ докум.	Підпис	Дата				
Розроб.	Софієнко А.Ю.						Літ.	Арк.
Перевір.	Третяк В.А.							Аркушів
								4
								39
Н. Контр.					<b>НТУУ «КПІ</b> <b>ім. Ігоря Сікорського»</b>			
Затверд.								

## ВСТУП

Мова програмування являє собою засіб опису обчислень для людей та машин. Сучасний світ залежить від мов програмування, оскільки все програмне забезпечення на всіх комп'ютерах світу написане на тій чи іншій мові програмування. Проблема в тому, що комп'ютери не розуміють звичні для людей мови. Тому перш ніж запустити програму на виконання, необхідно перетворити в форму, яку зможе зрозуміти та виконати комп'ютер.

Програмні системи, які виконують таке перетворення, називаються компіляторами. Розуміння принципів та методів проектування трансляторів можуть знадобитися при розробці систем, що вирішують широкий спектр задач в різних сферах.

						Лист
						5
Изм	Лист	№ докум	Подп	Дата		

# 1 ГРАМАТИКА МОВИ

## 1.1 Опис розробленої мови

Згідно із завданням розроблена мова повинна містити:

**Цикл:** `do while (<логічний вираз>) <список операторів> enddo`

**Умовний перехід:** `if(<логічний вираз>) then <список операторів> fi`

**Особливості арифметичного виразу:** `+*/()`, константи з фіксованою точкою

**Додатковий оператор:** тернарний оператор `(<ід> = <логічний вираз> ? <вираз> : <вираз>)`

**Роздільник:** ¶

## 1.2 Приклад коду програми

```
program test
begin
float a=0
int b =0
int c=0
read(a,b,c)
do while(a<c)
    write(a)
    if(a>b)then
        float temp =a+1
        do while(a<=temp)
            a=a+0.1
            write(a)
        enddo
    fi
    a=a+1
enddo
c = (c==a)?333:-999
write(c)
end
```

## 1.3 Граматика мови

**<програма> ::=** `program <ім'я програми> <перехід на нову строку> begin <перехід на нову строку> <список операторів> end`

**<перехід на нову строку> ::=** ¶ | <перехід на нову строку> ¶

						Лист
						6
Изм	Лист	№ докум	Подп	Дата		

**<список операторів> ::= <оператор> <перехід на нову строку> | <список операторів> <оператор> <перехід на нову строку>**

**<оператор> ::= <присвоїти> | <ввід> | <вивід> | <цикл> | <умовний перехід> | <тернарний оператор> | <оголошення нової змінної>**

**<присвоїти> ::= <індентифікатор> = <вираз>**

**<ввід> ::= read( <список ідентифікаторів> )**

**<вивід> ::= write( <список ідентифікаторів> )**

**<список ідентифікаторів> ::= <ідентифікатор> | <список ідентифікаторів>, <ідентифікатор>**

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

**<умовний перехід> ::= if( <логічний вираз> ) then <список операторів> fi**

**<логічний вираз> ::= <логічний терм> | <логічний вираз> or <логічний терм>**

**<логічний терм> ::= <логічний множник> | <логічний терм> and <логічний множник>**

**<логічний множник> ::= <відношення> | not <логічний множник> | [ <логічний терм> ]**

**<відношення> ::= <вираз> <знак відношення> <вираз>**

**<знак відношення> ::= != | <= | >= | < | > | ==**

**<тернарний оператор> ::= <ідентифікатор> = <логічний вираз> ? <вираз> : <вираз>**

**<вираз> ::= <терм> | <вираз> + <терм> | <вираз> - <терм>**

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

**<множина> ::= ( <вираз> ) | <ідентифікатор> | <літеральна константа>**

**<оголошення нової змінної> ::= <тип> <присвоїти>**

**<ідентифікатор> ::= <буква> | <ідентифікатор> <буква> | <ідентифікатор> <цифра>**

**<літеральна константа> ::= <ціле без знаку> | <дробове без знаку> | <ціле> | <дробове>**

**<тип> ::= int | float**

**<дробове> ::= -<дробове без знаку> | <дробове без знаку> | +<дробове без знаку>**

**<дробове без знаку> ::= <ціле без знаку> | .<ціле без знаку> | <ціле без знаку> .<ціле без знаку>**

**<ціле> ::= +<ціле без знаку> | -<ціле без знаку> | <ціле без знаку>**

**<ціле без знаку> ::= <ціле без знаку> <цифра> | <цифра>**

**<буква> ::= a | ... | z | A .. Z**

**<цифра> ::= 0 | ... 9**

						Лист
						7
Изм	Лист	№ докум	Подп	Дата		

## 2 СТРУКТУРА ТРАНСЛЯТОРА

### 2.1 Структура транслятора

Розроблений транслятор є трьох прохідним.

Тому його перевагами є незалежність кожної фази трансляції, що дає йому гнучкість і можливість зміни кожної фази, а також потребує невеликої кількості оперативної пам'яті.

Транслятор складається з трьох основних блоків: лексичного аналізатора, синтаксичного аналізатора, та генератора коду, та має наступну структуру:



Спочатку блок лексичного аналізатора зчитує початкову програму, проводить її лексичний аналіз і генерує ланцюжок лексем, який потрапляє в синтаксичний аналізатор. Синтаксичний аналізатор перевіряє порядок слідування лексем і визначає чи відповідає він граматиці. Потім ланцюжок лексем потрапляє в Генератор коду, який генерує на основі лексем код необхідного нам вигляду [1].

Розроблена мова програмування є типізованою та регістрозалежною.

### 3 ЛЕКСТЧНИЙ АНАЛІЗАТОР

Основна задача лексичного аналізатора – розбір вхідного рядка на лексичні одиниці.

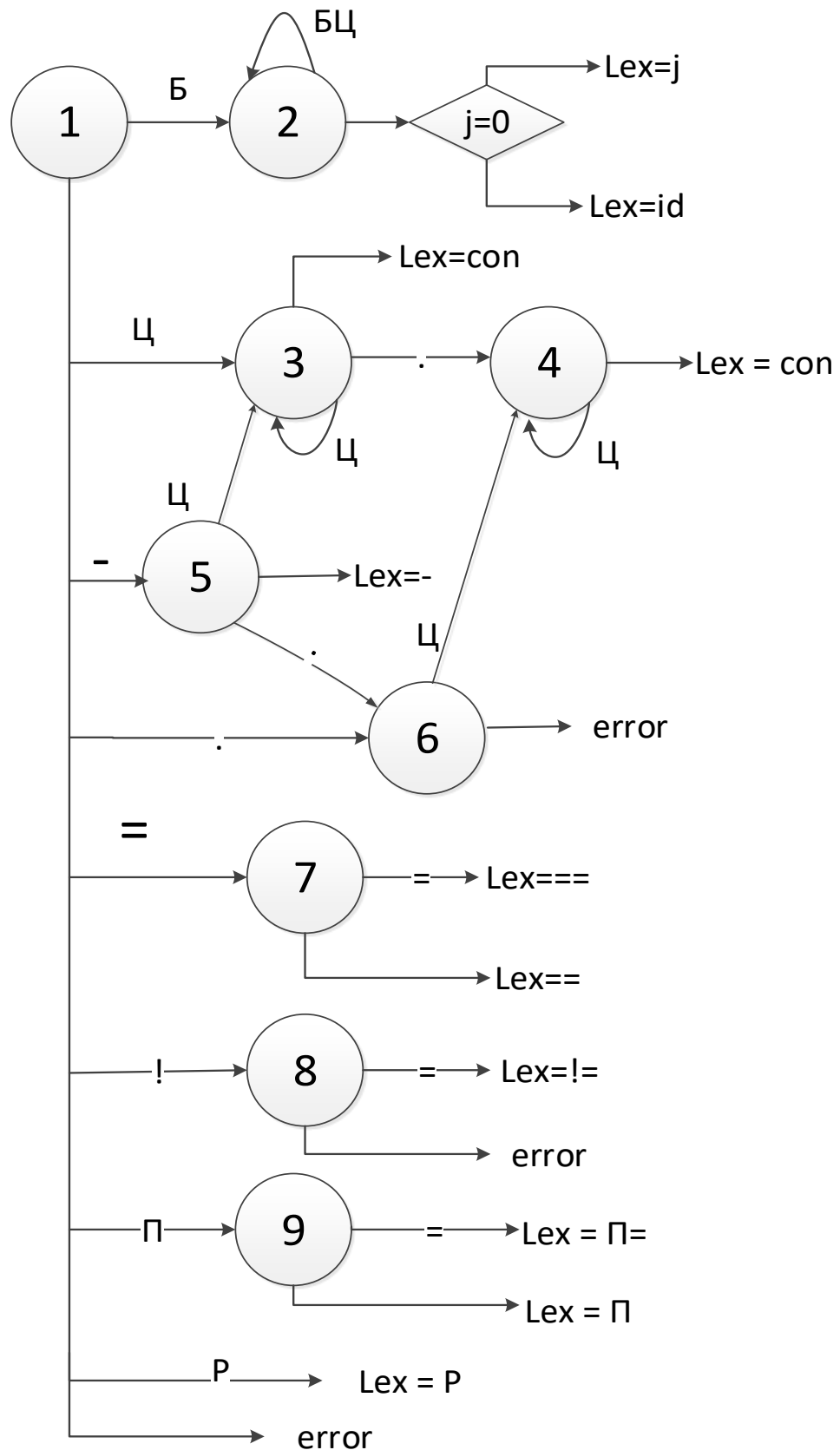
Лексична одиниця – це підрядок вхідного рядка. Вона може містити лише термінальні символи і не може містити інших лексем. Для синтаксичного аналізу лексема є найменшою одиницею мови, а з термінальними символами працює виключно лексичний аналізатор – сканер.

