Оглавление

[1.Поняття транслятора, компілятора, інтерпретатора. 2](#_Toc485066542)

[2. Структурна схема транслятора. 3](#_Toc485066543)

[3. Лексичний аналізатор: призначення, структури даних та принцип роботи. 4](#_Toc485066544)

[4. Синтаксичний аналізатор: призначення, структури даних та принцип роботи. 5](#_Toc485066545)

[5. Генератор коду: призначення, структури даних та принцип роботи. 6](#_Toc485066546)

[6. Формальні граматики: визначення, позначення, приклад. 7](#_Toc485066547)

[7. Запис граматик за допомогою БНФ: види правил, приклад. 8](#_Toc485066548)

[8. Визначення понять «породження рядка» та «безпосереднє породженнярядка». 9](#_Toc485066549)

[9. Визначення мови, сентенції (речення) та сентенціальної форми граматики G. 10](#_Toc485066550)

[10.Класифікація мов за Хомським. 11](#_Toc485066551)

[11.Визначення еквівалентності, однозначності та неоднозначності граматик. 12](#_Toc485066552)

[12.Властивості контекстно-вільних граматик. 13](#_Toc485066553)

[13.Лівий та правий вивід (розбір) при синтаксичному аналізі. 14](#_Toc485066554)

[14.Поняття низхідного синтаксичного аналізу. Приклад. 15](#_Toc485066555)

[15.Поняття висхідного синтаксичного аналізу. Приклад. 16](#_Toc485066556)

[16.Визначення регулярного виразу та операції над регулярними виразами. 17](#_Toc485066557)

[17.Регулярні визначення та додаткові позначення в регулярних виразах. 18](#_Toc485066558)

[18.Структура та побудова лексичного аналізатора. Граф автомату лексичного аналізатора. 19](#_Toc485066559)

[19.Стратегії синтаксичного розбору та типи синтаксичних аналізаторів. 20](#_Toc485066560)

[20.Визначення LL(k), LR(k), RL(k), RR(k) граматик. 21](#_Toc485066561)

[21.Алгоритм синтаксичного розбору за методом рекурсивного спуску. 22](#_Toc485066562)

[22.Алгоритм синтаксичного розбору за методом граматик передування. 23](#_Toc485066563)

[23.Способи визначення семантики мов програмування. Метасемантичні мови. 24](#_Toc485066564)

[24.Проста метасемантична мова. Приклад. 25](#_Toc485066565)

[25.Генерація коду для оператора присвоювання. 25](#_Toc485066566)

[26.Генерація коду для умовного оператора if-then-else. 25](#_Toc485066567)

[27.Генерація коду для оператора циклу while. 25](#_Toc485066568)

[28.Генерація коду для оператора циклу repeat. 26](#_Toc485066569)

[29.Генерація коду для оператора циклу for. 27](#_Toc485066570)

[30.Генерація коду для оператора вибору case. 28](#_Toc485066571)

# 1.Поняття транслятора, компілятора, інтерпретатора.

**Транслятор** – це програма чи комплекс програм, що здійснюють переклад тексту,

написаного однією мовою програмування (вхідна мова), в текст, поданий іншою мовою

(вихідна мова).

Розрізняють транслятори двох видів: 1) компілюючого типу; 2) інтерпретуючого типу;

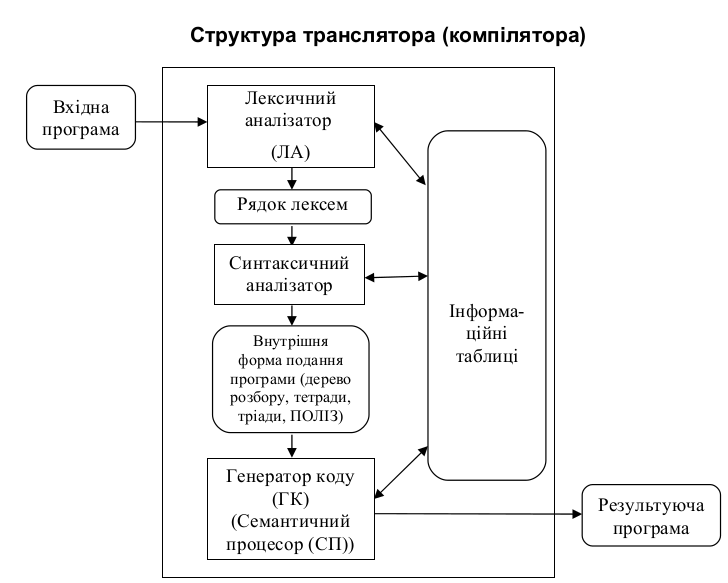
**Компілятор** – це транслятор, для якого вхідною мовою є мова високого рівня (наприклад

C, Pascal, Algol, та інші), а вихідною – мова асемблера чи мова машинних команд. До того

ж переклад вхідної програми вихідною мовою виконується одразу цілком.Варто зауважити, що вхідною та вихідною програмами компілятора завжди є текст.

**Інтерпретатор** – це транслятор, що здійснює пооператорний переклад тексту програми вихідною мовою з одночасним виконанням цих операторів. На виході інтерпретатора отримуємо результат роботи вхідної програми.

# 2. Структурна схема транслятора.



# 3. Лексичний аналізатор: призначення, структури даних та принцип роботи.

**Лексичний аналізатор** здійснює перетворення вхідного тексту програми, що подана рядком символів, в рядок лексем, поданий в цифровій формі, а також знаходить лексичні помилки.

**Принцип роботи** лексичного аналізатора:

1. Зчитування потоку символів з тексту вхідної програми.

2. Видалення з тексту «зайвих» символів, які не входять до складу токенів – пробіл, символи табуляції та нового рядка, тощо.

3. Групування отриманих символів у лексеми, що включає визначення границь кожної лексеми.

4. Виявлення та повідомлення про помилку, якщо лексема неправильна.

5. Формування і заповнення таблиць лексем та ідентифікаторів для їх наступної передачі синтаксичному аналізатору

**Лексема** – найменша одиниця інформації, яка обробляється синтаксичним аналізатором.

Приклади лексем:

1) односимвольні роздільники (, ; .), знаки операцій (+, -, \*, / тощо),

що є односимвольними роздільниками;

2) багатосимвольні роздільники (<=, <>, тощо);

3) ідентифікатори;

4) константи;

5) ключові слова (for, while та інші).

# 4. Синтаксичний аналізатор: призначення, структури даних та принцип роботи.

**Синтаксичний аналізатор** (parser, парсер) здійснює декомпозицію вхідної програми,

поданої рядком лексем, у структурні одиниці мови (оператори, описи, декларації тощо)

згідно граматики вхідної мови, а також знаходить синтаксичні помилки.

На етапі синтаксичного аналізу виконується перевірка синтаксичної коректності вихідної програми, представленої у вигляді потоку токенів і сукупності таблиць, і перетворення її в деяку внутрішню форму (найчастіше дерево розбору), зручну в подальшому для генерації об'єктного коду.

**Дерево розбору** є однією із внутрішніх форм подання вхідної програми, що містить

структурні одиниці мови, а також зв’язки між ними.

