**ЗМІСТ**

[ВСТУП 5](#_Toc515494671)

[1. ЗАВДАННЯ 6](#_Toc515494672)

[2. СТРУКТУРА ТРАНСЛЯТОРА 7](#_Toc515494673)

[3. ГРАМАТИКА МОВИ 8](#_Toc515494674)

[3.1 Приклад програми 10](#_Toc515494675)

[4. ЛЕКСИЧНИЙ АНАЛІЗАТОР 11](#_Toc515494676)

[4.1 Приклад роботи лексичного аналізатора 17](#_Toc515494677)

[5. СИНТАКСИЧНИЙ АНАЛІЗАТОР 19](#_Toc515494678)

[5.1 Приклад роботи синтаксичного аналізатора 28](#_Toc515494679)

[6.ПОЛЬСЬКИЙ ІНВЕРСНИЙ ЗАПИС 29](#_Toc515494680)

[6.1 Побудова ПОЛІЗ 29](#_Toc515494681)

[6.2 Виконання ПОЛІЗ 34](#_Toc515494682)

[6.3 Приклад роботи модуля виконання ПОЛІЗу 35](#_Toc515494683)

[ВИСНОВКИ 37](#_Toc515494684)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 38](#_Toc515494685)

[ДОДАТОК 1 39](#_Toc515494686)

# ВСТУП

В наш час штучні мови, які використовують для опису предметної області текстове представлення, широко застосовуються не тільки в програмуванні, але і в інших областях. З їх допомогою описується структура всіляких документів, тривимірних віртуальних світів, графічних інтерфейсів користувача і багатьох інших об'єктів, що використовуються в моделях і в реальному світі. Для того, щоб ці текстові описи були коректно складені, а потім правильно розпізнані і інтерпретовані, використовуються спеціальні методи їх аналізу і перетворення. В основі методів лежить теорія мов і формальних граматик, а також теорія автоматів. Програмні системи, призначені для аналізу та інтерпретації текстів, називаються трансляторами.  
 Незважаючи на те, що до теперішнього часу розроблені тисячі різних мов і їх трансляторів, процес створення нових додатків в цій області не припиняється. Це пов'язано як з розвитком технології виробництва обчислювальних систем, так і з необхідністю вирішення все більш складних прикладних задач.

# ЗАВДАННЯ

Використовуючи теоретичний матеріал по курсу «Лінгвістичне забезпечення САПР2», реалізувати на будь-якій мові програмування транслятор у відповідності до індивідуального завдання. Транслятор має складатися з 2-ох частин:

1) лексичний аналізатор

2) синтаксичний аналізатор

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

Індивідуальне завдання

Граматика мови має такі особливості :

1) в арифметичному виразі можуть використовуватися: +, -, \*, /, константи з плаваючою точкою, ();

2) оператор циклу має вигляд:

while <лог. вираз> do <операторний блок>

3) оператор умовного переходу :

if <лог. вираз> ? <операторний блок> : <операторний блок>

4) роздільник « ; »

5) додаткові оператори:

оператор goto <мітка>

6) метод лексичного аналізу: скінченний автомат.

7) метод синтаксичного аналізу: автомат з магазинною пам’яттю.

# СТРУКТУРА ТРАНСЛЯТОРА

Транслятор є програмою, котра здатна сприймати ланцюжок символів певного вигляду (тобто текст програми, написаний початковою мовою) і видавати інший рядок символів (програму на машинній мові, так звану об’єктну або цільову програму).

Розроблений транслятор є трьох прохідним(рисунок 2.1).

Вихідний текст програми

Лексичний аналізатор

Синтаксичний аналізатор

Побудова

ПОЛІЗ

Виконання ПОЛІЗ

Рисунок 3.1 ― Структура транслятора

До складу будь-якого транслятору входить три основних компоненти: лексичний і синтаксичний аналізатори, генератор ПОЛІЗ.

На фазі лексичного аналізу вихідний текст програмується у вигляді ланцюжка незв’язаних один з одним символів, розбиваючись в лексеми: імена змінних, констант, службові слова, вирази.

Після лексичного аналізу слідує етап синтаксичного аналізу, на якому перевіряється правильність слідування лексем, тобто граматичні правила вихідного ланцюжка.

Генератор ПОЛІЗ виконує генерацію коду, тобто переводить вихідний текст програми в ПОЛІЗ.

# ГРАМАТИКА МОВИ

В цьому розділі представлена граматика мови для транслятора. Існують різні способи запису синтаксичних правил, що в основному визначається умовними позначеннями і обмеженнями на структуру правил, прийнятими в використовуваних метамовах.

Мови, за допомогою котрих описується інші мови,називаються метамовами. Метамова, запропонованa Бекусoм і Наурум, вперше було використано для опису синтаксису реальної мови програмування Алгол 60. Поряд з новими позначеннями метасимволов, в ній використовувалися змістовні позначення нетерміналів. Це зробило опис мови наочніше і дозволило надалі широко використовувати дану нотацію для опису реальних мов програмування.

Згідно з завданнями була розроблена така граматика.