Згідно з варіантом курсової роботи лексичний аналізатор має бути реалізований методом скінченного автомата. На основі завдання було створено таблицю переходів (таблиця 3.1) та діаграму станів (діаграма 3.1) скінченного автомата лексичного аналізатора.

$\alpha$	Мітка переходу	$\beta$	Семантична підпрограма
1	Б	2	[ $\neq$ ] помилка
	Ц	3	
	-	5	
	.	6	
	=	7	
	!	8	
	П	9	
	Р		[ $=$ ] виділення лексеми Р
2	Б	2	[ $\neq$ ] lex=j або lex=id
	Ц	2	
3	Ц	3	[ $\neq$ ] lex=con
	.	4	
4	Ц	4	[ $\neq$ ] lex=con
5	Ц	3	[ $\neq$ ] помилка
	.	6	

$\alpha$	Мітка переходу	$\beta$	Семантична підпрограма
6	Ц	4	[ $\neq$ ] помилка
7	=		[ $=$ ] lex= == [ $\neq$ ] lex= =
8	=		[ $=$ ] lex= != [ $\neq$ ] помилка
9	=		[ $=$ ] lex= П= [ $\neq$ ] lex= П

Таблиця 3.1 – Таблиця переходів скінченного автомата лексичного аналізатора.



Діаграма 3.1 – Діаграма станів скінченного автомата лексичного аналізатора.



### 3.1 Таблиця лексем

№	Лексема	№	Лексема
1	Program	19	>
2	begin	20	==
3	end	21	?
4	¶	22	+
5	read	23	-
6	write	24	*
7	do	25	/
8	while	26	(
9	if	27	)
10	then	28	[
11	fi	29	]
12	or	30	:
13	and	31	int
14	not	32	float
15	!=	33	enddo
16	<=	34	idn
17	>=	35	,
18	<	36	=

Таблиця 3.1.1 - Лексеми мови

### 3.2 Приклад роботи лексичного аналізатора

Окрім того, що сканер розбиває вхідний текст програми, він його частково перевіряє. На етапі лексичного аналізу в тексті прогами можуть бути виявлені

символи, яких немає в граматиці мови (Рисунок 3.2) та неоголошені змінні (Рисунок 3.3)

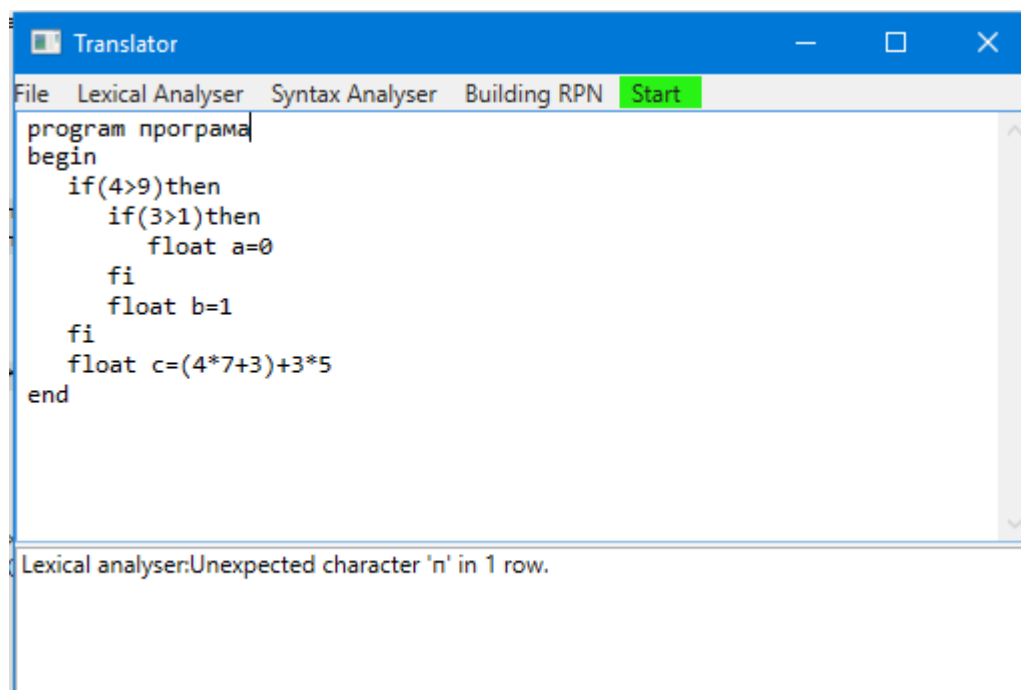


Рисунок 3.2 – Приклад неуспішного лексичного аналізу. Символ «п» відсутній у граматиці мови.

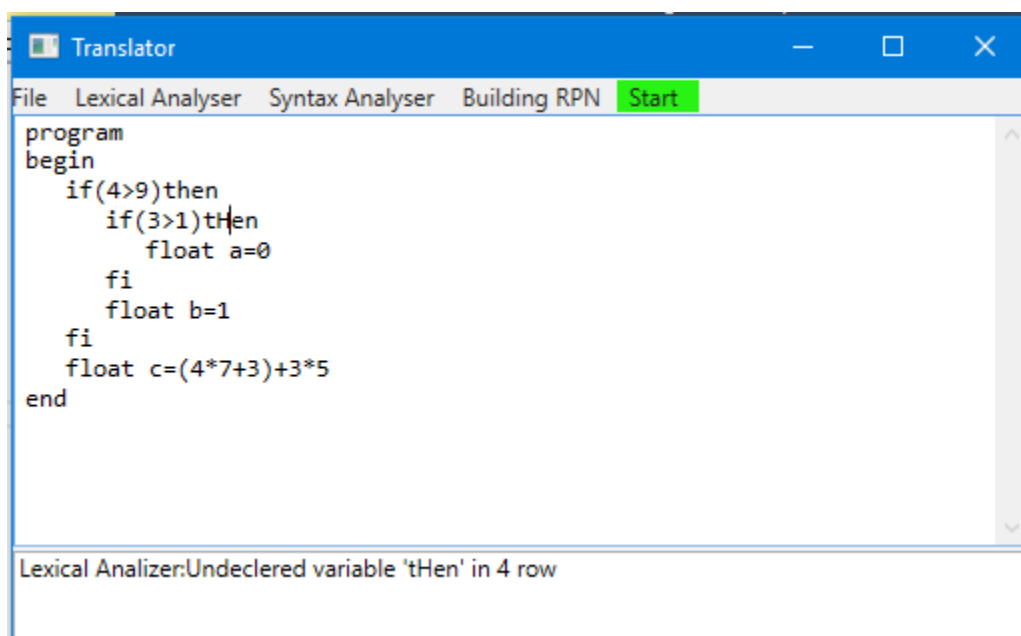


Рисунок 3.3 – Приклад неуспішного лексичного аналізу. Змінна «tHen» не була оголошена.