# 5. Генератор коду: призначення, структури даних та принцип роботи.

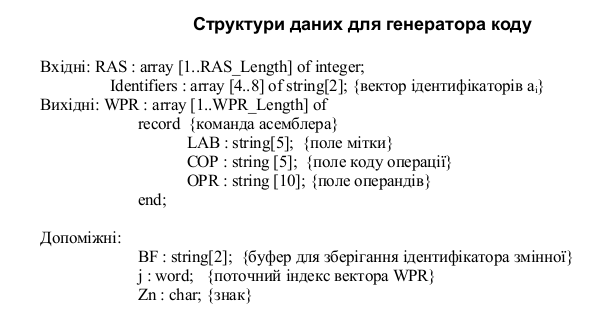
**Призначення:**

Генератор коду (семантичний процесор) перетворює вхідну програму, подану одною із

внутрішніх форм, в текст (оператори, команди) вихідною мовою на основі семантики

вхідної мови.

**Структури даних:**



**ПРО RAS:**

У внутрішньому поданні дерево розбору часто подається в вигляді цілочисельного

вектора. Елементи такого вектора мають наступний зміст:

– додатне ціле число є номером правила;

– від’ємне ціле число є посиланням (індексом елемента вектора, в якому розміщується

номер потрібного правила).

Подання дерева розбору у вигляді числового вектора називається лінійною формою

подання дерева розбору.

**Принцип роботи:**

*\* Структура програми генератора коду (семантичного процесора)*

Program CodeGen;

const <визначення границь вхідних/вихідних структур даних>

var RAS : array[1..LR] of integer;

<опис вихідних і допоміжних даних>

<опис допоміжних процедур>

<опис процедур SPR>

begin

<початкові установки>

<ввід вектора RAS>

SPR(LRAS);

<вивід результату>

end.

RAS - лінійна формою подання дерева розбору з ЛЕКЦІЇ 13

LRAS - довжина RAS

*\* Структура рекурсивної семантичної процедури*

Розглянемо структуру рекурсивної семантичної процедури, яка виконує обхід дерева

розбору, представленого в лінійній формі з посиланнями.

Procedure SPR (u : integer); (\*u – індекс правила чи посилання в векторі RAS\*)

var

k, (\*уточнений індекс вибраного правила в векторі RAS\*)

i, (\*номер вибраного синтаксичного правила\*)

k1, k2, ..., kn : integer; (\*допоміжні змінні\*)

begin

if RAS[u] < 0 then k := – RAS[u]

else k := u;

i := RAS[k];

case i of

1: (\*текст синтаксичного правила 1\*)

<семантичне визначення 1>;

2: (\*текст синтаксичного правила 2\*)

< семантичне визначення 2>;

...

n: (\*текст синтаксичного правила n\*)

< семантичне визначення n>;

end; (\*case\*)

end; (\*SPR\*)

# 6. Формальні граматики: визначення, позначення, приклад.

**Алфавіт мови (T)** – попередньо визначена довільна непуста скінченна множина символів

(так званих термінальних символів) для побудови речень (рядків) мови.

T \* – множина всіх рядків алфавіту Т (тобто тих, що складаються лише із термінальних

символів), включаючи порожній рядок ε.

T + – множина всіх рядків в алфавіті Т, не включаючи порожній рядок ε.

T \* = T + ∪ ε.

Формальна граматика G є сукупністю чотирьох понять G = (T, N, P, S).

Т – алфавіт мови;

N – множина нетермінальних символів граматики.

V=T∪N – це об’єднання множин термінальних і нетермінальних символів.

P – множина правил граматики виду α→β, ‘→ ‘ означає «є за визначенням».

S – аксіома граматики, один із нетермінальних символів граматики (S ∈ N), з якого

починається опис граматики.

Приклад. Розглянемо граматику, що породжує цілі двійкові константи.

Множина правил Р:

1. <рядок> → <двійковий рядок>

2. <двійковий рядок> → <цифра>

3. <двійковий рядок> → <двійковий рядок><цифра> – рекурсивне правило Р

4. <цифра> → 0

5. <цифра> → 1

T = {0, 1}

N = {<рядок>, <двійковий рядок >, <цифра>}

S = <рядок>

Правило називається рекурсивним, якщо один і той самий нетермінальний символ

знаходиться зліва і справа від стрілки.

# 7. Запис граматик за допомогою БНФ: види правил, приклад.

Нота́ція Бе́куса—Нау́ра (англ. Backus-Naur form, BNF) — це спосіб запису правил контекстно-вільної граматики.

БНФ визначає скінченну кількість символів (нетерміналів), а також правила заміни символу на якусь послідовність букв (терміналів) і символів. Спочатку є один нетермінал (нетермінали записуються в кутових дужках). Потім він замінюється на деяку послідовність терміналів та нетерміналів. Потім процес повторюється, доки не вийде ланцюжок , що складається лише з терміналів.

Послідовність складена з терміналів, нетерміналів, називається метавиразом.

Наприклад: <ім’я> ’:=’ <вираз>.

Для визначення структури нетерміналу у правилі граматики використовується ‘::=’. Наприклад: <оператор присвоєння> ::= <ім’я> ’:=’ <вираз>.

Синтаксичні правила, записані у вигляді <поняття> ::= <метавираз> називаються формами Бе́куса—Нау́ра (БНФ). Поняття записане праворуч від ʼ::=ʼ називається лівою частиною, а метавираз – правою. Знак ʼ::=ʼ не є символом мови і називається метасимволом.

Для позначення кількох варіантів правої частини використовується символ ʼ|ʼ. Наприклад:

<вираз> ::= <число> | <змінна> | <константа> | <число>’+’<число> | …

Також за допомогою БНФ можна записувати рекурсивні правила. Наприклад:

<число> ::= <цифра> <рядок цифр>

<рядок цифр> ::= <цифра> <рядок цифр> | <пусто>

# 8. Визначення понять «породження рядка» та «безпосереднє породженнярядка».

Кажуть, що рядок α **породжує рядок** β, якщо існує ланцюжок виводів:

α = γ 0 ⇒ γ 1 ⇒ γ 2 ⇒......⇒ γ n = β.

Породження позначається α ⇒+ β, якщо β виводиться з α більш ніж за один крок.

Породження позначається α ⇒\* β, якщо може бути одна з двох ситуацій: α ⇒+ β або α ⇒ β.

Друга ситуація називається **безпосереднім породженням рядка**. Рядок α безпосередньо породжує рядок β (α ⇒ β), якщо α = γ 1 <U> γ 2 , β = γ 1 δ γ 2 , та існує правило

<U> → δ, де α, β, δ, γ 1, γ 2 ∈V \* , <U>∈N, або, можна сказати інакше: рядок β безпосередньо виводиться з рядка α.

# 9. Визначення мови, сентенції (речення) та сентенціальної форми граматики G.

**Сентенцією** або реченням граматики G називається рядок, що складається

лише з термінальних символів і виводиться з аксіоми граматики.

**Сентенціальною формою** граматики G з аксіомою S називається будь-який

рядок термінальних та/або нетермінальних символів, що виводиться з аксіоми граматики.

**Мовою L граматики G** називається множина всіх сентенцій (речень), які

можуть бути породжені граматикою G.

# 10.Класифікація мов за Хомським.

Класифікація формальних мов була вперше викладена в книзі «Синтаксичні структури» у 1957 році. Згідно з цією класифікацією існує чотири типи формальних граматик.

• **Граматика типу 0** має правила вигляду

α → β, де α∈V , β∈V .

До даного типу відносяться всі природні мови.

• **Граматика типу 1** має правила виду

α<u>β → α γ β, де α, γ,β∈V , <u>∈N, α – лівий контекст, β – правий контекст.