1. < програма > program < ім’я програми> var <список оголошень >

{ <список операторів > }

1. < ім’я програми >::= < ідентифікатор>
2. < список оголошень >::=<оголошення>;|

|<список оголошень>;< оголошення >

1. <oголошення>::=<тип> <список ідентифікаторів>|

|<підтип> < список міток >

1. <тип>::=num
2. <підтип>::=label
3. < список ідентифікаторів > ::= < ідентифікатор>|

|< список ідентифікаторів > ,< ідентифікатор>

1. < список міток > ::= < мітка >|< список міток > ,< мітка >
2. < мітка >::= < ідентифікатор>
3. <список операторів > ::= <оператор>; |<список операторів ><оператор>;
4. <оператор> ::= < присвоєння >|<вводу>|< виводу >|<цикл>|

|< умовний >|<безумовний>|<безумовний перехід>

1. < присвоєння >::=< ідентифікатор>=<вираз >
2. <вводу>::=read(< список ідентифікаторів > )
3. <виводу>::= write(< список ідентифікаторів > )
4. <цикл>::=while <логічний вираз> do <операторний блок>
5. <умовний>::= if <логічний вираз>?<операторний блок>:
6. <операторний блок>
7. <безумовний>::= goto < мітка >
8. <безумовний перехід>=< мітка >:
9. <вираз>::= <терм>|<вираз> +<терм>|<вираз> -<терм>
10. <терм>::=<множ.>|<терм>\*<множ.>|<терм>\<множ.>
11. <множ.>::= < ідентифікатор>|<константа>|(<вираз>)
12. < операторний блок >::=<оператор> |{ <список операторів>}
13. <логічний вираз>::=< логічний терм>|

|< логічний вираз>||< логічний терм>

1. < логічний терм>::=< логічний множ.>|

|< логічний терм> && < логічний множ.>

1. < логічний множ.> ::= <вираз><знак відношення><вираз>|

|(<логічний вираз>)

1. <знак відношення>::= < |<=|!=| > |>=|==
2. < ідентифікатор>::=<буква>|< ідентифікатор >< буква >|

|< ідентифікатор ><цифра>

1. <константа>::=<чзфт>|<чзфт>E<порядок>|-<чзфт>|-<чзфт>E<порядок>
2. <чзфт>::=<цбз>|<цбз>.|.<цбз>|<цбз>.<цбз>
3. <порядок>::=<цбз>|<знак><цбз>
4. <цбз>::=<цифра>|<цбз><цифра>
5. <знак>::=+|-
6. <буква>::=a|b|c|…|z
7. <цифра>::=0|1|…|9

В граматиці також описані зручні та інтуїтивні оператори вводу, виводу, оператор умови та циклу.

## 3.1 Приклад програми

program first

var

num b,c;

label k

{

read(b);

c=-0.5;

if(b>100 && c>10) ? { c=(3+(c\*8)); } :

{

while(c<0) do{

b=b+1;

c=c+1;

write(b,c);

};

goto k;

};

k:;

write(b,c);

c=(2\*b)/c;

}

# 4. ЛЕКСИЧНИЙ АНАЛІЗАТОР

Лексичний аналіз - перша фаза процесу трансляції, призначена для групування символів вхідного ланцюжка в більші конструкції,так звані лексеми.

Таблиця лексем – це термінальний словник граматики. Вона не залежить від тексту програми і є одною на всі вхідні програми(таблиця 4.1).

Незважаючи на те, що лексичний аналізатор обробляє вхідний ланцюжок, зручніше на його вхід подавати не просто окремі символи, а символи, згруповані за категоріями. Тому, перед лексичним аналізатором здійснюється додаткова обробка, що зіставляє з кожним символом його клас, що дозволяє сканеру маніпулювати єдиним поняттям для цілої групи символів(таблиця 4.2).

Згідно до варіанта курсової роботи було використано метод лексичного аналізу – скінченний автомат.

Розглянемо реалізацію сканера за допомогою скінченого автомату. Побудова скінченого автомата можна здійснити з використанням ряду граматик з лівої рекурсією і діаграм Вірта. На практиці, замість праволінійної граматики, зручніше використовувати діаграми Вірта. Вони набагато наглядніші при описі правил. Крім цього, між діаграмами Вірта, що не містять нетерміналів, і кінцевими автоматами існує однозначна відповідність. Практично це два еквівалентних метода представлення однієї моделі, яка одночасно може служити як механізмом породження, так і механізмом розпізнавання.

Скінченний автомат, еквівалентний діаграмі Вірта, що складається тільки з терміналів, будується наступним чином:

1. Початкова дуга діаграми перетвориться в початковий стан кінцевого автомата.
2. Кінцева дуга діаграми утворює заключне стан кінцевого автомата.
3. Виходи окремих дуг, що з'єднують символи, і точки розгалуження інших дуг діаграми утворюють безліч інших станів кінцевого автомата.
4. Кінцеві стану діаграми є допускають станами кінцевого автомата.

Таблиця 4.1. Вхідна таблиця лексем

|  |  |  |  |
| --- | --- | --- | --- |
| Лексема | Код лексеми | Лексема | Код лексеми |
| program | 1 | <= | 19 |
| var | 2 | != | 20 |
| { | 3 | == | 21 |
| } | 4 | > | 22 |
| ; | 5 | >= | 23 |
| : | 6 | + | 25 |
| num | 7 | - | 26 |
| label | 8 | \* | 27 |
| read | 9 | / | 28 |
| write | 10 | ( | 29 |
| while | 11 | ) | 30 |
| do | 12 | ! | 31 |
| goto | 13 | = | 32 |
| if | 14 | , | 33 |
| ? | 15 | id | 34 |
| || | 16 | con | 35 |
| && | 17 | lab | 36 |
| < | 18 |  | |

Таблиця 4.2. Таблиця класів

|  |  |
| --- | --- |
| Символи | Клас |
| a-d,f-z | Б |
| 0-9 | Ц |
| / \*(),;,?: | ОР |
| < | < |
| > | > |
| ! | ! |
| = | = |
| +- | + |
| e | E |
| . | . |
| \_ | \_ |

Тоді скінченний автомат можна вважати деревом,в якому вузли відповідають станам,а дуги-переходам(рисунок 4.1)[3].

Проте представляти скінченні автомати у канонічному вигляді,які враховують всі множини і початковий стане не зручно та громіздко.Тому будемо задавати скінченний автомат у вигляді таблиці переходів(таблиця 4.3)[2].