LexemTable							
Lexem list				Identifier list			
Row	Substring	TerminalCode		Type	Value	Name	
1	program	Pogram		Pogram	0	test2	
1	test2	Identifier		Float	-14.2	startValue	
1	¶	NewLine		Int	-44	buffer	
2	begin	Begin					
2	¶	NewLine					
3	float	Float					
3	startValue	Identifier					
3	=	Assign					
3	3	Constant					
3	*	Multiple					
3	(	OpenBracket					
3	5	Constant					
3	-	Minus					
3	2	Constant					
3	*	Multiple					
3	4,2	Constant					
3	)	CloseBracket					
3	-	Minus					
3	4	Constant					
3	¶	NewLine					
4	int	Int					
4	buffer	Identifier					
4	=	Assign					
4	startValue	Identifier					
4	*	Multiple					

Рисунок 3.4 – Службове вікно. Дані отримані після лексичного аналізу.

# 4 СИНТАКСИЧНИЙ АНАЛІЗАТОР

## 4.1 Опис синтаксичного аналізатора

Мета синтаксичного розбору – визначення у вхідному тексті мовних конструкцій, що описуються граматикою. На вході синтаксичний аналізатор отримує побудовані лексичним аналізатором вихідні таблиці лексем. В процесі своєї роботи синтаксичний аналізатор будує дерево виводу для отриманої послідовності лексем, на цьому базуються основні методи синтаксичного розбору. На виході синтаксичний аналізатор повинен видати вердикт, чи є отримана послідовність лексем правильною для даної мови.

Згідно з умовою, в розробленому трансляторі використовується метод перевірки на основі скінченного автомата. Але алгоритму скінченного автомата недостатньо для розбору мови, що описується контекстно-незалежною граматикою, що містить рекурсії з само вставленням, наприклад конструкції з дужок. Для реалізації таких мов потрібен автомат з магазинною пам'яттю, що і використовується в даному трансляторі [1].

## 4.2 Діаграма станів синтаксичного аналізатора.

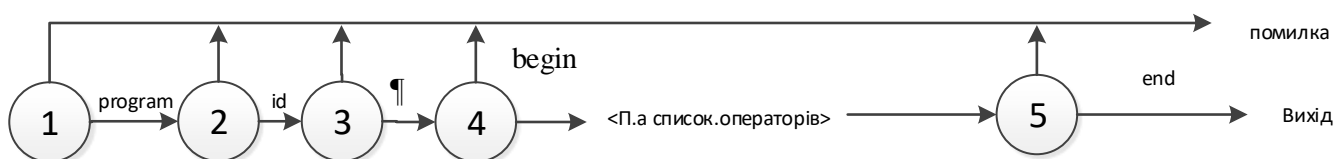


Рисунок 4.2.1 - діаграма станів головного автомату.

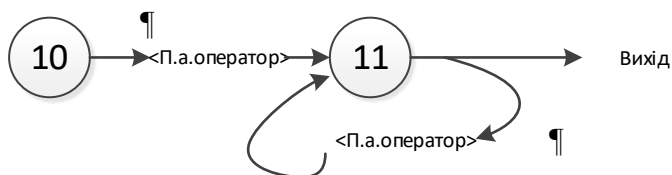
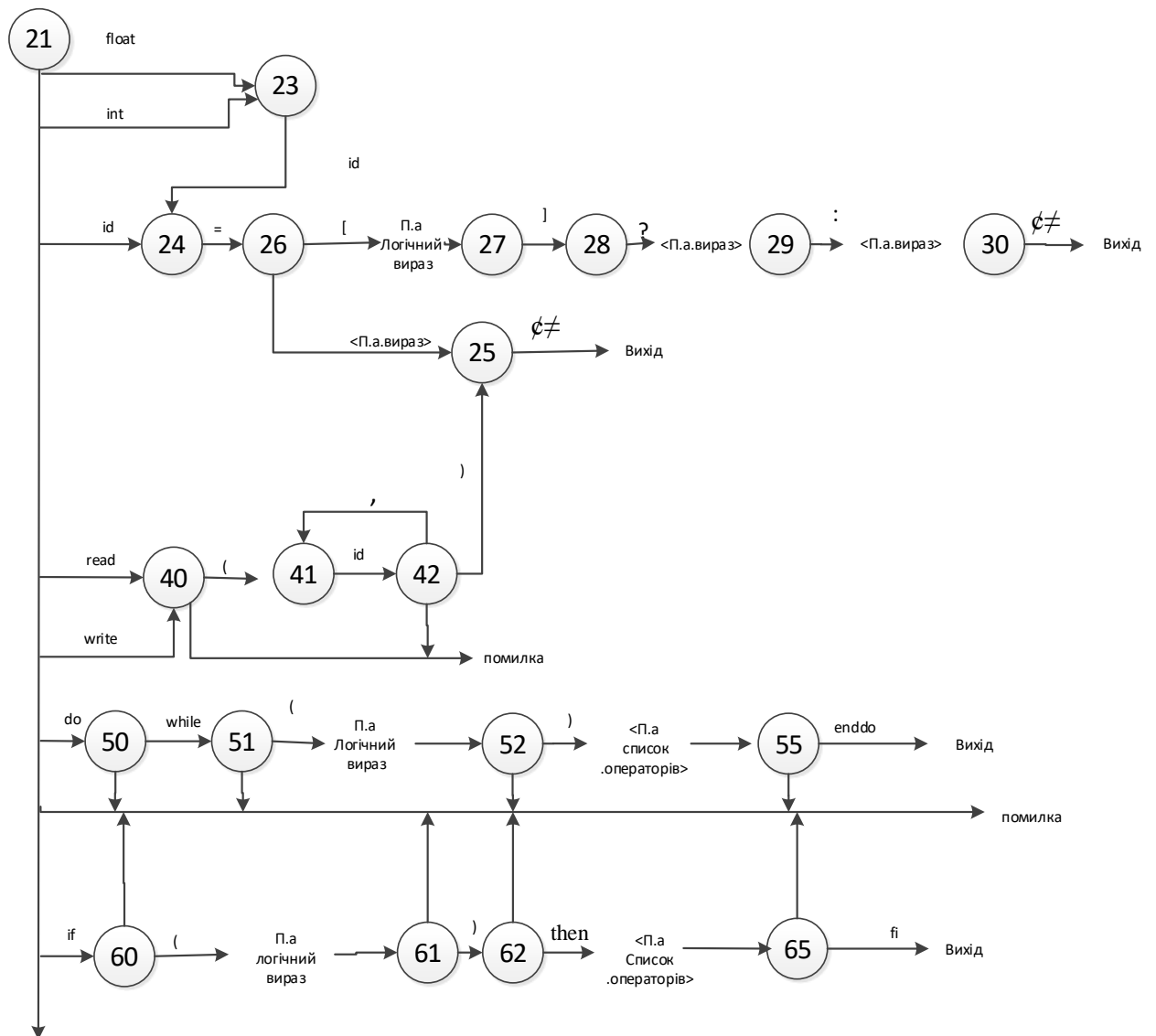


Рисунок 4.2.2 Діаграма підавтомату «список операторів»



4.2.3 Діаграма станів підавтомата «оператор».