Мови, що породжуються граматиками типу 1, називаються контекстно-залежними

мовами (КЗ-мовами) або мовами безпосередніх складових (БС-мовами).

• **Граматика типу 2** має правила вигляду

<u> → α, де <u>∈N, α∈V .

Такі граматики називаються контекстно-вільними граматиками, що породжують

контекстно-вільні мови (КВ-мови).

До КВ-мов відносяться практично всі мови програмування. Граматики типу 2

використовується для побудови синтаксичних аналізаторів.

• **Граматика типу 3** має правила тільки одного з двох наступних видів:

перший: <u> → а

<u> → a <v>

або

другий: <u> → а

<u> → <v>a , де а∈T ,<u>∈N , <v>∈N.

Граматики типу 3 також називають регулярними або автоматними граматиками. Мови, які

породжуються такими граматиками, називаються регулярними мовами.

# 11.Визначення еквівалентності, однозначності та неоднозначності граматик.

**1. Еквівалентність граматик.**

Граматики називаються еквівалентними, якщо вони породжують одну і ту ж саму мову.

Наведені нижче граматики 1 та 2 є еквівалентними:

Граматика 1:

1. <рядок> → <рядок символів>

2. <рядок символів> → а

3. <рядок символів> → <рядок символів>а

Граматика 2:

1. <рядок> → <рядок символів>

2. <рядок символів> → <символ>

3. <рядок символів> → <рядок символів><cимвол>

4. <символ> → а

**2. Однозначність граматики.**

Граматика називається однозначною, якщо для будь-якого речення, породжуваного цією

граматикою, всі можливі схеми його виводу приводять до одного і того ж дерева виводу.

**3. Неоднозначність граматики.**

Граматика називається неоднозначною, якщо для одного і того ж речення, породжуваного

граматикою, існує декілька неспівпадаючих дерев виводу.

# 12.Властивості контекстно-вільних граматик.

**1.** Будь-яку ε-вільну КВ-граматику можна привести до нормальної форми Грейбах з правилами вигляду:

<х> → b α ,

де b ∈ T, α∈ N\*, тобто α - рядок нетерміналів, можливо порожній.

**2.** Будь-яку ε-вільну КВ-граматику можна привести до нормальної форми Хомського з правилами вигляду:

1. <х> → <A><B>,

2. <x> → b,

де <x>,<A>,<B> ∈ N; b ∈ T.

**3.** Для будь-якої КВ-мови L з граматикою G існує ε-вільна КВ-граматика G', така що

L(G') = L(G) \ { ε }.

# 13.Лівий та правий вивід (розбір) при синтаксичному аналізі.

Нехай граматика визначена правилами:

1. <оператор> → <змінна>:= <вираз>

2. <вираз> → <терм>

3. <вираз> → <терм> + <вираз>

4. <терм> → <змінна>

5. <терм> → (<вираз>)

6. <змінна> → a

7.<змінна> → b

8. <змінна> → c

При лівосторонньому розборі на кожному кроці розкривається найлівіший нетермінал, а при правосторонньому – найправіший.

**Для рядка a:= b + c зробимо лівосторонній вивід** (розбір):

<оператор> ⇒ 1 <змінна>:= <вираз> ⇒ 6 a:= <вираз> ⇒ 3

a:= <терм> + <вираз>⇒ 4 a:= <змінна> + <вираз> ⇒ 7 a:= b + <вираз> ⇒ 2 a:= b + <терм>⇒ 4

a:=b + <змінна> ⇒ 8 a:= b + c

**Для цього ж рядка зробимо правосторонній вивід**:

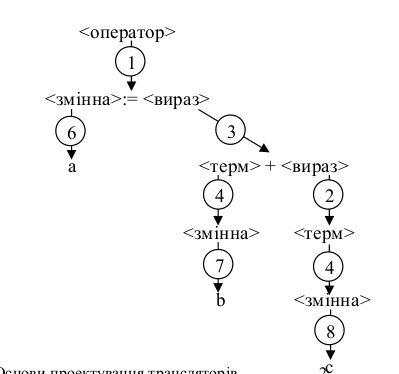
<оператор> ⇒ 1 <змінна>:= <вираз> ⇒ 3 <змінна>:= <терм> + <вираз>⇒ 2

<змінна>:= <терм> + <терм> ⇒ 4 <змінна>:= <терм> + <змінна> ⇒ 8

<змінна>:= <терм> + c ⇒ 4 <змінна>:= <змінна> + c ⇒ 7 <змінна>:= b + c ⇒ 6 a:= b + c

# 14.Поняття низхідного синтаксичного аналізу. Приклад.

Низхідний аналіз – аналіз, при якому рядок терміналів виводиться з аксіоми.

Приклад:

1. <оператор> → <змінна>:= <вираз>

2. <вираз> → <терм>

3. <вираз> → <терм> + <вираз>

4. <терм> → <змінна>

5. <терм> → (<вираз>)

6. <змінна> → a

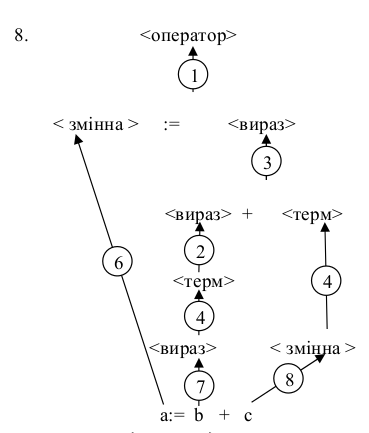
7.<змінна> → b

8. <змінна> → c

# 15.Поняття висхідного синтаксичного аналізу. Приклад.

Висхідний аналіз базується на понятті «основа». Якщо деякий рядок терміналів та/або нетерміналів прямо приводиться до нетерміналу, то він називається безпосередньо приводимою фразою. Найлівіша безпосередньо приводима фраза рядка називається основою. Суть алгоритму висхідного аналізу полягає у пошуку основи й заміни її нетерміналом, до якого вона безпосередньо приводиться – згортка. Розбір завершиться коли цим нетерміналом стане аксіома граматики.

Розглянемо висхідний аналіз (розбір) рядка a:= b + c за такою граматикою:

1. <оператор> → <змінна>:= <вираз>

2. <вираз> → <терм>

3. <вираз> → <вираз> + <терм>

4. <терм> → <змінна>

5. <терм> → (<вираз>)

6. <змінна> → a

7. <змінна> → b

8. <змінна> → c

# 16.Визначення регулярного виразу та операції над регулярними виразами.

Регулярний вираз R задає (визначає) мову L(R). Мови, які можуть бути задані регулярними виразами, називаються регулярними множинами або регулярними (автоматними) мовами.

Регулярний вираз над алфавітом Т визначається наступними правилами:

**1.** ε є регулярним виразом, що визначає множину { ε }, тобто множину, що містить

порожній рядок.

**2.** Якщо а є символом з алфавіту Т, то цей же символ а є простим регулярним виразом,

що визначає множину {a}, тобто множину, що містить рядок а.

**3.** Якщо А і В – регулярні вирази, що визначають мови L(A) і L(B), тоді

1) А | B є регулярним виразом, що визначає мову L(A) U L(B).

2) AB є регулярним виразом, що визначає мову (множину) L(A) L(B), після речення мови А безпосередньо слідує речення мови В.

3) А\* є регулярним виразом, що визначає мову (L(A)) \*.

Регулярний вираз характеризується трьома операціями:

1) альтернатива;

2) конкатенація;

3) ітерація.