>

=

Lex=>

Lex=>=

Lex=<

=

Lex=<=

<

ОР

Lex=OP

\_

Lex=id

Так

Lex=TRM

Hi

Lex=j

Б

БЕЦ

Lex=con

Ц

Ц

**.**

error

Ц

Ц

E

+

Ц

Lex=con

error

Ц

Lex=con

error

Ц

**.**

Е

error

error

Lex=&&

&

&

|

Lex=||

error

|

=

Lex= ==

Lex= =

=

!

=

Lex= !=

error

Риисунок 4.1-Діаграма станів

Таблиця 4.3. Таблиця переходів

|  |  |  |  |
| --- | --- | --- | --- |
| А | Мітка переходу | В | Семантична підпрограма |
| 1 | Б | 2 | [≠]помилка |
| Ц | 3 |
| . | 4 |
| OP | 21 |
| < | 9 |
| > | 11 |
| ! | 13 |
| = | 15 |
| & | 17 |
| | | 19 |
| - | 3 aбо 21 |
| + | 21 |
| 2 | Б | 2 | [≠]виділення лексеми TRM або id |
| Е | 2 |
| Ц | 2 |
| 3 | Ц | 3 | [≠]виділення лексеми con |
| . | 5 |
| Е | 6 |
| 4 | Ц | 5 | [≠]помилка |
| 5 | Ц | 5 | [≠]виділення лексеми con |
| Е | 6 |
| 6 | Ц | 8 | [≠]помилка |
| + | 7 |
| 7 | Ц | 8 | [≠]помилка |
| 8 | Ц | 8 | [≠]виділення лексеми con |

|  |  |  |  |
| --- | --- | --- | --- |
| 9  Продовження таблиці 4.3 | = | 10 | [≠]виділення лексеми <  [=]перехід на стан 10 |
| 10 |  |  | виділення лексеми <= |
| 11 | = | 12 | [≠]виділення лексеми >  [=]перехід на стан 12 |
| 12 |  |  | виділення лексеми >= |
| 13 | = | 14 | [≠]помилка  [=]перехід на стан 14 |
| 14 |  |  | виділення лексеми != |
| 15 | = | 16 | [≠]виділення лексеми =  [=]перехід на стан 16 |
| 16 |  |  | виділення лексеми == |
| 17 | | | 18 | [≠]помилка  [=]перехід на стан 18 |
| 18 |  |  | виділення лексеми || |
| 19 | & | 20 | [≠]помилка  [=]перехід на стан 20 |
| 20 |  |  | виділення лексеми && |
| 21 |  |  | виділення лексеми OP |

## 4.1 Приклад роботи лексичного аналізатора

Для перевірки правильності роботи лексичного аналізатора, було написано тестову програму без помилок. В результаті обробки тестової програми за допомогою лексичного аналізатора було отримано таблицю лексем(рисунок 4.1.)

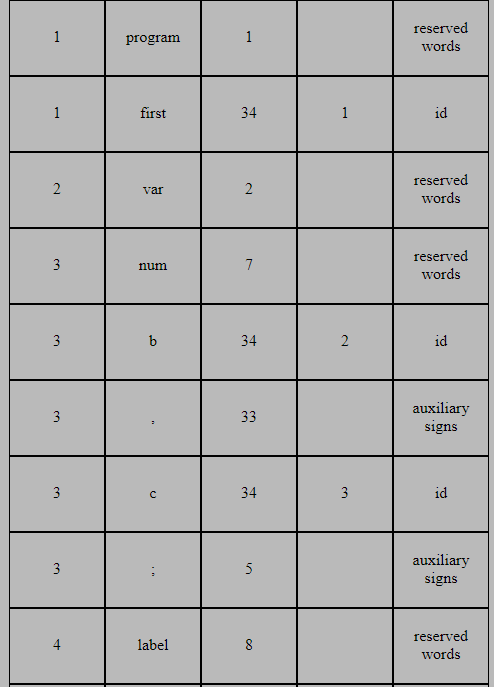


Рисунок 4.1 — Результат роботи лексичного аналізатора

При спробі аналізу програми з навмисно допущеною помилкою, лексичний аналізатор виведе інформацію про наявність помилки (рисунок 4.2). Також лексичний аналізатор,не допускає використання неоголошених змінних,і контролює,щоб ім’я змінної не починалося з цифри(рисунок 4.3) . Розроблений лексичний аналізатор здатний виділяти лексеми з вхідного рядка, розрізняти ідентифікатори, константи та мітки,а також групувати їх за призначенням.



Рисунок 4.2 — Приклад програми з помилкою

# 5. СИНТАКСИЧНИЙ АНАЛІЗАТОР

Після розбиття на лексеми в трансляторі іде фаза синтаксичного аналізу (граматичного розбору), мета якого – перевірка правильності порядку лексем.

Синтаксичний розбір (розпізнавання) є першим етапом синтаксичного аналізу. Саме при його виконанні здійснюється підтвердження того, що вхідний ланцюжок символів є програмою, а окремі підланцюжки становлять синтаксично правильні програмні об'єкти. Слідом за розпізнаванням окремих підланцюжків здійснюється аналіз їх семантичної коректності на основі накопиченої інформації. Потім проводиться додавання нових об'єктів в об'єктну модель програми або в проміжне представлення.

Розбір призначений для доказу того, що аналізований вхідний ланцюжок, належить або не належить множині ланцюжків породжуваних граматикою даної мови. Виконання синтаксичного розбору здійснюється розпізнавачами, які є автоматами. Тому даний процес також називається розпізнаванням вхідного ланцюжка. Мета в тому, щоб відповісти на питання: чи належить аналізований ланцюжок множині правильних ланцюжків заданої мови. Відповідь "так" дається, якщо така приналежність встановлена. В іншому випадку дається відповідь "ні". Отримання відповіді "ні" пов'язане з поняттям відмови. Єдина відмову на будь-якому рівні веде до загального відмови[3].