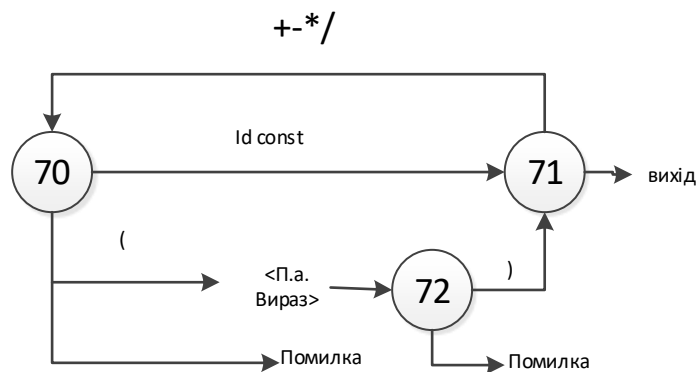


Рисунок - 4.2.4 Діаграма станів підавтомата «арифметичний вираз»

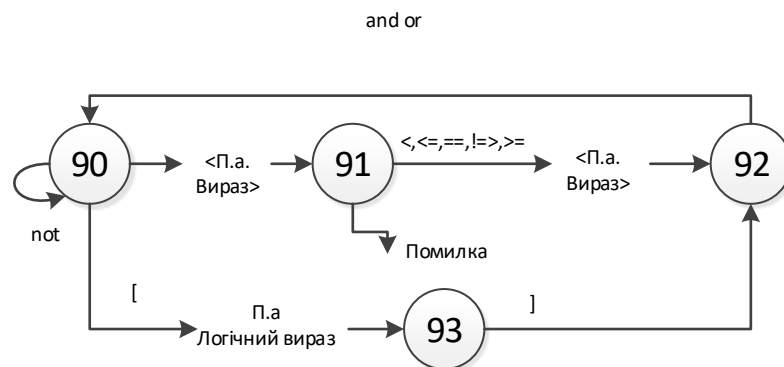


Рисунок 4.2.5 - Діаграма станів підавтомата «логічний вираз»

$\alpha$	Мітка переходу	В	Стек	Семантична підпрограма
1	program	2		[≠]помилка
2	id	3		[≠]помилка
3	¶	4		[≠]помилка
4	begin	< п/а список операторів>	↓5	[≠]помилка
5	end			[=] вихід, [≠]помилка

Рисунок 4.2.1 - Діаграма станів головного автомату.

$\alpha$	Мітка переходу	В	Стек	Семантична підпрограма
10	¶	< п/а оператор>	↓11	[≠]помилка
11	¶	< п/а оператор>	↓11	[≠]вихід

Рисунок 4.2.2 - Діаграма підавтомата «списку операторів»

<b>α</b>	<b>Мітка переходу</b>	<b>В</b>	<b>Стек</b>	<b>Семантична підпрограма</b>
21	int float id read write do if	23 23 24 40 40 50 60		[≠]помилка
23	id	24		[≠]помилка
24	=	26		[≠]помилка
25	ϕ≠			[≠]вихід
26	[	< п/а логічний вираз>	↓27	[≠]< п/а вираз>↓25
27	]	28		[≠]помилка
28	?	< п/а вираз>	↓29	[≠]помилка
29	:	< п/а вираз>	↓30	[≠]помилка
30	ϕ≠			[≠]вихід
40	(	41		[≠]помилка
41	id	42		[≠]помилка
42	) ,	25 41		[≠]помилка
50	while	51		[≠]помилка
51	(	< п/а логічний вираз>	↓52	[≠]помилка
52	)	< п/а список операторів>	↓55	[≠]помилка
55	enddo			[=] вихід
60	(	< п/а логічний вираз>	↓61	[≠]помилка

<b>α</b>	<b>Мітка переходу</b>	<b>В</b>	<b>Стек</b>	<b>Семантична підпрограма</b>
61	)	62		[≠]помилка
62	then	< п/а список операторів >	↓65	[≠]помилка
65	fi			[=] вихід

Таблиця 4.2.3 - Таблиця переходів підавтомата «оператор».

<b>α</b>	<b>Мітка переходу</b>	<b>β</b>	<b>Стек</b>	<b>Семантична підпрограма</b>
70	id	71		[≠]помилка
	const	71		
	(	<п/а вираз>	↓72	
72	)	71		[≠]помилка
71	+	70		[≠]вихід
	-	70		
	*	70		
	/	70		

Таблиця 4.2.4 - Таблиця переходів підавтомату «вираз»

<b>α</b>	<b>Мітка переходу</b>	<b>β</b>	<b>Стек</b>	<b>Семантична підпрограма</b>
90	not	90		[≠]п/а вир ↓91
	[	< п/а логічний вираз >	↓93	
91	<	<п/а вираз>	↓91	[≠]помилка
	≤	<п/а вираз>	↓91	
	==	<п/а вираз>	↓91	
	≠	<п/а вираз>	↓91	
	>	<п/а вираз>	↓91	



$\alpha$	Мітка переходу	$\beta$	Стек	Семантична підпрограма
	$\geq$	<п/а вираз>	↓91	
92	and	90		[≠] вихід
	or	90		
93	]	92		[≠] помилка

Таблиця 4.2.5 - Таблиця переходів МПА підавтомата «логічний вираз»

### 4.3 Приклад роботи лексичного аналізатора

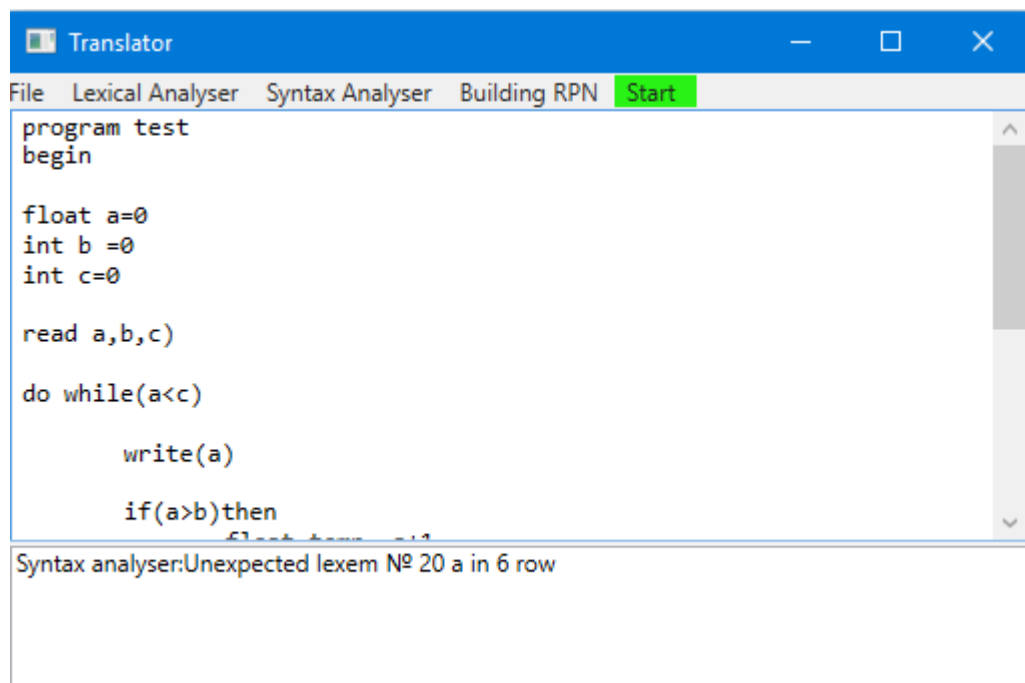


Рисунок 4.3.1 – Приклад неуспішного синтаксичного аналізу. Була пропущена відкриваюча дужка.