Якщо Е 1 і Е 2 – регулярні вирази, тоді:

1. Альтернатива двох регулярних виразів Е 1 і Е 2 позначається: Е = Е 1 | Е 2 або Е = Е 1 + Е 2

2. Конкатенація двох регулярних виразів Е 1 і Е 2 позначається: Е = Е 1 Е 2

3. Ітерація регулярного виразу Е 1 – це багатократне його повторення від 0 до ∞ (можливо

жодного разу) і позначається: Е = E 1 \* = ε | E 1 | E 1 E 1 | E 1 E 1 ...E 1 , або як {E 1 }.

# 17.Регулярні визначення та додаткові позначення в регулярних виразах.

Для зручності запису регулярним виразам можна давати імена і використовувати ці

імена як символи в інших регулярних виразах. Сукупність декількох регулярних виразів, яким були дані імена, називається регулярним визначенням, загальний вид якого такий:

D 1 → R 1

D 2 → R 2

...

D n → R n , де D i – імена регулярних виразів R i .

В регулярних виразах R i можуть використовуватися вже визначені раніше імена D i , тобто кожне R i визначене на множині T U {D 1 , D 2 , ... , D i-1 }

**Додаткові позначення в регулярних виразах**:

**1. Унарний постфіксний оператор +** означає “один або більше екземплярів”.

Якщо R – регулярний вираз, що визначає мову L(R), то R + є регулярним виразом, що визначає мову (L(R)) + . Оператор + має той же пріоритет і асоціативність, що й оператор \*. Мають місце дві тотожності: 1) R\* = R + | ε 2) R + = R R\*

**Приклад:** DIGIT → 0 | 1 | 2 | ... | 9 INTNUM → DIGIT +

**2. Унарний постфіксний оператор ?** означає “один екземпляр або жодного екземпляра”.

Якщо R – регулярний вираз, то R? – регулярний вираз, що описує мову L(R) U {ε}.

**3. Класи символів.**

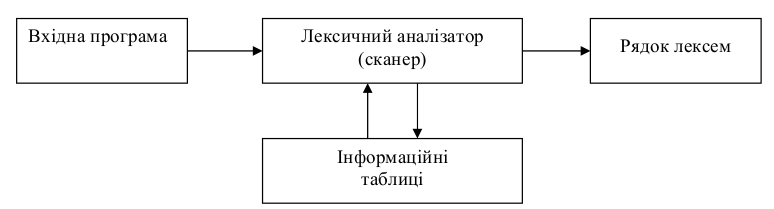
Для скороченого запису деякої множини символів можна використовувати позначення у вигляді класів символів. Клас символів [abc] позначає регулярний вираз a | b | c, тобто [abc] = а | b | с,

де а, b, с – символи алфавіту. Клас символів [a-z] позначає регулярний вираз а | b | .... | z, тобто

[a-z] = a | b | ... |z, де а, b ... z – символи алфавіту.

**Приклад:** Регулярний вираз ідентифікатора з використанням класів символів: [A-Za-z] [A-Za-z0-9]\*

# 18.Структура та побудова лексичного аналізатора. Граф автомату лексичного аналізатора.



Допустимі лексеми, що виділяються лексичним аналізатором (ЛА) даного прикладу:

• Ключові слова; • цілі десяткові константи; • ідентифікатор • одиночні роздільники і знаки операцій;

Крім того, ЛА розпізнає ознаки початку і кінця коментарів (\* текст коментаря\*) і пропускає

текст коментаря без формування лексем.

**Позначення станів автомата ЛА:**

° S – початковий стан;

° INP – стан введення поточного символу програми;

° CNS – стан виділення константи;

° IDN – стан виділення ідентифікатора;

° BCOM – стан визначення символів початоку коментаря;

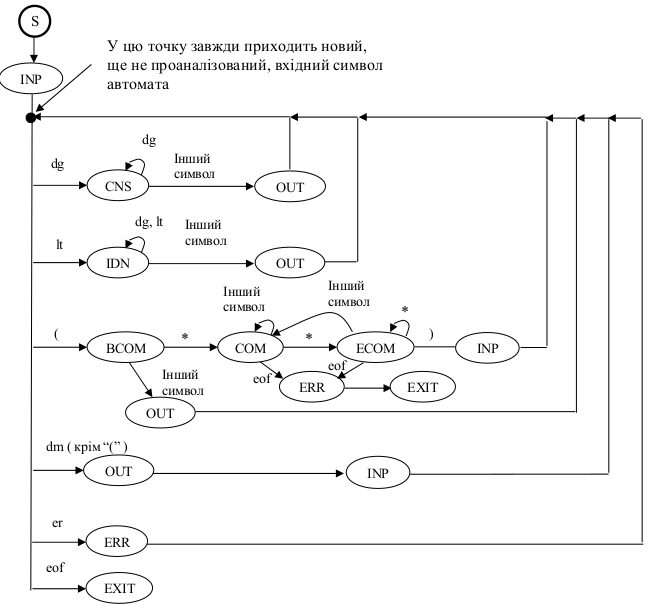
° COM – стан пропуску (видалення) символів коментаря;

° ECOM – стан визначення символів кінеця коментаря;

° ERR – стан обробки помилки та видача повідомлення про помилку;

° OUT – стан виведення лексеми;

° EXIT – кінцевий стан.



dg – цифра (0..9), lt – буква (A .. Z, а .. z), dm – роздільник, er – помилковий символ, eof – символ кінця файлу.

# 19.Стратегії синтаксичного розбору та типи синтаксичних аналізаторів.

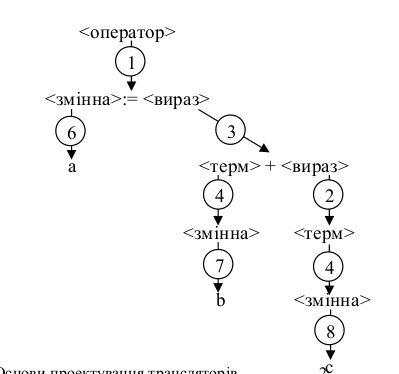
Існує три основних типи синтаксичних аналізаторів, що реалізують три стратегії розбору:

1) стратегія аналізу зверху вниз (низхідний аналіз);

2) стратегія аналізу знизу вверх (висхідний аналіз);

3) змішана стратегія.

При нисхідному аналізі рядок терміналів виводиться з аксіоми граматики.

Приклад:

1. <оператор> → <змінна>:= <вираз>

2. <вираз> → <терм>

3. <вираз> → <терм> + <вираз>

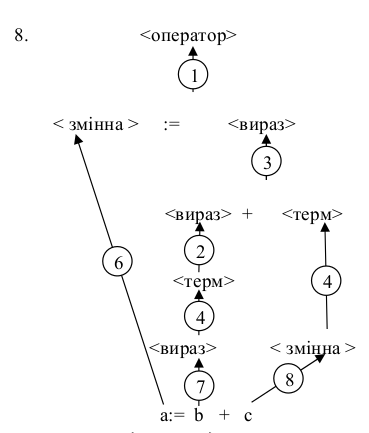
4. <терм> → <змінна>

5. <терм> → (<вираз>)

6. <змінна> → a

7.<змінна> → b

8. <змінна> → c

 Суть алгоритму висхідного аналізу полягає в заміні терміналів та/або нетерміналів нетерміналом, до якого вони зводяться до тих пір, поки все не зведеться до аксіоми граматики.