Щоб отримати відповідь "так" щодо всього ланцюжка, треба його отримати для кожного правила, що забезпечує розбір окремого підланцюжка.

В цій роботі буде розглянуто синтаксичний розбір методом автомату з магазинною пам’яттю.

Автомат з магазинною пам’яттю (МПА, МП-автомат, магазинний автомат, стековий автомат) – це сімка

**,**

де

Q – скінченна множина станів;

Σ – вхідний алфавіт;

Z – алфавіт магазинної пам’яті;

δ – це відображення множини  в множину скінченних підмножин множини , тобто . Формально аргументами  є трійки δ(α, х, γ), де , х – вхідний символ або пустий ланцюжок, . Вихід δ складають пари (β, ζ), де β – новий стан, ζ – ланцюжок магазинних символів, що замінює γ на вершині стека. Наприклад, якщо  магазинний символ видаляється, якщо  магазин не змінюється;

q0 – початковий стан автомата;

– початковий символ магазину, котрий інакше називають маркером магазинної пам’яті;

F – скінченна множина заключних станів.

Конфігурацією МПА називається трійка , де – поточний стан автомата, х – невикористана частина вхідного рядка (якщо  вважається, що вхідний ланцюжок прочитано), γ – вміст магазину, найлівіший символ ланцюжка γ називається верхнім символом магазину або вершиною магазину, якщо , магазин вважається пустим.

Як і скінченні автомати, МПА поділяються на детерміновані та недетерміновані. Інтуїтивно, МПА є детермінованим, якщо в будь-якій конфігурації в нього немає можливості вибору різних переходів. Якщо вихід δ(α, х, γ) містить більше однієї пари, то МПА не є детермінованим. Однак, навіть якщо вихід δ(α, х, γ) містить одну пару, існує можливість вибору переходу по пустому ланцюжку δ(α, , γ). Таким чином, МПА  визначається як детермінований, якщо виконуються наступні умови:

δ(α, х, γ) містить не більше одного елемента для кожного α з Q, х з Σ та γ з множини Z;

якщо δ(α, х, γ) не пусто для деякого х з Σ, то δ(α, , γ) має бути пустою.

В даному випадку МП-автомат може продовжувати свою роботу при завершенні вхідного ланцюжка, але не може продовжувати роботу, якщо вичерпано магазин.

МПА як і скінченний автомат можна зображати графічно, при чому усі стани розташовуються в колах, початковий стан позначається стрілкою, що входить в нього, а вихідні стани позначаються подвійними колами[2].

При реалізації синтаксичного аналізатора будемо використовувати автомат для контролю послідовності лексем в процесі граматичного розбору.

}

;

program

id

var

1

2

3

<п/а оголошення>

4

<п/а оголошення>

Error

{

<п/а оператор>

Error

5

; 

<п/а оператор>

6

Quit

₵

Рисунок 5.1 — Діаграма для аксіоми програми

| **α** | **Мітка переходу** | **β** | **Стек** | **Семантична підпрограма** |
| --- | --- | --- | --- | --- |
| 1 | program | 2 |  | [≠]помилка |
| 2 | id | 3 |  | [≠]помилка |
| 3 | var | <п/а оголошення> | 4 | [≠]помилка |
| 4 | ;  } | <п/а оголошення>  <п/а оператор> | 4  5 | [≠]помилка |
| 5 | ; | 6 |  | [≠]помилка |
| 6 | }  ₵ | 201 | 5 | [=]вихід |

Таблиця 5.1. Список переходів МПА аксіоми програми

Підавтомат < оголошення > об’єднує <oголошення>::=<тип>

<список ідентифікаторів> |<підтип> < список міток >

Наведемо його діаграму (рисунок 5.2) і список переходів (таблиця 5.2).

₵

error

num

102

101

103

id

,

Quit

label

104

error

105

lab

Quit

,

₵

Рисунок 5.2 — Діаграма станів підавтомату <оголошення >

Таблиця 5.2. Список переходів МПА підавтомата <оголошення >

| **α** | **Мітка переходу** | | **β** | **Стек** | **Семантична підпрограма** |
| --- | --- | --- | --- | --- | --- |
| 101 | num  label | | 102  104 |  | [≠]помилка |
| 102 | id | | 103 |  | [≠]помилка |
| 103 | ,  ₵ | | 102 |  | [=] вихід |
| 104 | lab | 105 | |  | [≠]помилка |
| 105 | ,  ₵ | 104 | |  | [=] вихід |

Підавтомат < оператор> об’єднує < присвоєння >|<вводу>|

< виводу >|<цикл>|< умовний >|<безумовний>|<безумовний перехід>

Наведемо його діаграму (рисунок 5.3) і список переходів (таблиця 5.3).

error

lab

goto

213

error

error

204

=

Quit

<п/а вираз>

id

201

202

error

203

₵

(

)

read

205

206

id

,

Quit

error

write

₵

do

<п/а опер. блок>

while

<п/а лог. вираз>

207

208

Quit

:

₵

if

<п/а лог.вираз>

209

error

?

<п/а опер. блок>

210

error

<п/а опер. блок>

211

:

error

Quit

lab

212

error

error

Рисунок 5.3 — Діаграма станів підавтомату <оператор>

Таблиця 5.3. Список переходів МПА підавтомата < оператор >

| **α** | **Мітка переходу** | | **β** | **Стек** | **Семантична підпрограма** |
| --- | --- | --- | --- | --- | --- |
| 201 | id  read  write  while  if  lab  goto | | 202  204  204  < п/а лог. вираз >  < п/а лог. вираз >  212  213 | 207  209 | [≠]помилка |
| 202 | = | | <п/а вираз> | 203 | [≠]помилка |
| 203 | ₵ | |  |  | [=] вихід |
| 204 | ( | 205 | |  | [≠]помилка |
| 205 | id | 206 | |  | [≠]помилка |
| 206 | ,  ) | 205 | |  | [≠]помилка  [=] вихід |
| 207 | do | < п/а опер. блок > | | 208 | [≠]помилка |
| 208 | ₵ |  | |  | [=] вихід |
| 209 | ? | < п/а опер. блок > | | 210 | [≠]помилка |
| 210 | : | < п/а опер. блок > | | 211 | [≠]помилка |
| 211 | ₵ |  | |  | [=] вихід |
| 212 | : |  | |  | [≠]помилка  [=] вихід |
| 213 | lab |  | |  | [≠]помилка  [=] вихід |

Наведемо також приклад МПА підавтомата <вир.> (рисунок 5.4) і список переходів (таблиця 5.4).