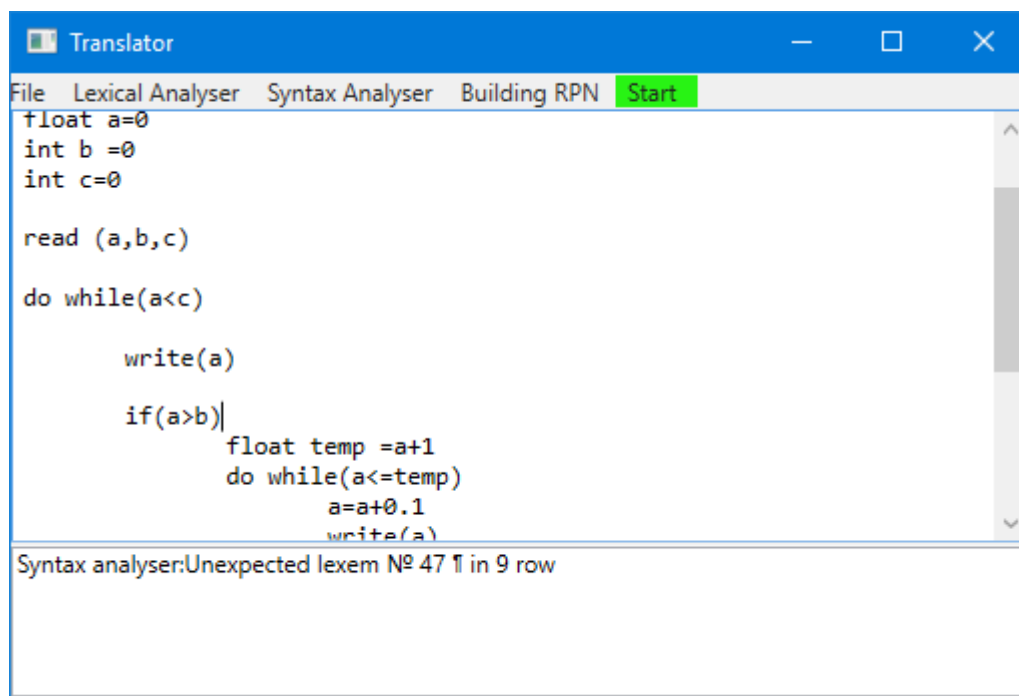


Рисунок 4.3.2 – Приклад неуспішного синтаксичного аналізу. Була пропущена лексема «then» відкриваюча дужка.

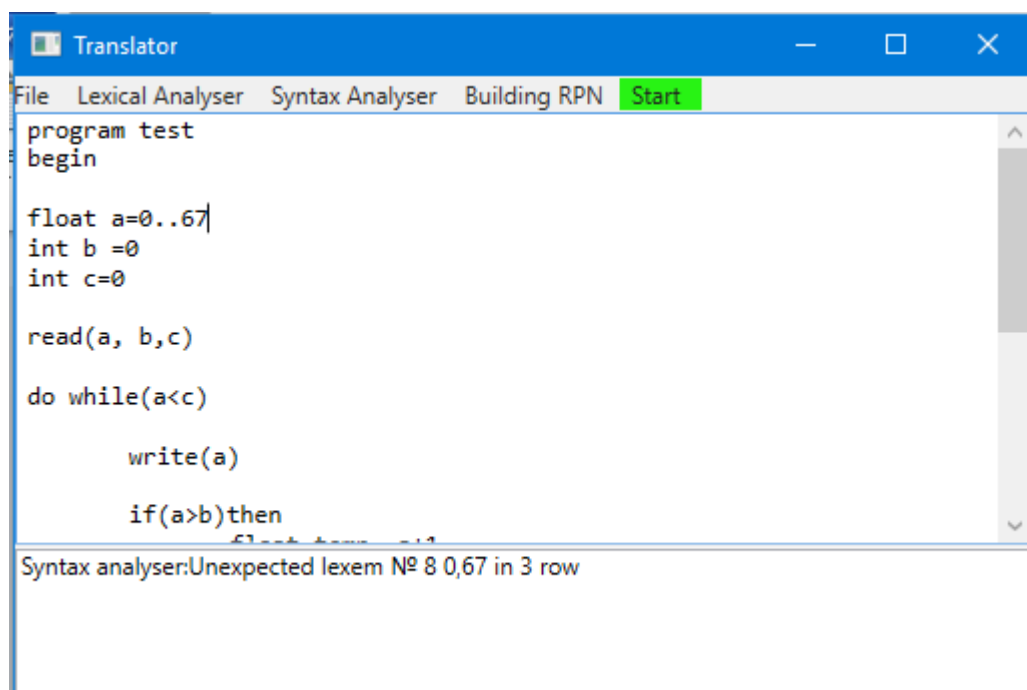


Рисунок 4.3.3 – Приклад неуспішного синтаксичного аналізу. Лексичний аналізатором було розпізнано дві лексеми «0» та «0.67» між якими відсутній оператор.

Number	InputLexem	NumberState	StackString	
1	program	1		
2	test2	2		
3	↑	3		
4	begin	4		
5	↑	10	5	
6	float	21	11 5	
7	startValue	23	11 5	
8	=	24	11 5	
9	3	26	11 5	
10	3	70	25 11 5	
11	*	71	25 11 5	
12	(	70	25 11 5	
13	5	70	72 25 11 5	
14	-	71	72 25 11 5	
15	2	70	72 25 11 5	
16	*	71	72 25 11 5	
17	4,2	70	72 25 11 5	
18	)	71	72 25 11 5	
19	)	72	25 11 5	
20	-	71	25 11 5	
21	4	70	25 11 5	
22	↑	71	25 11 5	
23	↑	25	11 5	
24	↑	11	5	
25	int	21	11 5	
26	buffer	23	11 5	
27	=	24	11 5	
28	startValue	26	11 5	
29	startValue	70	25 11 5	
30	*	71	25 11 5	

Рисунок 4.3.4 – Службове вікно. Таблиця переходів синтаксичного аналізатора.

У випадку, коли синтаксичний аналізатор не виявив помилок, запускається модуль побудови проміжних форм.

## 5 ГЕНЕРАЦІЯ ТА ВИКОНАННЯ ПРОМІЖНОЇ ФОРМИ ПЕРЕДСТАВЛЕННЯ

### 5.1 Опис генератору коду

Генератор коду виконує основну функцію транслятора – перетворює початковий код користувача в інший формат команд. Даний транслятор перетворює код, розроблений мовою у польський інверсний запис (полізу). Такий формат запису характеризується тим, що символи операцій розміщуються після операндів у порядку виконання.

Генератор коду даного транслятора викликається після завершення роботи лексичного та синтаксичного аналізаторів та приймає на вхід таблицю вхідних лексем, сформовану лексичним аналізатором, а на вихід повертає символи у форматі польського запису упорядковані за алгоритмом сортувальної станції Ейдсгера Дейкстри [2].

Генерація полізу здійснюється на основі таблиці пріоритетів (Таблиця 5.1) та стеку.

1	if, while,	7	and
2	(	8	not
3	), ¶, then, ?, :	9	>, >=, <, <=, ==, !=
4	=	10	+, -
5	[	11	*, /
6	] or		

Таблиця 5.1 - Таблиця пріоритетів

Для побудови полізу потрібно перетворити деякі службові слова на конструкції, що можна помістити в полізу та які простіше сприйняти комп'ютеру при ви-

конанні або ж які просто перетворити на асемблер. Для цього було переписано оператори циклу, умовного переходу, тернарний оператор, оператори вводу та виводу на конструкції, на основі міток та логічних виразів, які будуть слугувати операндами для полізу та операторами умовного та безумовного переходів, які аналогічні асемблерним командам JNE та JMP відповідно [2].

## 5.1 Реалізація побудови полізу

В модулі побудови відбувається перебір всіх лексем.

Коли поточна лексема дорівнює оператору while, генератор міток повертає мітку, що вказує на поточну лексему, вона поміщається у вихідний список, там в контейнер разом з позначкою while. Далі він поміщається до стеку.

Коли при переборі була виявлена закриваюча дужка, та на вершині стека знаходиться контейнер з позначкою while, логічний вираз для циклу вже поміщений у вихідний список. Тому відбувається генерація нової мітки, що не містить ніякої інформації про перехід. Посилання на неї додається у вихідний список, за нею додається команда умовного переходу по хибі (УПХ). Контейнер на вершині стеку модифікується: до нього додається посилання на щойно згенеровану мітку, що не вказує на жодну лексему.