Є три алгоритми синтаксичного аналізу, що працюють по низхідній стратегії:

1) аналізуюча машина Кнута;

2) алгоритм, що працює по методу рекурсивного спуску;

3) алгоритм таблично-керованого передбачаючого розбору.

Вмсхідну стратегію розбору реалізують алгоритми розбору без повернення, що будуються на основіі спеціальних граматик з певними обмеженнями.

# 20.Визначення LL(k), LR(k), RL(k), RR(k) граматик.

Перша буква L чи R у позначенні вказує напрям перегляду вхідного рядка (зліва направо

чи справа наліво). Друга буква (L чи R) в позначенні означає вид виводу (лівий чи правий відповідно). Буква k в дужках означає кількість символів вхідного рядка, що аналізуються наперед при виконанні розбору.

LL(k)-граматика – це граматика, що допускає безповоротний синтаксичний аналіз при вводі вхідного рядка зліва направо і реалізує лівий вивід із загляданням наперед і аналізом k символів вхідного рядка.

LR(k)-граматика – --- зліва направо --- правий ---

RL(k)-граматика – --- справа наліво --- лівий ---

RR(k)-граматика – --- справа наліво --- правий ---

З використанням LL(k) і RR(k)-граматик реалізується низхідна стратегія синтаксичного розбору, а з використанням LR(k) і RL(k)-граматик – висхідна. На практиці використовуються в основному LL(k) та LR(k)-граматики.

# 21.Алгоритм синтаксичного розбору за методом рекурсивного спуску.

Метод рекурсивного спуску реалізує безповоротний аналіз за рахунок обмежень на

правила граматики:

1) правила не повинні містити лівобічної рекурсії. Якщо такі правила є, то вони замінюються правилами з правосторонньою рекурсією або представляються в ітераційній формі.

2) якщо є декілька правил з однаковою лівою частиною, то права частина повинна починатися з різних термінальних символів (нормальна форма Грейбах).

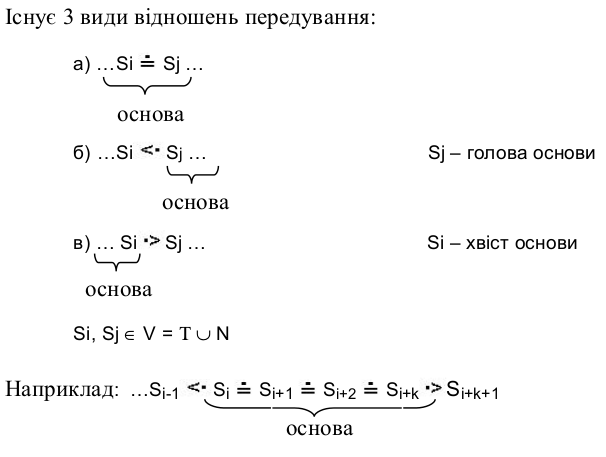
**Суть методу**: кожному нетерміналу граматики ставиться у відповідність окрема процедура

або функція, яка обробляє цей термінал. Якщо в цьому нетерміналі присутні інші нетермінали, то в функції, яка обробляє його, викликаються функції, що обробляють ці нетермінали.

# 22.Алгоритм синтаксичного розбору за методом граматик передування.

Метод граматик простого передування розроблений Віртом і Вебером і є висхідним

аналогом аналізуючої машини Кнута без повернень.



На кожному кроці аналізу шукаємо основу і замінюємо її відповідним нетерміналом, поки не

отримаємо аксіому граматики.

# 23.Способи визначення семантики мов програмування. Метасемантичні мови.

Розрізняють формальну і неформальну семантику.

Формальна семантика (для побудови компіляторів) – це зміст (сенс) синтаксичних правил вхідної мови, що виражені в термінах вихідної мови та в діях по формуванню значень вихідної мови.

Неформальна семантика (для користувача) – це зміст (сенс) речень вхідної мови, що виражений в термінах речень вихідної мови.

В якості метасемантичної мови може бути або спеціалізована метасемантична мова, або

інша мова програмування. Серед відомих метасемантичних мов можна назвати:

1) Дворівневі або w-граматики використовувалися для опису семантики мови на основі використання метаправил, які дозволяють описати контекстно-залежні характеристики мови у вигляді правил КВ-граматики.

2) Система продукцій дозволяє сформувати породжуюче правило, яке визначає семантику на основі контекстно-залежних характеристик мови.

3) Віденська метамова дозволяє описати процедуру, що породжує речення на деякій об’єктній мові абстрактної машини, інтерпретуючи яку можна згенерувати код для конкретного комп’ютера.

4) Метод атрибутних граматик оснований на тому, що правила КВ-мови доповнюються атрибутами, здійснивши обчислення яких можна отримати вихідну програму. Атрибути приписуються нетерміналам і можуть бути успадкованими або синтезованими.

# 24.Проста метасемантична мова. Приклад.

В якості метасемантичної мови може бути або спеціалізована метасемантична мова, або

інша мова програмування. Серед відомих метасемантичних мов можна назвати

**Проста метасемантична мова** – це множина семантичних визначень, які описуються

трьома простими видами об’єктів і які ставляться у відповідність кожному синтаксичному

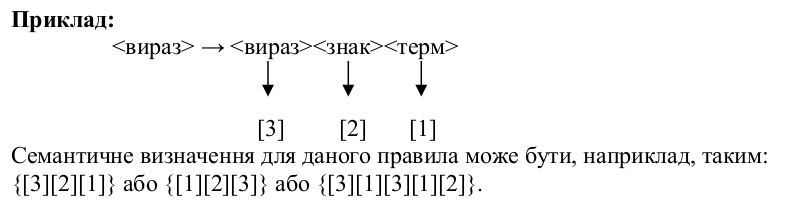
правилу.

**Семантичне визначення** – це опис відповідності між кожним правилом (іноді

конструкцією) вхідної мови та конструкцією (-ями) (фрагментом коду) вихідної мови, а

також опис дій, що виконують семантичний аналіз та генерацію коду, для даного правила

граматики.

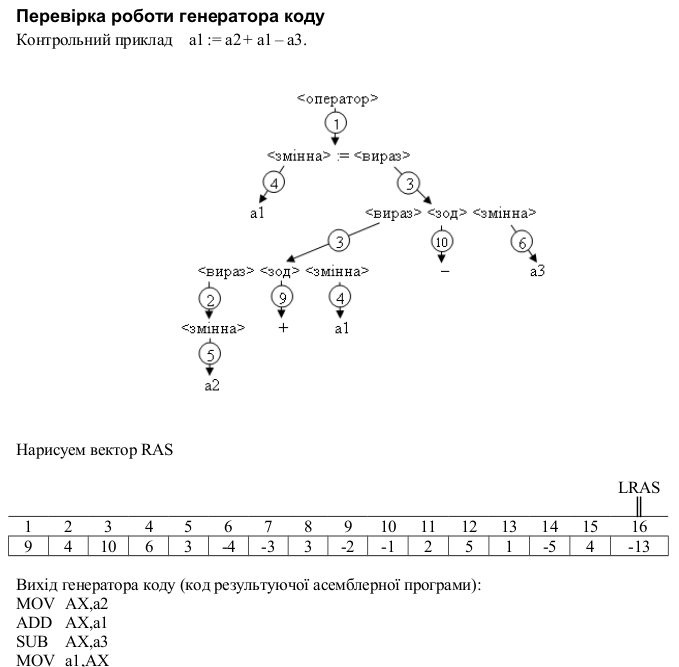


Порядок слідування клауз (посилань для спуску по дереву) в семантичному визначенні

задає порядок розбору.