)

Quit

+ - \* /

Id con

301

303

(

<п/а вираз>

302

error

Рисунок 5.4 — Діаграма станів підавтомату < вир>

Таблиця 5.4. Список переходів МПА підавтомата <вир.>

| **α** | **Мітка переходу** | **β** | **Стек** | **Семантична підпрограма** |
| --- | --- | --- | --- | --- |
| 301 | id  con  ( | 303  303  <п/а вираз> | 302 | [≠]помилка |
| 302 | ) | 303 |  | [≠]помилка |
| 303 | + | - | \* | / |  ₵ | 301 |  | [=]вихід |

Також для роботи програми потрібен підавтомат <лог.вираз>,тож представим його у вигляді діаграми(рисунок 5.5), а також побудуємо таблицю переходів,для відповідного підавтомата(таблиця 5.5).

₵

(

)

< > <= >= == !=

<п/а вираз>

Quit

**||** &&

401

403

<п/а лог. вираз>

404

error

<п/а вираз>

402

Рисунок 5.5 — Діаграма станів підавтомату < лог. вир>

Таблиця 5.5. Список переходів МПА підавтомата <лог.вир>

| **α** | **Мітка переходу** | **β** | **Стек** | **Семантична підпрограма** |
| --- | --- | --- | --- | --- |
| 401 | ( | <п/а лог.вираз> | 404 |  |
| ₵ | <п/а вираз> | 402 |
| 402 | < |>| >= | <= | == |!= | <п/а вираз> | 403 | [≠]помилка |
| 403 | && | **||** | 401 |  |  |
| ₵ |  | [=]вихід |
| 404 | ) | 403 |  |  |

Одним із важливих підавтоматів,для реалізації МПА підавтомата <оператор > є підавтомат <опер.блок>.Тож побудуємо його діаграму станів(рисунок 5.6),а також його таблицю списка переходів(таблиця 5.6).

₵

}

;

Quit

{

501

<п/а оператор>

502

503

<п/а оператор>

error

error

Рисунок 5.6 — Діаграма станів підавтомату <опер. блок>

Таблиця 5.6. Список переходів МПА підавтомата <опер.блок>

| **α** | **Мітка переходу** | **β** | **Стек** | **Семантична підпрограма** |
| --- | --- | --- | --- | --- |
| 501 | { | <п/а оператор> | 502 | [≠]помилка |
| 502 | ; | 503 |  | [≠]помилка |
| 503 | } |  |  | [=]вихід |
| ₵ | <п/а оператор> | 502 |  |

Продовження таблиці 5.6

## 5.1 Приклад роботи синтаксичного аналізатора

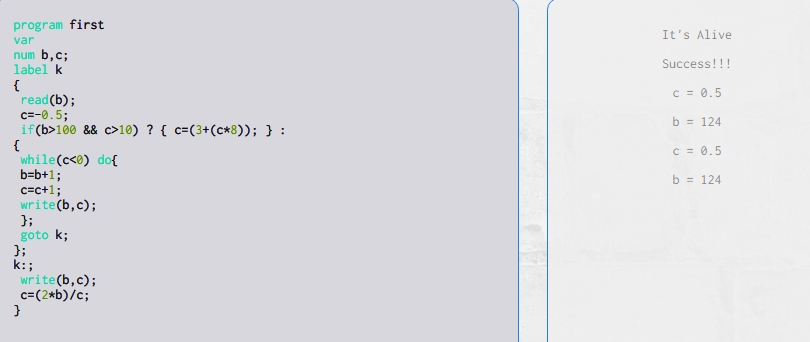
Приклад синтаксичного аналізатора з програмою без синтаксичних помилок показаний нижче(рисунок 5.7).

Рисунок 5.7 — Результат роботи лексичного аналізатора

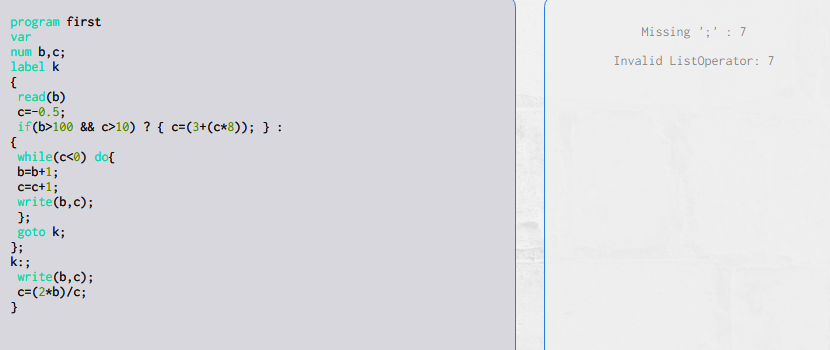
Приклад програми, що містить синтаксичні помилки, показано нижче (рисунок 5.8).

Рисунок 5.8 — Результат роботи лексичного аналізатора

# 6.ПОЛЬСЬКИЙ ІНВЕРСНИЙ ЗАПИС

## 6.1 Побудова ПОЛІЗ

У розроблюваному трансляторі, проміжна форма подання програми – польський інверсний запис (ПОЛІЗ). Він використовується у зв’язку з рядом причин:

1. Операнди в ПОЛІЗ, розташовані в такому ж порядку як і у вхідній програмі.