При виявленні у вхідній послідовності оператору enddo. Зі стека видаляються всі елементи та додаються до вихідного рядка, доки не буде виявлено перший контейнер з позначкою while. В контейнері присутні дві мітки: перша відразу ж поміщається у вихідний список, за нею поміщається команда безумовного переходу (БП) та інша мітка з контейнера, що не мала вказівника. При цьому додається вказівник на поточний елемент у вихідному списку.

Оскільки в контейнері були лише посилання на мітки, у вихідній послідовності отримали по 2 посилання на кожну мітку. Це важливо, оскільки далі модуль, що відповідає за виконання, буде ігнорувати мітки, які містять вказівник на

						Лист
Изм	Лист	№ докум	Подп	Дата		24

власне положення в полізі, а по мітках, з вказівником на позицію відмінну від їхньої позиції, буде відбуватися перехід.

Навантаження на службові слова циклу показано в таблиці 5.1.1.

Початковий оператор	Елементи полізу
do	
while	Мітка Мі з поточною позицією
(	
Логічний вираз	Логічний вираз у поданні поліза
)	Мі+1, УПХ
Список операторів	Список операторів у поданні поліза
enddo	Мі, БП, мітка Мі+1 з поточною позицією

Таблиця 5.1.1 - Навантаження на службові слова циклу do while

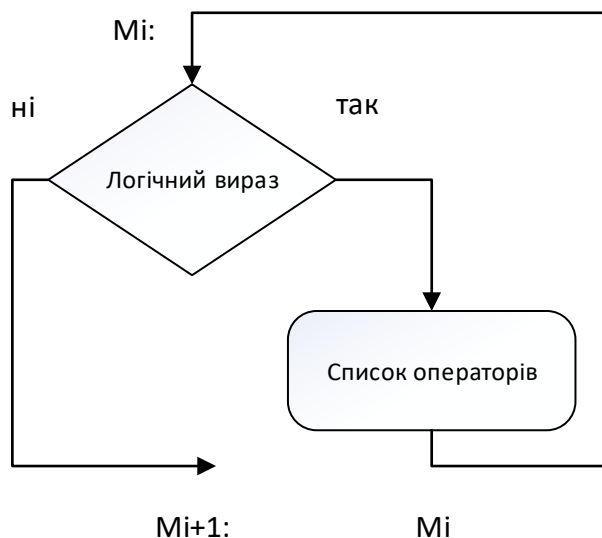


Рисунок 5.1.1 - Блок-схема оператора циклу

Конструкція умовного переходу if та тернарний оператор, будується подібним чином.

Початковий оператор	Елементи полізу
---------------------	-----------------

if	
(	
Логічний вираз	Логічний вираз у поданні поліза
)	
then	Мі УПХ
Список операторів	Список операторів у поданні поліза
fi	Мітка Мі з поточною позицією

Таблиця 5.1.2 - Навантаження на службові слова умовної конструкції if.



Рисунок 5.1.2 - Блок-схема оператора циклу

Початковий оператор	Елементи полізу
[	
Логічний вираз	Логічний вираз у поданні поліза
]	
?	Мі, УПХ
Арифметичний вираз	Арифметичний вираз у поданні поліза
:	Мі+1, БП, мітка з поточною позицією Мі
Арифметичний вираз	Арифметичний вираз у поданні поліза
¶	мітка з поточною позицією Мі+1

Таблиця 5.1.3 - Навантаження на службові слова конструкції тернарного оператора.

Оператор вводу read переноситься до полізу, як змінна, в яку відбувається запис даних та унарний оператор RD. Оскільки оператор read підтримує одночасний ввід декількох змінних, подібні дії ми проводимо для всіх параметрів. Аналогічним чином перетворюється оператор write.

UniversalTableWindwos			Stack	Output
Input				
begin float startValue = 3 * ( 5 - 2 * 4,2) - 4 int buffer = startValue * 3 - 2 write ( startValue, buffer) end				test2
begin float startValue = 3 * ( 5 - 2 * 4,2) - 4 int buffer = startValue * 3 - 2 write ( startValue, buffer) end				test2
float startValue = 3 * ( 5 - 2 * 4,2) - 4 int buffer = startValue * 3 - 2 write ( startValue, buffer) end				test2
= 3 * ( 5 - 2 * 4,2) - 4 int buffer = startValue * 3 - 2 write ( startValue, buffer) end				test2 startValue
3 * ( 5 - 2 * 4,2) - 4 int buffer = startValue * 3 - 2 write ( startValue, buffer) end				= test2 startValue
( 5 - 2 * 4,2) - 4 int buffer = startValue * 3 - 2 write ( startValue, buffer) end				= test2 startValue 3
( 5 - 2 * 4,2) - 4 int buffer = startValue * 3 - 2 write ( startValue, buffer) end				*= test2 startValue 3
5 - 2 * 4,2) - 4 int buffer = startValue * 3 - 2 write ( startValue, buffer) end				(*= test2 startValue 3
- 2 * 4,2) - 4 int buffer = startValue * 3 - 2 write ( startValue, buffer) end				(*= test2 startValue 3 5
2 * 4,2) - 4 int buffer = startValue * 3 - 2 write ( startValue, buffer) end				-(*= test2 startValue 3 5
* 4,2) - 4 int buffer = startValue * 3 - 2 write ( startValue, buffer) end				-(*= test2 startValue 3 5 2
4,2) - 4 int buffer = startValue * 3 - 2 write ( startValue, buffer) end				*-(*= test2 startValue 3 5 2
) - 4 int buffer = startValue * 3 - 2 write ( startValue, buffer) end				*-(*= test2 startValue 3 5 2 4,2
- 4 int buffer = startValue * 3 - 2 write ( startValue, buffer) end				*= test2 startValue 3 5 2 4,2 -
4 int buffer = startValue * 3 - 2 write ( startValue, buffer) end				= test2 startValue 3 5 2 4,2 -
int buffer = startValue * 3 - 2 write ( startValue, buffer) end				= test2 startValue 3 5 2 4,2 -
int buffer = startValue * 3 - 2 write ( startValue, buffer) end				= test2 startValue 3 5 2 4,2 -
= startValue * 3 - 2 write ( startValue, buffer) end				= test2 startValue 3 5 2 4,2 -
startValue * 3 - 2 write ( startValue, buffer) end				= test2 startValue 3 5 2 4,2 -
* 3 - 2 write ( startValue, buffer) end				= test2 startValue 3 5 2 4,2 -
3 - 2 write ( startValue, buffer) end				= test2 startValue 3 5 2 4,2 -
- 2 write ( startValue, buffer) end				= test2 startValue 3 5 2 4,2 -
2 write ( startValue, buffer) end				= test2 startValue 3 5 2 4,2 -
write ( startValue, buffer) end				= test2 startValue 3 5 2 4,2 -
end				= test2 startValue 3 5 2 4,2 -
end				= test2 startValue 3 5 2 4,2 -

Рисунок 5.1.1 - Службове вікно транслятора. Покрокова побудова полізу.



## 6 МОДУЛЬ ВИКОНАННЯ ПОЛІЗУ

Даний блок приймає на вхід згенерований поліз та виконує його послідовно переглядаючи кожен символ інверсного запису. Кожен символ має свої функції, які й виконує даний блок. Усі арифметичні вирази використовують додатковий стек для проміжних обчислень. Поточний символ, який представляє собою арифметичну операцію застосовується для двох або однієї комірки стеку(бінарна та унарна операції ) та записує результат у комірку пам'яті останнього операнда.

Для взаємодії з користувачем при виконанні відкривається консоль. Ввід та вивід відбувається при виконанні операцій RD та WT стандартними методами для роботи з консоллю.

Подібно до С-подібних мов, при присвоєнні змінній цілого типу значення з плаваючою точкою дробова частина ігнорується.