# 25.Генерація коду для оператора присвоювання.

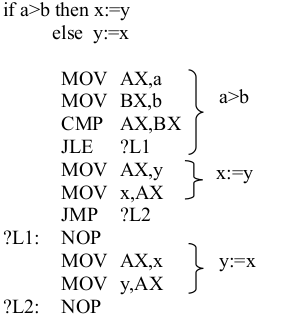
(Лекція 15, если что.)

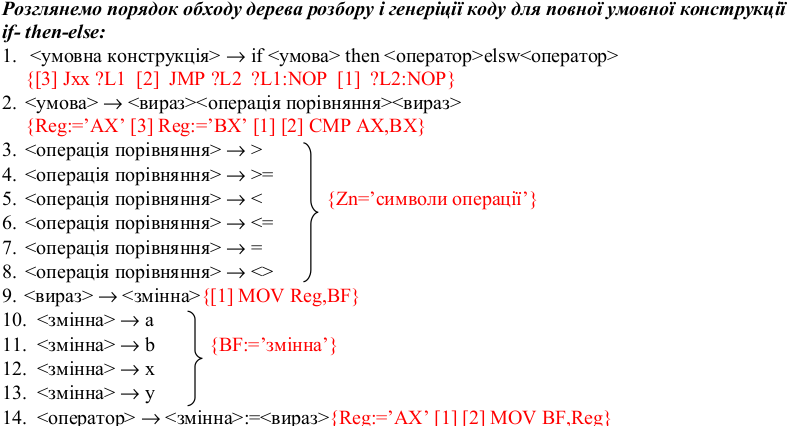


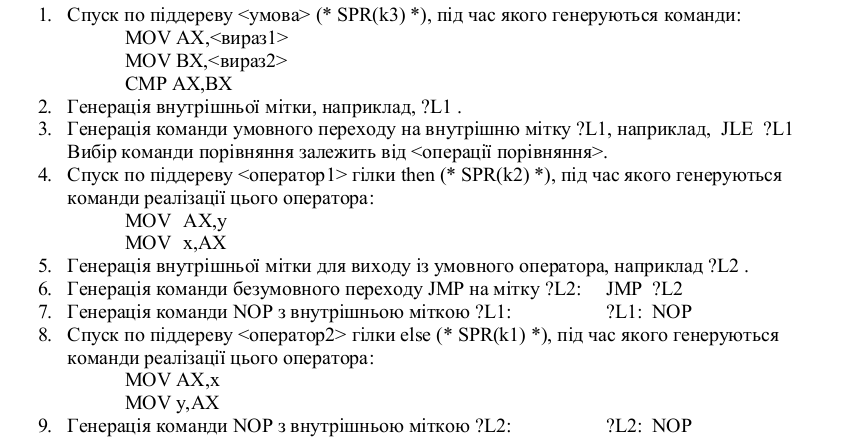
# 26.Генерація коду для умовного оператора if-then-else.

( Лекція 16, если что)

**Приклад:**







# 27.Генерація коду для оператора циклу while.

**Синтаксис:**

1. <цикл-while> → while <умова> do <оператор>

2. <умова> → <вираз1><операція порівняння><вираз2>

3. <операція порівняння> → >

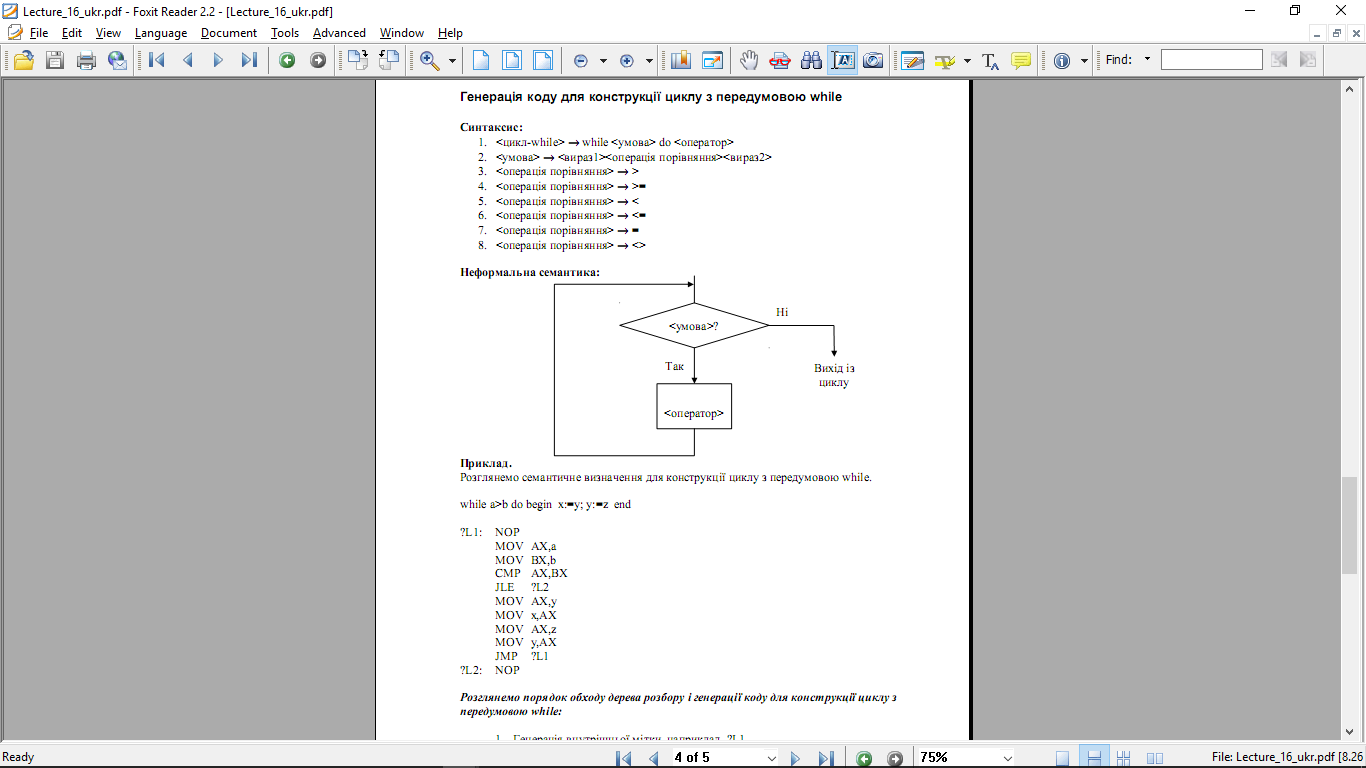
4. <операція порівняння> → >=

5. <операція порівняння> → <

6. <операція порівняння> → <=

7. <операція порівняння> → =

8. <операція порівняння> → <>



**Розглянемо порядок обходу дерева розбору і генерації коду для конструкції циклу**

**передумовою while:**

1. Генерація внутрішньої мітки, наприклад, ?L1 .

2. Генерація команди NOP з внутрішньою міткою ?L1: ?L1: NOP

3. Спуск по піддереву <умова> (\* SPR(k2) \*), під час якого генеруються ком

обчислення цієї умови:

4. Генерація внутрішньої мітки для виходу із циклу, наприклад, ?L2 .