2. Операції в ПОЛІЗ, розташовані в порядку виконання з врахуванням пріоритету операцій.

3. Операції в ПОЛІЗ, розташовані безпосередньо за своїми операндами.

Оскільки, стадії лексичного та синтаксичного аналізу не впливають на порядок слідування операцій, то після етапу синтаксичного аналізу генерується ПОЛІЗ вхідної програми, по якому програма і буде виконуватися. Для побудови ПОЛІЗ створено алгоритм (алгоритм Дейкстри), який базується на явному завданні пріоритетів операцій (таблиця 6.1) .

Таблиця 6.1. Пріоритети операцій

|  |  |
| --- | --- |
| Операція | Пріорітет |
| ( { | 0 |
| ) } | 1 |
| || | 2 |
| && | 3 |
| < > <= >= != == | 4 |
| + - | 5 |
| \* / | 6 |

Для побудови ПОЛІЗ введемо такі правила:

1. Iдентифікатори,константи та мітки переходять із входу прямо в ПОЛІЗ.

2. Cимволи із найбільшим пріоритетом, заносяться в стек, нічого не виштовхуючи.

3. Якщо стек порожній, поточна операція заноситься в стек.

4. Якщо пріоритет поточної операції менше пріоритету операції, яка знаходиться на вершині стеку, то всі операції зі стеку заносяться в ПОЛІЗ доки на вершині стеку не зустрінемо оператор з пріоритетом меншим ніж у поточної операції.

Для побудови полізу потрібно перетворити деякі службові слова на конструкції, що можна помістити в поліз та які простіше сприйняти комп’ютеру при виконанні або ж які просто перетворити на асемблер. Для цього було переписано оператори циклу, умовного переходу, оператори вводу та виводу на конструкції, на основі міток та логічних виразів, які будуть слугувати операндами для полізу та операторами умовного та безумовного переходів[2].

Для трансляції в ПОЛІЗ операторів керування – оператора циклу та оператора умовного переходу, введемо в ПОЛІЗ дві додаткові команди:

m goto– безумовний перехід на мітку m;

<лог. вир> m CTBM–якщо логічний вираз хибний, перехід на мітку m.  
 Що ж розглянемо оператор циклу,який був заданий у варіанті.Загальний вигляд:**while ЛВ do А**.Побудуємо,блок-схему циклу(рисунок 6.1).

+

\_

ЛВ

А

Рисунок 6.1 — Блок-схема оператора циклу

Записавши цей алгортим в формі ПОЛІЗ,отримаємо такий результат:

:ПЛВ CTBM ПА goto :,де

ПЛВ=ПОЛІЗ(ЛВ)  
ЛВ- логічний вираз

CTBM-умовний перехід по хибі;

А-тіло циклу

ПА= ПОЛІЗ(А)

Визначимо навантаження на службові слова.

:

CTBM

goto:

А також функціональне навантаження на службові слова:

* **while**-генерує мітку , записує в стек while;генерує в ПОЛІЗ : ;
* **do**-очищує стек до while виключно,якщо знаходить в стеку while,генерує робочу мітку і заносить в ПОЛІЗ CTBM;
* **;**-очищує стек до while виключно,якщо знаходить в стек while,генерує в ПОЛІЗ goto і видаляє з стеку while;

Розглянемо оператор умовного переходу ,який був заданий у варіанті.Загальний вигляд: **if ЛВ ? А : D** . Побудуємо,блок-схему умовного переходу (рисунок 6.2).

Рисунок 6.2 — Блок-схема оператора умовного переходу

ЛВ

Так

А

В

Записавши цей алгортим в формі ПОЛІЗ,отримаємо такий результат:

ПЛВ CTBM ПА goto: ПВ :,де

ПЛВ=ПОЛІЗ(ЛВ);  
ЛВ- логічний вираз;

CTBM-умовний перехід по хибі;

А-тіло умовного оператора ;

В-тіло умовного оператора ;

ПВ= ПОЛІЗ(В);

ПА= ПОЛІЗ(А);

Визначимо навантаження на службові слова.

;

goto:

:

Функціональне навантаження на службові слова:

* If - записує в стек if ;
* ?- очищує стек до if виключно.При цьому генерується нова мітка і ПОЛІЗ генерується . Потім мітка записується в таблицю міток і дописується в вершину стека,так що в вершині стека буде .
* :- виштовхує зі стека всі знаки до першого if виключно,в ПОЛІЗ генерується goto,і в стек записується ,заносим значення мітки таблицю міток
* ;- виштовхує зі стека все до першого , генерує в ПОЛІЗ і видаляє з стеку

Опишемо також оператори ввода/ввиводу.

Загальний вигляд операторів:

read(a,b)

write(a,b)

Розклад операторів(відповідно):

a b 2 RD  
a b 2WT

Функціональне навантаження на службові слова:

* **read/write**-генерує ознаку ввода/виводу true,присвоює лічильник вводу/виводу в 0;
* **(** - має пріоритет 0,заноситься в стек;
* **,** - нарощує лічильник,якшо ознака вводу/виводу true;
* **)** - якщо ознака вводу/виводу true,то збільшує лічильник ввода/виводу на 1,і генерує в ПОЛІЗ лічильник ввода/ввиводу по ознаці ввода/виводу генерує RD чи WT,дивлячись яка ознака true,після цього ознаку вводу/виводу присвоює false;

Після задання всіх можливих службових слів,наведемо результат згенерованого ПОЛІЗу(рисунок 6.3).

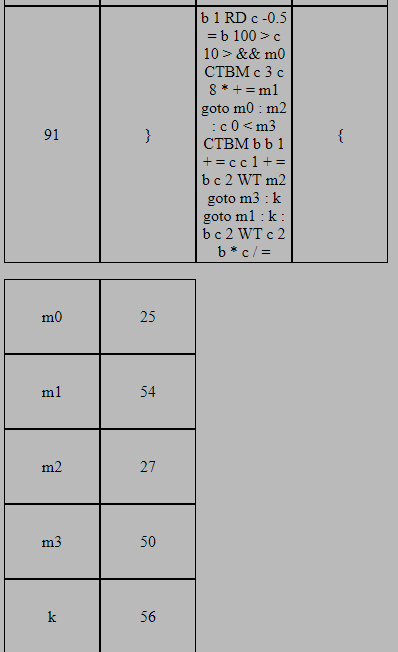


Рисунок 6.3 — Приклад згенерованого ПОЛІЗ та таблиця міток

## 6.2 Виконання ПОЛІЗ

Виконання ПОЛІЗ відбувається в модулі виконання ПОЛІЗу.В даному модулі,відбувається прохід по кожному символу ПОЛІЗу Усі арифметичні вирази використовують додатковий стек для проміжних обчислень. Головна функція аналізує вхідний символ по додатковому полю,який вказує на приналежність до певного класу,та перенаправляє його до функцій,які в свою чергу,виконують необхідні дії.

Загалом виділяються функційї,для:

* + 1. Індефікатор/константа(функція додає символ в стек).
    2. Мітка(функція яка аналізує наступний символ,і відповідно до нього виконується певні дії).
    3. Арифметичний знак(функція виконує арифметичну операцію над двома чи однією останніми комірками стеку, та записує результат у комірку пам’яті останнього операнда ).
    4. Логічний знак(функція виконує логічні операцію над двома останніми комірками стеку, та записує результат,у вигляді булевської змінної ,в комірку пам’яті останнього операнда ).
    5. Вивода(функція виводить в консоль додатку,змінні,кількість яких вказується в останній комірці,а самі змінні знаходяться в наступних)
    6. Ввода(функція видає модальне вікно,в яке потрібно,ввести значення зазначених змінних).

## 6.3 Приклад роботи модуля виконання ПОЛІЗу

Наведемо приклади покрокового виконання ПОЛІЗу у вигдялі таблиці (рисунок 6.3.1).

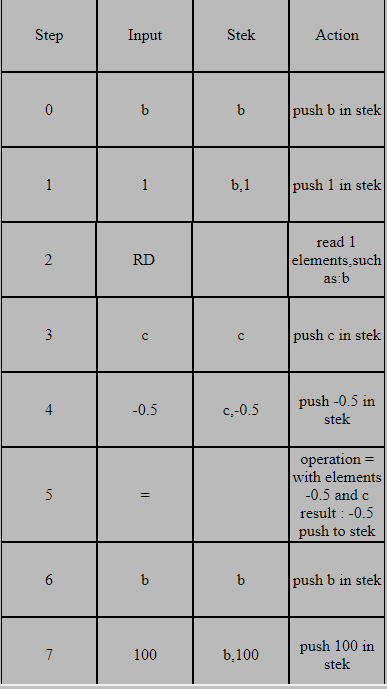


Рисунок 6.3.1 — Приклад покрокового виконання ПОЛІЗу

В інтерфейсі користувача додатку,продемонструємо вивід(рисунок 6.3.2) та ввід змінних(рисунок 6.3.3).



Рисунок 6.3.2 — Вивід змінних



Рисунок 6.3.3 — Ввід змінних

# ВИСНОВКИ

В даній курсовій роботі було розроблено додаток,для інтерпритації розробленої мови. Додаток включає незалежні блоки : лексичний аналізатор,синтаксичний аналізатор та генератор коду,модуль виконання ПОЛІЗу та графічний інтерфейс. Графічний інтерфейс надає можливість зручного набору вхідного тексту,так як константи і резервні слова,мають свій особливий синтаксис . Що запобігає різного роду помилкам,тому даний додаток,буде корисним,тим хто тільки починає вчити основи програмування. Також додаток,може використовуватися,як програму для обчислення складних,арифметичних операцій.

Виконавши курсову роботу,я освоїв методи та принципи побудови трансляторів,також були удосконаленні навички в програмуванні алгоритмів.

Програмне забезпечення було реалізовано за допомогою Node.js v9.7.1 та nw.js v0.28.3,а також ES6,html5 і css3.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Медведєва, В.М. Транслятори: внутрішнє подання програм та інтерпретація: Навчальний посібник / В.М. Медведєва, В.А. Третяк. – К.: НТУУ "КПІ", 2015. – 144 с.
2. Медведєва, В.М. Транслятори: лексичний та синтаксичний аналізатори / В.М. Медведєва, В.А. Третяк. – К.: НТУУ "КПІ", 2012. – 148 с.
3. Методичне забезпечення навчальних курсів з основ розробки трансляторів [Електронний ресурс] : [Веб-сайт].– Режим доступу: http://www.softcraft.ru/translat (дата звернення 20.05.2018).
4. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. - М.: Мир, 1978.
5. Льюис Ф., Розенкранц Д., Стринз Р. Теоретические основы проектирования компиляторов. - М.: Мир, 1979.
6. Рейоурд-Смит В.Дж. Теория формальных языков. Вводный курс. - М.: Радио и связь, 1988
7. Ceкpeты JavaScript ниндзя: Пер. с англ . -М.: ООО "И.Д. Вильяме", 2015. - 416 с.: ил. - Парал. тит. англ.

# ДОДАТОК 1

Розробка компілятора мови загального призначення

Опис програмного коду

УКР.НТУУ “КПІ”. **ТЕФ\_АПЕПС\_ТР\_52**

Листів 10

2018

Код модуля виконання ПОЛІЗу

// variables

let finalStek=[];

let finalPoliz=stekObj.filter(()=>1).reverse();

let finalPolizLenght=finalPoliz.length;

let action='';

// // used func for lab8

function calculetePolizSecondWay(){

let kk=0;

let iterator=0;

tableLab8.innerHTML+=(`<div>

<p class="lab1">Step</p>

<p class="lab1">Input</p>

<p class="lab1">Stek</p>

<p class="lab1">Action</p

</div> `);

while(iterator<finalPolizLenght){

//action='';

let last = finalPoliz[finalPolizLenght-iterator-1];

if(last.lex=='id'||last.lex=='con'){

finalStek.push(last);

action=`push ${last.value} in stek`

}

else if (last.lex=='lab'||last.lex=='auxiliry lab') {

iterator=calcLabelProces(last,iterator);

if(kk>10000){

writeInBlock(consol,`${new Error('loop,dont work\'s ')}`)

throw new Error('loop,dont work\'s ');

break;

}

}

else if(last.lex=='sign') {

calcWithSign(last);

}

else if(last.lex=='logic sign'){

calcWithLogicSign(last);

}

else if(last.lex=='WT'){

calcWriteVariables();

}

else if(last.lex =='RD'){

calcReadVariables();

}

else{

writeInBlock(consol,`${new Error('Undefined'+last.value)}`)

throw new Error('Undefined '+last.value)

}

console.log('next step')

tableLab8.innerHTML+=(`<div>

<p class="lab1">${kk}</p>

<p class="lab1"> ${last.value}</p>

<p class="lab1">${display()}</p>

<p class="lab1">${action}</p>

</div> `);

iterator++;

kk++;

}

};

function calcReadVariables(){

let readCount=+finalStek.pop().value;

action =`read ${readCount} elements,such as:`;

for(let i=0;i<readCount;i++){

let readablevariable=finalStek.pop().value;

let temp = prompt(`Enter a variable: ${readablevariable}`);

tableID.set(readablevariable,temp) ;

action+=`${readablevariable} `

}

}

function calcWriteVariables(){

let writeCount=finalStek.pop().value;

let writablevariable;

action=`write ${writeCount} elements,such as:`;

for(let i=0;i<writeCount;i++){

writablevariable=finalStek.pop();

writeInBlock(consol,`${writablevariable.value} = ${tableID.get(writablevariable.value)}`);

action+=`${writablevariable.value} `

}

}

function calcWithSign(last){

let first=finalStek.pop().value;

let second=finalStek.pop().value;

let temp={lex:'con'};

if(last.value == '='){

tableID.set(second,first);

temp.value=tableID.get(second);

}

else{

second=tableID.has(second) ? +tableID.get(second) :+second;

first=tableID.has(first) ? +tableID.get(first) :+first;

if(last.value == '+'){

temp.value=+(second+first);

finalStek.push(temp);

}

else if(last.value == '-'){

temp.value=+(second-first);

finalStek.push(temp);

}

else if(last.value == '\*'){

temp.value=+(second\*first);

finalStek.push(temp);

}

else if(last.value == '/') {

temp.value=+(second/first);

finalStek.push(temp);;

}

else {

writeInBlock(consol,`${new Error('undefined sign'+last.value)}`);

throw new Error('undefined sign'+last.value);

}

}

action=`operation ${last.value} with elements

${first} and ${second}

result : ${temp.value} push to stek`;

};

function calcWithLogicSign(last){

let first=finalStek.pop().value;

let second=finalStek.pop().value;

let temp={lex:'bool'}

if(last.value == '&&') {

temp.value=(second && first);

finalStek.push(temp);

}

else if(last.value == '||') {

temp.value=(second || first);

finalStek.push(temp);

}

else{

second=tableID.has(second) ? +tableID.get(second) :+second;

first=tableID.has(first) ? +tableID.get(first) :+first;

if(last.value == '=='){

temp.value=(second == first);

finalStek.push(temp);

}

else if(last.value == '<'){

temp.value=(second < first);

finalStek.push(temp);

}

else if(last.value == '<='){

temp.value=(second <= first);

finalStek.push(temp);

}

else if(last.value == '>'){

temp.value=(second > first);

finalStek.push(temp);

}

else if(last.value == '>=') {

temp.value=(second >= first);

finalStek.push(temp);;

}

else if(last.value == '!=') {

temp.value=(second != first);

finalStek.push(temp);;

}

else {

writeInBlock(consol,`${new Error('undefined sign'+last.value)}`);

throw new Error('undefined sign'+last.value)

}

}

action=`operation ${last.value} with elements

${first} and ${second}

result : ${temp.value} push to stek`;

};

function calcLabelProces(last,iterator){

let temp = finalPoliz[finalPolizLenght-iterator-2];

let count=0;

if(temp.lex =='auxiliry words'){

if(temp.value=="goto"){

return count = finalPolizLenght-(finalPolizLenght - labelMap.get(last.value));

action = ` go to ${last.value}`

}

else if (temp.value=="CTBM") {

let loopExpression=finalStek.pop().value;

let move = loopExpression ? iterator+1 : finalPolizLenght-(finalPolizLenght - labelMap.get(last.value));

action = loopExpression ? `loopExpression= ${loopExpression} so go on` :`loopExpression= ${loopExpression} so go to ${last.value}`

return move

}

else{

writeInBlock(consol,`${ new Error(`undefined auxiliry words${temp.value}`)}`);

throw new Error(`Undefined auxiliry words${temp.value}`);

}

}

else if (temp.value==":") {

return ++iterator;

}

else{

writeInBlock(consol,`${ new Error(`undefined auxiliry words${temp.value}`)}`);

throw new Error(`Undefined auxiliry words${temp.value}`);

}

return count;

}

// process POLIZ

let lab8=()=>{

finalStek=[];

finalPoliz=stekObj.filter(()=>1).reverse();

finalPolizLenght=finalPoliz.length;

calculetePolizSecondWay();

};

lab8();