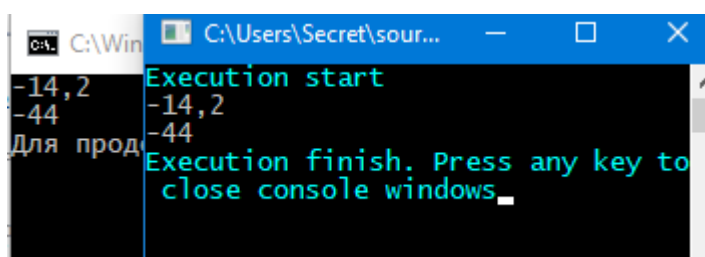
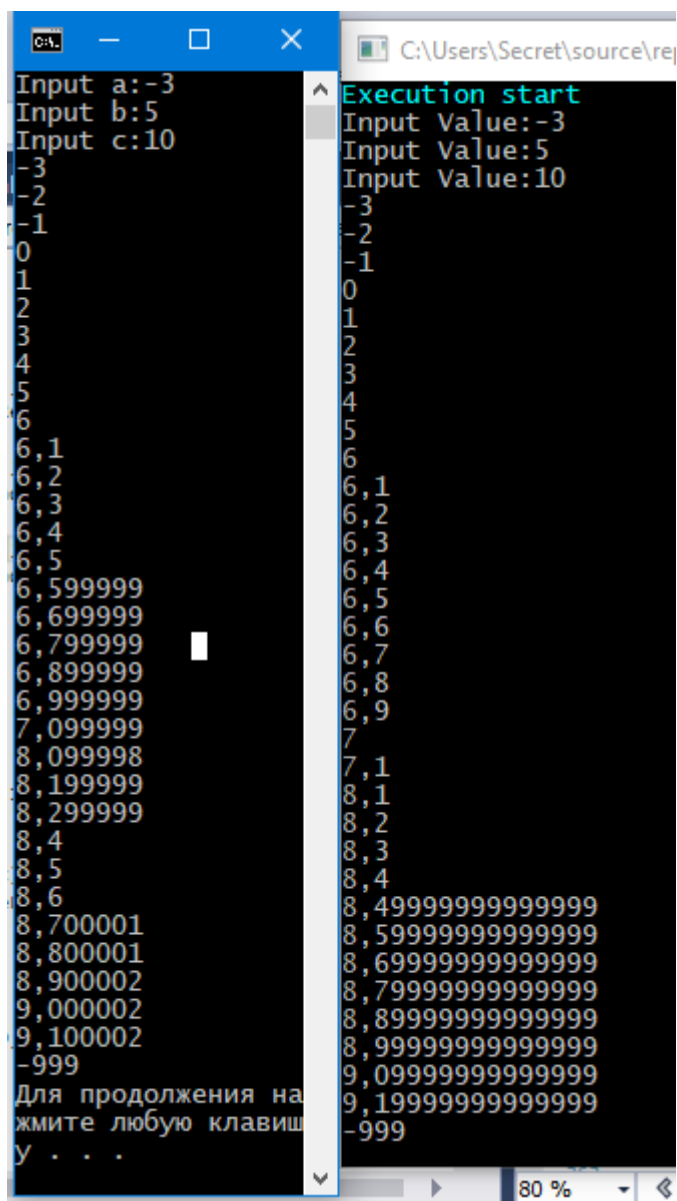


Рисунок 6.1 - Порівняння виводу програми на С# (ліворуч) та програми на розробленій мові(праворуч).



```
Input a:-3
Input b:5
Input c:10
-3
-2
-1
0
1
2
3
4
5
6
6,1
6,2
6,3
6,4
6,5
6,599999
6,699999
6,799999
6,899999
6,999999
7,099999
8,099998
8,199999
8,299999
8,4
8,5
8,6
8,700001
8,800001
8,900002
9,000002
9,100002
-999
Для продовження на
жміть будь-яку клавіш
У . . .

Execution start
Input Value:-3
Input Value:5
Input Value:10
-3
-2
-1
0
1
2
3
4
5
6
6,1
6,2
6,3
6,4
6,5
6,6
6,7
6,8
6,9
7
7,1
8,1
8,2
8,3
8,4
8,4999999999999999
8,5999999999999999
8,6999999999999999
8,7999999999999999
8,8999999999999999
8,9999999999999999
9,0999999999999999
9,1999999999999999
-999
```

Рисунок 6.2 - Порівняння виводу програми на С# (ліворуч) та програми на розробленій мові(праворуч).

UniversalTableWindwos		
Number	Stack	RPN
1		test2 startValue 3 5 2 4,2 *- 4 -= buffer startValue 3 * 2 -= startValue WT buffer WT
2	test2	startValue 3 5 2 4,2 *- 4 -= buffer startValue 3 * 2 -= startValue WT buffer WT
3	startValue test2	3 5 2 4,2 *- 4 -= buffer startValue 3 * 2 -= startValue WT buffer WT
4	3 startValue test2	5 2 4,2 *- 4 -= buffer startValue 3 * 2 -= startValue WT buffer WT
5	5 3 startValue test2	2 4,2 *- 4 -= buffer startValue 3 * 2 -= startValue WT buffer WT
6	2 5 3 startValue test2	4,2 *- 4 -= buffer startValue 3 * 2 -= startValue WT buffer WT
7	4,2 2 5 3 startValue test2	*- 4 -= buffer startValue 3 * 2 -= startValue WT buffer WT
8	8,4 5 3 startValue test2	-* 4 -= buffer startValue 3 * 2 -= startValue WT buffer WT
9	-3,4 3 startValue test2	* 4 -= buffer startValue 3 * 2 -= startValue WT buffer WT
10	-10,2 startValue test2	4 -= buffer startValue 3 * 2 -= startValue WT buffer WT
11	4 -10,2 startValue test2	-= buffer startValue 3 * 2 -= startValue WT buffer WT
12	-14,2 startValue test2	= buffer startValue 3 * 2 -= startValue WT buffer WT
13	test2	buffer startValue 3 * 2 -= startValue WT buffer WT
14	buffer test2	startValue 3 * 2 -= startValue WT buffer WT
15	startValue buffer test2	3 * 2 -= startValue WT buffer WT
16	3 startValue buffer test2	* 2 -= startValue WT buffer WT
17	-42,6 buffer test2	2 -= startValue WT buffer WT
18	2 -42,6 buffer test2	-= startValue WT buffer WT
19	-44,6 buffer test2	= startValue WT buffer WT
20	test2	startValue WT buffer WT
21	startValue test2	WT buffer WT
22	test2	buffer WT
23	buffer test2	WT

Рисунок 6.3 - Службове вікно транслятора. Приклад покрокового виконання програми.

## 7 ОПИС ПРОЕКТУ

Для зручності розробки проект було розділено на 6 окремих частин.

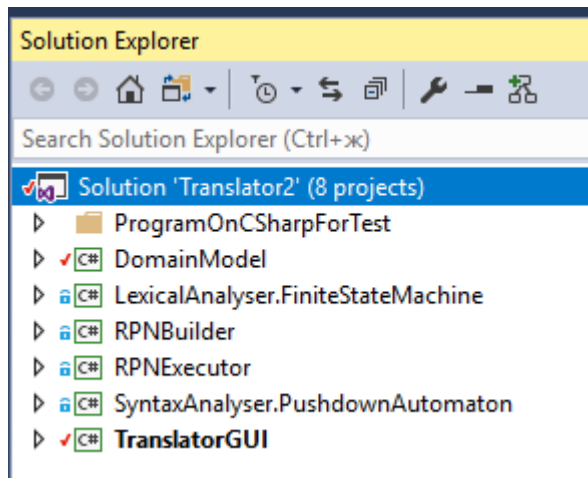


Рисунок 7.1 – Знімок частини екрану «Solution Explorer» Visual Studio з розробленим проектом

Модуль	Опис	Після компіляції
DomainModel	Містить основні абстракції, наприклад IOperator, ISymbol, та класи, наприклад Constant, VariablePool, ServiceWord та інші сутності, що використовують у всьому проекті.	dll
LexicalAnalyser.FiniteStateMachine	Реалізація інтерфейса ILexialAnalyser з DomainModel. Відповідає за перетворення тексту програми на лексеми.	dll
SyntaxAnalyser.PushdownAutomaton	Реалізація інтерфейса ISyntaxAnalyser з DomainModel. Відповідає за перевірку слідування лексем.	dll
RPNBuilder	Реалізація інтерфейса IRPNBuilder з DomainModel. Приймає список лексем, та записує програму в польському записі за допомогою алгоритму Дейкстри	dll

RPNExecutor	Реалізація інтерфейса IRPNExecutor з DomainModel. Відповідає за виконання вхідного поліза	dll
TranslatorGUI	Графічний інтерфейс транслятора	exe

Таблиця 7.1 - Опис модулів програми

## 7.1 Інтерфейс програми

Інтерфейс користувача розроблено на технології WPF, складається з одного головного вікна (наприклад Рисунок 4.3.1 ) та одного службового вікна (наприклад Рисунок 5.1.1 ) яке приймає різний вигляд залежно від даних. У верхній частині головного вікна знаходиться меню, через яке можна відкрити або ж зберегти файл, запустити введений код на виконання та відкрити службове вікно.

						Лист
Изм	Лист	№ докум	Подп	Дата		32

## ВИСНОВКИ

У курсовій роботі був реалізований транслятор, який перетворює конструкції спеціально розробленої мови у польський інверсний запис. Були спроектовані три незалежних блоки : лексичний аналізатор, синтаксичний аналізатор та генератор коду. У якості додатка була написана програма, яка приймає згенерований поліз та виконує його. Так як блок-виконувач виходить за рамки задач транслятора , він не був описаний.

Розроблений транслятор може використовуватися для отримання базових навичок у програмуванні та написання простих розрахункових програм. Перевагою даного транслятору перед комерційними продуктами є швидкодія та зрозумілість, так як він розрахований на написання короткий простих програм.

Виконання курсової роботи дала змогу глибоко зрозуміти методи та принципи побудови транслятора.

Для реалізації поставленої задачі використовувалися мова програмування C#, засоби Microsoft Visual Studio 2017 та технологія WPF для графічного представлення інтерфейсу користувача.

						Лист
Изм	Лист	№ докум	Подп	Дата		33

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Медведєва, В.М. Транслятори: лексичний та синтаксичний аналізатори : Навчальний посібник / В.М. Медведєва, В.А. Третяк. – К.: Політехніка, 2012. – 148 с.
2. Медведєва, В.М. Транслятори: внутрішнє подання програм та інтерпретація: Навчальний посібник / В.М. Медведєва, В.А. Третяк. – К.: Політехніка, 2015.

						Лист
Изм	Лист	№ докум	Подп	Дата		34

# ДОДАТОК

Опис програмного коду

УКР.НТУУ «КПІ ім. І. Сікорського»\_ ТЕФ\_АПЕПС\_ТР\_52

Листів 4

Київ – 2018

						Лист
Изм	Лист	№ докум	Подп	Дата		35



Код класу, що виконує поліз

```
public class RPNExecutor:IRPNExecutor
{
    Stack<IRPNElement> stack = new Stack<IRPNElement>();
    List<IRPNElement> inputList;
    private int i;

    public List<SnapCalculationRPN> SnapsList { get; private
set; } = new List<SnapCalculationRPN>();

    IEnumerable<ISnap> IRPNExecutor.SnapsList
=>SnapsList.Cast<ISnap>();
    public RPNExecutor(List<IRPNElement> inputList)
    {
        this.inputList = inputList;
        this.i = 0;
    }

    public void Execute()
    {
        for (; i < inputList.Count; i++)
        {
            SnapsList.Add(new SnapCalculationRPN(stack, input-
List.Skip(i).ToList()));

            if (inputList[i] is Identifier)
stack.Push(inputList[i]);
            else if (inputList[i] is ConstantLink)
stack.Push(inputList[i]);
            else if (inputList[i] is Label label)
            {
                if (label.Position.Value != i)
stack.Push(inputList[i]);
            }
            else if (inputList[i] is Operator operator)
            {
                IRPNElement resultOperation = null;

                if (operator.Sign == "*") resultOperation = Mul-
tiple(GetFromStackDigitValue(), GetFromStackDigitValue());
                else if (operator.Sign == "/") resultOperation =
Devide(GetFromStackDigitValue(), GetFromStackDigitValue());
                else if (operator.Sign == "+") resultOperation =
Plus(GetFromStackDigitValue(), GetFromStackDigitValue());
            }
        }
    }
}
```

						Лист
Изм	Лист	№ докум	Подп	Дата		36

```

        else if (oprator.Sign == "-") resultOperation =
Minus(GetFromStackDigitValue(), GetFromStackDigitValue());

        else if (oprator.Sign == "and") resultOperation
= And(GetFromStackBoolValue(), GetFromStackBoolValue());
        else if (oprator.Sign == "or") resultOperation =
Or(GetFromStackBoolValue(), GetFromStackBoolValue());
        else if (oprator.Sign == "not") resultOperation
= Not(GetFromStackBoolValue());

        else if (oprator.Sign == ">") resultOperation =
More(GetFromStackDigitValue(), GetFromStackDigitValue());
        else if (oprator.Sign == "<") resultOperation =
Less(GetFromStackDigitValue(), GetFromStackDigitValue());
        else if (oprator.Sign == ">=") resultOperation =
MoreEqual(GetFromStackDigitValue(), GetFromStackDigitValue());
        else if (oprator.Sign == "<=") resultOperation =
LessEqual(GetFromStackDigitValue(), GetFromStackDigitValue());
        else if (oprator.Sign == "==") resultOperation =
Equal(GetFromStackDigitValue(), GetFromStackDigitValue());
        else if (oprator.Sign == "!=") resultOperation =
NotEqual(GetFromStackDigitValue(), GetFromStackDigitValue());

        else if (oprator.Sign == "RD")
        {
            Identifier link = stack.Pop() as Identifier;
            if (link == null) throw new Excep-
tion($"parameter in Read must be Link type ");

            Console.Write("Input Value:");
            try
            {
                link.Value = Dou-
ble.Parse(Console.ReadLine());
                stack.Push(link);
            }
            catch (FormatException)
            {
                throw new Exception("Inputed value isn`t
digit");
            }
        }
        else if (oprator.Sign == "WT")
        {
            var value = stack.Pop();
            if (value is Identifier link)
            {
                Console.WriteLine(link.Substring+ ":
"+link.Value == null ? "null" : link.Value.Value.ToString());
            }
        }

```

						Лист
						37
Изм	Лист	№ докум	Подп	Дата		

```

        else if (value is Constant cons)
        {
            Console.WriteLine(cons.Value);
        }
    }

    else if (oprator.Sign == "UT")
    {
        var value = stack.Pop();
        if (value is Label l)
        {
            if (i != l.Position.Value)
            {
                i = l.Position.Value;
            }
        }
        else throw new Exception("Problem with un-
coditional transmit");
    }
    else if (oprator.Sign == "CTbM")
    {
        var value1 = stack.Pop();
        var value2 = stack.Pop();

        //todo:here maybe problem
        if (value1 is Label l && value2 is BoolType
b)
        {
            if (i != l.Position.Value && b.Value ==
false)
            {
                i = l.Position.Value;
            }
        }
        else throw new Exception("Problem with codi-
tional transmit by mistake");
    }

    else if (oprator.Sign == "=")
    {
        var value1 = stack.Pop();
        var value2 = stack.Pop();
        if (value2 is Identifier link)
        {
            if (value1 is ConstantLink cons)
link.Value = cons.Value;
            else if (value1 is DigitType d)
link.Value = d.Value;
        }
        else throw new Exception("Seems, we have a
problem: first operand isn't link");
    }

```

```

        }
        else continue;

        if (resultOperation != null)
        {
            stack.Push(resultOperation);
        }
    }
}

double GetFromStackDigitValue()
{
    var value = stack.Pop();

    if (value is ConstantLink c) return c.Value;
    else if (value is Identifier l) return l.Value.Value;
    else if (value is DigitType d) return d.Value;
    else throw new Exception("Can`t convert into constant ot
Link value from stack");
}

bool GetFromStackBoolValue()
{
    var value = stack.Pop();
    if (value is BoolType b) return b.Value;
    else throw new Exception("Can`t convert into Bool value
from stack");
}

```

						Лист
Изм	Лист	№ докум	Подп	Дата		39