5. Генерація команди умовного переходу на внутрішню мітку ?L2 за умовою,

оберненою заданій, наприклад, JLE ?L2.

6. Спуск по піддереву <оператор> (\* SPR(k1) \*), під час якого генеруються команди

реалізації цього оператора:

7. Генерація команди безумовного переходу JMP на початок циклу, тобто на мітку ?

L1:

JMP ?L1

8. Генерація команди NOP з внутрішньою міткою ?L2: ?L2: NOP

# 28.Генерація коду для оператора циклу repeat.

**Синтаксис:**

1. <цикл repeat> → repeat <список операторів> until <умова>

2. <умова> → <вираз><операція порівняння><вираз>

3. <список операторов> → <оператор>

4. <список операторов> → <список операторов>;<оператор>

5. < операція порівняння > → >

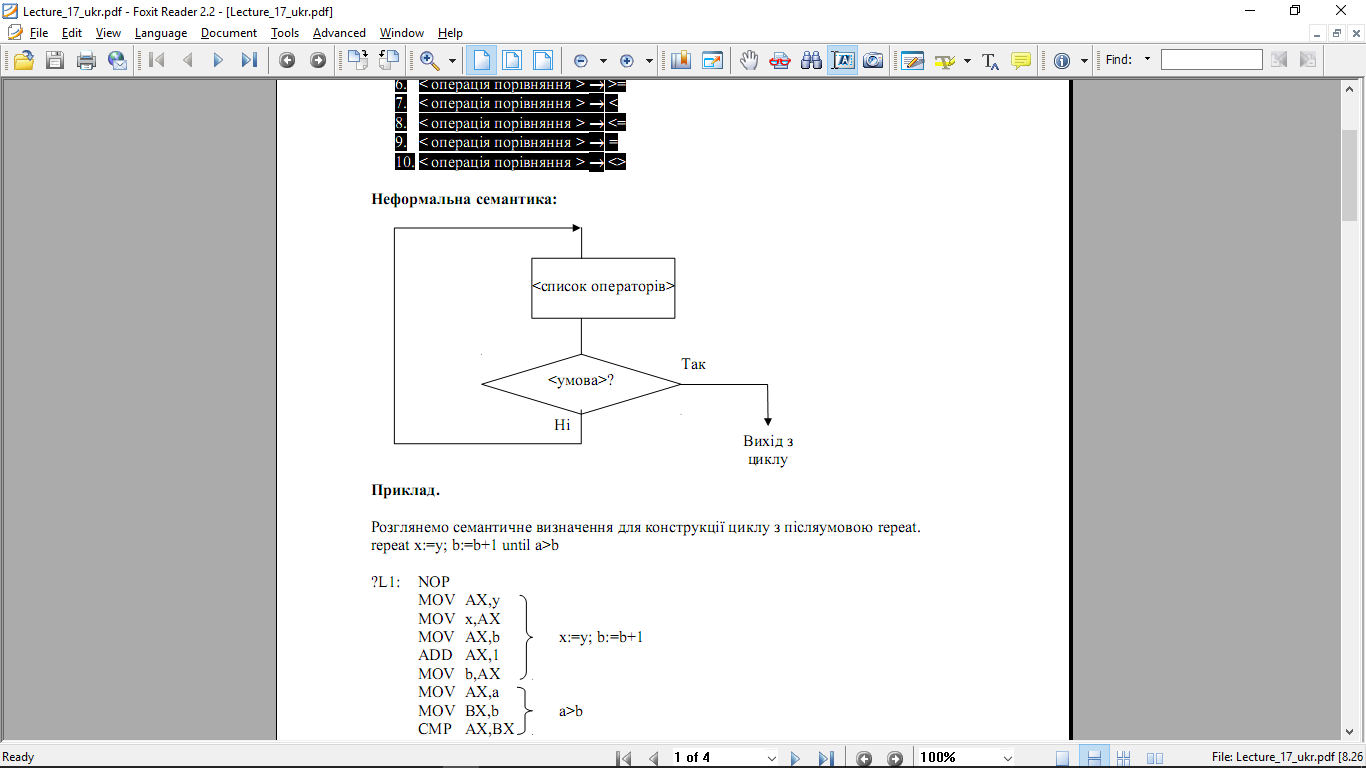
6. < операція порівняння > → >=

7. < операція порівняння > → <

8. < операція порівняння > → <=

9. < операція порівняння > → =

10. < операція порівняння > → <>



**Розглянемо порядок обходу дерева розбору і генерації коду для конструкції циклу з**

**післяумовою repeat:**

1. Генерація внутрішньої мітки, наприклад ?L1 .

2. Генерація команди NOP з внутрішньою міткою початку циклу ?L1: ?L1: NOP

3. Спуск по піддереву <список операторів> (\* SPR(k2) \*), під час якого генеруються

команди реалізації цього списку операторів:

4. Спуск по піддереву <умова> (\* SPR(k1) \*), під час якого генеруються команди реалізації

цієї умови:

5. Генерація команди умовного переходу з внутрішньою міткою ?L1 на початок циклу:

JLE ?L1

# 29.Генерація коду для оператора циклу for.

**Синтаксис:**

1. <цикл for> → for <змінна>:= <нижня межа> to <верхня межа> by <крок> do

<оператор>

2. <нижня межа> → <вираз>

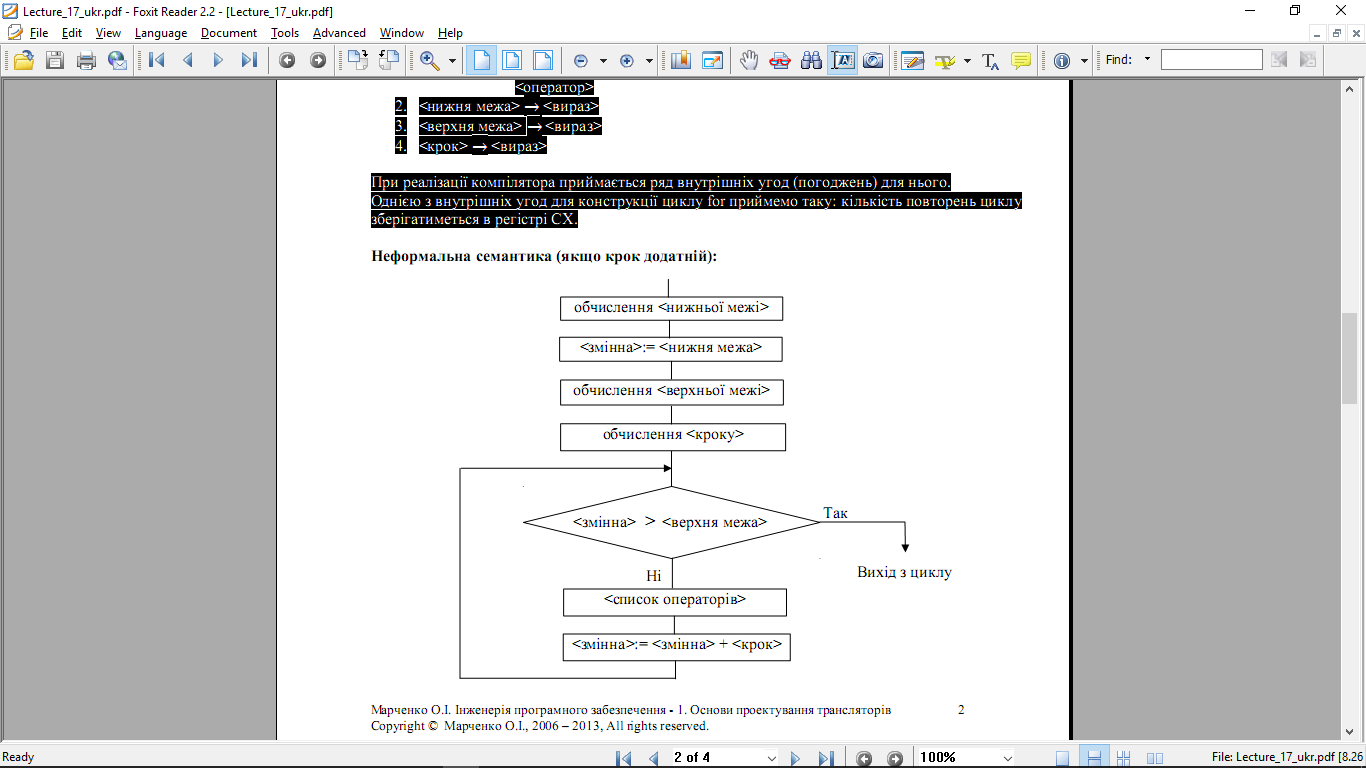
3. <верхня межа> → <вираз>

4. <крок> → <вираз>

При реалізації компілятора приймається ряд внутрішніх угод (погоджень) для нього.

Однією з внутрішніх угод для конструкції циклу for приймемо таку: кількість повторень циклу

зберігатиметься в регістрі СХ.



**Розглянемо порядок обходу дерева розбору і генерації коду для конструкції циклу з**

**лічильником (параметром) for:**

1. Спуск по піддереву <нижня межа> (\* SPR(k4) \*)

2. Спуск по піддереву <змінна> (\* SPR(k5) \*)

3. Генерація команди пересилки <нижня межа> → <змінна>

4. Спуск по піддереву <верхня межа>, (\* SPR(k3) \*)

5. Генерація команди завантаження <верхньої межі>

6. Генерація команди порівняння і команди умовного переходу з внутрішньою міткою ?

L1 для виходу з циклу

7. Спуск по піддереву <крок> (\* SPR(k2) \*)

8. Генерація команд пересилки <кроку> в робочу комірку

9. Генерація команд розрахунку числа повторень циклу

10. Пересилка числа повторень в СX

11. Генерація команди NOP з внутрішньою міткою ?L2 (початок операторів циклу)

12. Спуск по піддереву <оператор>, де генеруються команди його реалізації (\* SPR(k1) \*)

13. Генерація команд зміни значення змінної циклу на величину <кроку>

14. Генерація команди LOOP з переходом на початок циклу

15. Генерація команди NOP з міткою виходу з циклу.

# 30.Генерація коду для оператора вибору case.

**Синтаксис:**

1. <конструкція case> → case <вираз> of <список альтернатив> <альтернатива

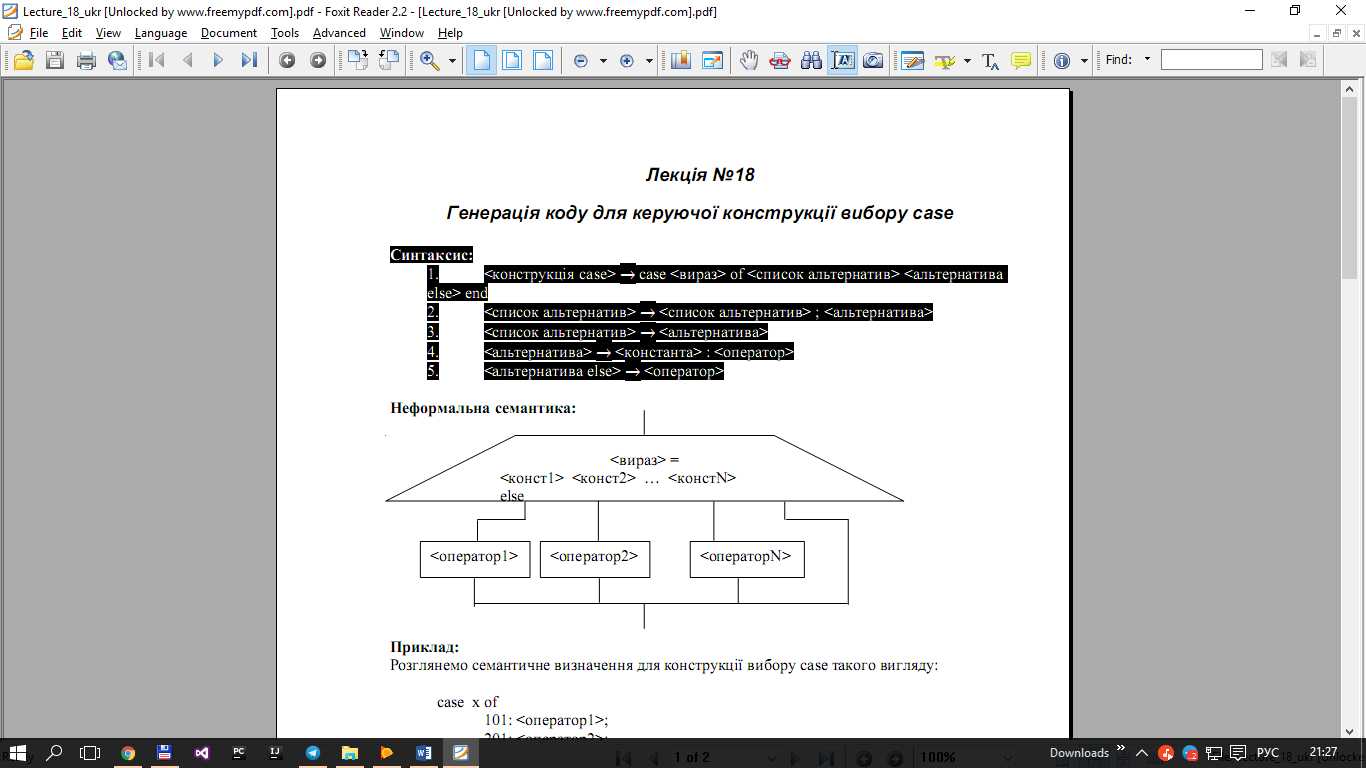
else> end

2. <список альтернатив> → <список альтернатив> ; <альтернатива>

3. <список альтернатив> → <альтернатива>

4. <альтернатива> → <константа> : <оператор>

5. <альтернатива else> → <оператор>



**Розглянемо порядок обходу дерева розбору і генерації коду для**

**конструкції вибору case:**

1. Спуск по піддереву <вираз> (\* SPR (k2) \*), під час якого генеруються

команди обчислення цього виразу (результат записується у AX)

2. Перший прохід по піддереву <список альтернатив> (\* Fl:=true; SPR (k1) \*).

Під час першого проходу визначається кількість альтернатив і формується

масив CaseConst.

3. Цикл генерації команд порівняння значення <виразу> з елементами масиву

CaseConst

4. Генерація команди переходу на оператор, наступний за case оператором:

5. Другий прохід по піддереву <список альтернатив> (\* Fl:=false; SPR (k1) \*).

Під час другого проходу виконується генерація команд <оператора> для

кожної альтернативи і подальша генерація команди JMP ?L1 для переходу

на оператор, наступний за case оператором:

6. Генерація команди NOP, з внутрішньою міткою ?L1: