ЛАБОРАТОРНА РОБОТА №5.
РОЗРОБКА СИНТАКСИЧНОГО АНАЛІЗАТОРА (4 год)

Мета: оволодіти навиками побудови синтаксичних аналізаторів. Навчитись визначати граматику мови програмування у термінах розширеної нотації Бекуса-Наура.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Кожна мова програмування описується за допомогою набору правил, які визначають структуру правильної програми. Як зазначалось раніше, найбільш зручним формалізмом для опису синтаксичних конструкцій мови програмування є контекстно-вільні граматики (наприклад, широко поширена нормальна форма Бекуса-Наура).

Граматики однаково допомагають вирішувати задачі як програмістів, що використовують мову, так і розробників компіляторів для даної мови:

- Граматика надає точну і досить легку для розуміння синтаксичну специфікацію мови програмування.
- Для деяких класів граматик ми можемо автоматично сконструювати ефективний аналізатор, який визначає, чи є вхідна програма синтаксично правильною.
- Акуратно створена граматика може додати мові програмування таку структуру, яка буде корисна і при трансляції вхідної програми в правильний об'єктний код, і при визначенні помилок.
- Компілятори, розроблені на базі граматик, можна досить легко розширяти новими конструкціями, як результат розвитку мови.

Ще раз підкреслимо, що за допомогою контекстно-вільних граматик визначається тільки контекстно-вільна складова мови програмування, тобто тільки те, яким чином записується та чи інша конструкція мови. Інша важлива частина визначення правильної програми – коректність використання типів у програмі – не може бути визначена за допомогою контекстно-вільних граматик. Тому якщо програма відповідає граматиці, це ще не означає, що вона цілком є правильною.

Форма Бекуса-Наура

Один з найбільш розповсюджених способів опису синтаксису мови програмування – це форми Бекуса-Наура (БНФ). Цей спосіб був розроблений Бекусом і Науром (Backus J.W., Naur P.) для опису Алгола-60, однак, надалі він

використаний для багатьох інших мов. Синтаксис виразів мови програмування задається деякою сукупністю БНФ або синтаксичних правил.

У кожній мові є своя система понять. Зокрема, будь-який конкретний оператор є представником загального поняття «оператор», будь-який ідентифікатор є представником загального поняття «ідентифікатор».

Деякі поняття містять у собі інші поняття. Наприклад, поняття «оператор присвоєння» по суті містить в собі ще два поняття: це «ідентифікатор» і «вираз».

Будемо позначати поняття мови кутовими дужками (наприклад, <ідентифікатор>) і називати поняття нетермінальними символами, будемо розглядати поняття, як щось неподільне.

Символи і лексеми мови будемо називати термінальними символами або терміналами (наприклад, символ присвоєння ':='). Вони також розглядаються як неподільні. Послідовність, яка складена з терміналів і нетерміналів, називається метавиразом.

Взагалі, усяку фразу вигляду

```
<поняття> має структуру <метавираз>
```

можна переписати так:

```
<поняття> ::= <метавираз>.
```

Синтаксичні правила, записані у вигляді

```
<поняття> ::= <метавираз>,
```

називаються формами Бекуса-Наура (БНФ). При записі правил у формі Бекуса-Наура використовуються такі типи об'єктів:

- основні символи (або термінальні символи, зокрема, ключові слова);
- металінгвистичні змінні (метазмінні або нетермінальні символи), значеннями яких є ланцюжки основних символів описуваної мови. Металінгвистичні змінні зображуються словами в кутових дужках (<...>);
- металінгвистичні зв'язки (метасимволи зв'язування) (::=, |). Вертикальна риска (|) використовується для визначення альтернатив, а метасимвол ::= заміняє фразу «має структуру».

Розширена форма Бэкуса-Наура

Дві сукупності БНФ називаються еквівалентними, якщо задають одну й ту ж формальну мову.

Для запису еквівалентних БНФ у більш короткому і наочному вигляді алфавіт метасимволів розширюється символами "(", ")", "[", "]", "{", "}". Метавирази з такими символами називаються розширеними, а БНФ – розширеними БНФ, або скорочено РБНФ.

При визначенні синтаксису мов Pascal і Modula-2 Вірт використовував саме розширену форму Бекуса-Наура (РБНФ):

- нетермінали записуються як окремі слова;
- термінали записуються в лапках, наприклад, ‘for’;
- вертикальна риска (|), як і раніше, використовується для визначення альтернатив;
- круглі дужки використовуються для групування;
- квадратні дужки використовуються для визначення можливого входження символу або групи символів;
- фігурні дужки використовуються для визначення можливого повторення символу або групи символів;
- Коментарі містяться між символами (* … *);
- ε еквівалентно [ ].

Синтаксичний аналіз

Синтаксичний аналіз – це процес, що визначає, чи належить деяка послідовність лексем граматиці мови програмування. В принципі, для будь-якої граматики можна побудувати синтаксичний аналізатор, але граматики, які використовуються на практиці, мають спеціальну форму. Наприклад, відомо, що для будь-якої контекстно-вільної граматики може бути побудований аналізатор, складність якого не перевищує O(n3) для вхідного рядка довжиною n, але в більшості випадків для заданої мови програмування ми можемо побудувати таку граматику, що дозволить сконструювати і більш швидкий аналізатор.

Аналізатори реальних мов зазвичай мають лінійну складність; це досягається за рахунок перегляду вхідної програми зліва направо із загляданням уперед на один термінальний символ (лексичний клас).

Вхід синтаксичного аналізатора – це послідовність лексем і таблиці представлень, які є виходом лексичного аналізатора.

На виході синтаксичного аналізатора отримуємо дерево граматичного розбору і таблиці ідентифікаторів та типів, які є входом для наступного перегляду компілятора (наприклад, це може бути перегляд, який здійснює контроль типів – семантичний аналіз).

Існує три основних типи синтаксичних аналізаторів (англ.: parser), що реалізують три стратегії розбору:

1) стратегія аналізу зверху вниз (низхідний аналіз);

2) стратегія аналізу знизу вверх (висхідний аналіз);

3) змішана стратегія.

Низхідні аналізатори формують висновок, починаючи від аксіоми граматики і закінчуючи ланцюгом термінальних символів.

Популярність низхідних аналізаторів пов'язана з тим, що їх достатньо легко можна побудувати вручну, наприклад, методом рекурсивного спуску. Крім того, LL-граматики легко узагальнюються: граматики, що не є LL-граматиками можуть бути проаналізовані методом рекурсивного спуску з поверненнями.

Метод рекурсивного спуску

Одним з найбільш простих і найбільш популярних методів низхідного синтаксичного аналізу є метод рекурсивного спуску (recursive descent method).

Метод заснований на тому, що в склад синтаксичного аналізатора входить множина рекурсивних процедур граматичного розбору, по одній для кожного правила граматики.

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).

КОНТРОЛЬНІ ПИТАННЯ

1) Що таке синтаксичний аналіз?
2) Які є методи синтаксичного аналізу?
3) Які основні функції синтаксичного аналізу?
4) Що таке метод рекурсивного спуску?
5) Що таке форма Бекуса-Наура?

ЛІТЕРАТУРА

1) Основи проектування трансляторів: Конспект лекцій : [Електронний ресурс] : навч. посіб. для студ. спеціальності 123 – «Комп'ютерна інженерія» / О. І. Марченко ; КПІ ім. Ігоря Сікорського. – Київ: КПІ ім. Ігоря Сікорського, 2021.
2) Сопронюк Т.М. Системне програмування. Частина ІІ. Елементи теорії компіляції: Навчальний посібник у двох частинах. – Чернівці: ЧНУ, 2008. – 84 с.
3) Alfred V. Aho, Monica S. Lam, Ravi Seth, Jeffrey D. Ullma. Compilers, principles, techniques, and tools, Second Edition, New York, 2007. – 1038 с.

ЗАВДАННЯ

1. Визначити задану граматику у термінах Бекуса-Наура.
2. Розробити алгоритм роботи синтаксичного аналізатора за допомогою методу рекурсивного спуску.
3. Написати і налагодити програму, яка моделює роботу розробленого синтаксичного аналізатора.
4. Скласти звіт про виконану роботу. У звіті мають бути:
   - Опис граматики у термінах розширеної нотації Бекуса-Наура;
   - Схема дерева розбору для визначеної граматики;
   - Вхідні дані для програми і результати роботи програми;
   - Коди програми, яка моделює роботу синтаксичного аналізатора.
5. Дати відповідь на контрольні запитання викладача.

Варіанти індивідуальних завдань студент отримує у викладача!

ПРИКЛАД ВИКОНАННЯ

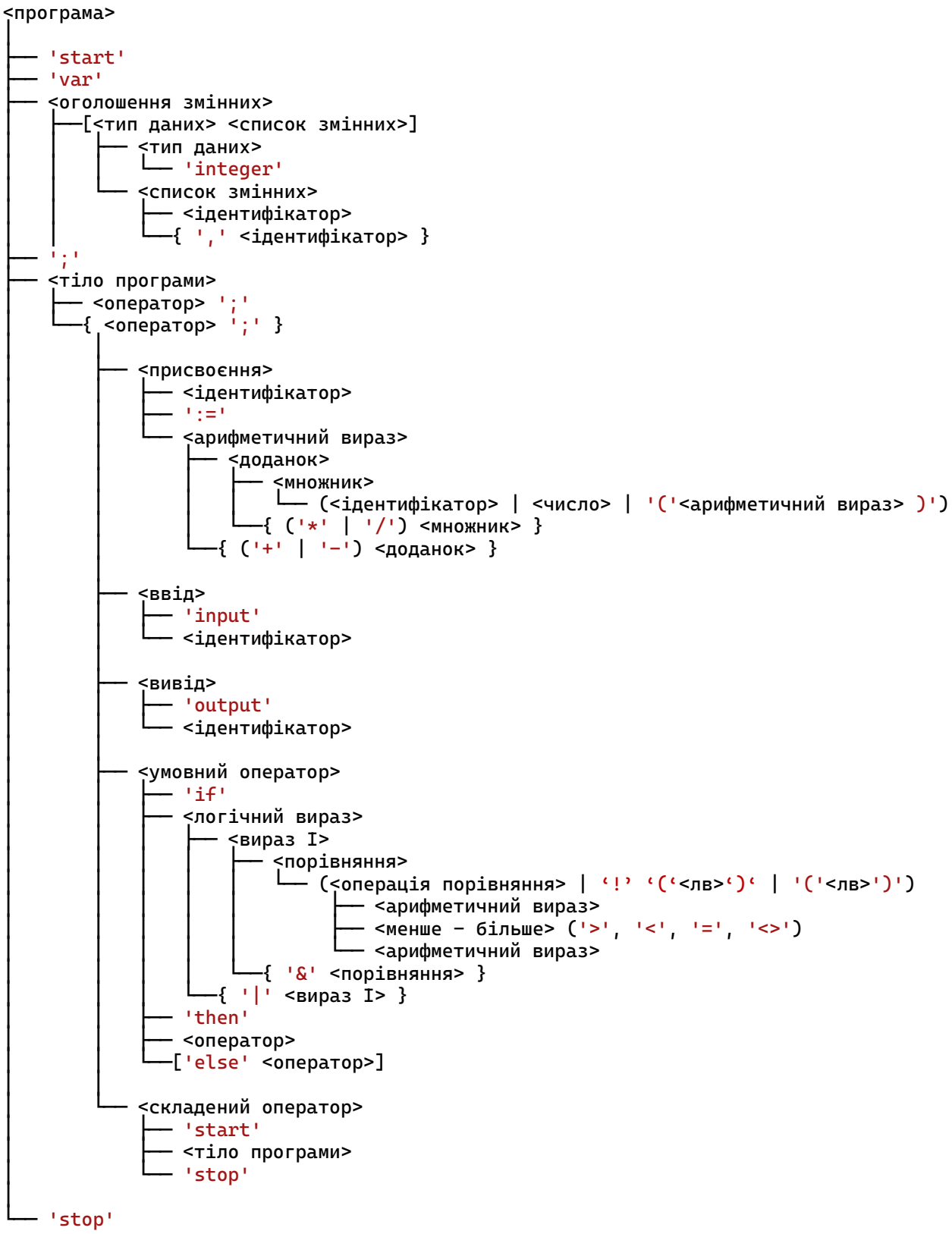
Завдання: реалізувати синтаксичний аналізатор для вхідної мови програмування.

Програма на вхідній мові програмування має починатись з ключового слова **start**, далі має йти розділ опису змінних **var**. Між розділом **var** і ключовим словом **stop** розміщуються оператори програми. Операторів є 4: оператор вводу даних **input**, оператор виводу даних **output**, оператор присвоєння **:=** і умовний оператор **if – then [- else]**. Кожен оператор має завершуватись символом крапка з комою **;**. Оператор присвоєння дозволяє присвоїти деякій змінній значення арифметичного виразу. Допустимі арифметичні операції - **+ , -, *, /**. Операндами можуть бути змінні, цілі додатні константи і інші вирази, взяті в дужки. В умовному операторі використовуються логічні вирази, допустимі такі операції порівняння **> , < , = , <>** і такі логічні операції **! , & , |**. Ідентифікатори (імена змінних) можуть бути довжиною до 4-х символів і складаються лише з маленьких латинських літер або цифр. Перший символ ідентифікатора завжди літера. Тип даних лише один – **integer**, при оголошення декількох змінних вони записуються через кому, вкінці опису змінних ставиться символ крапка з комою **;**. Коментарі починаються з двох косих ліній **// …** .

Спочатку **опишемо вхідну мову програмування у термінах розширеної форми Бекуса-Наура**:

<програма> = ‘**start**‘ ‘**var**‘ <оголошення змінних> ‘**;**’ <тіло програми> ‘**stop**‘
<оголошення змінних> = [<тип даних> <список змінних>]
<тип даних> = ‘**integer**‘
<список змінних> = <ідентифікатор> { ‘**,**‘ <ідентифікатор> }
<ідентифікатор> = <буква> { <буква або цифра>}
<буква або цифра> = <буква> |<цифра>
<буква> = ‘**a**‘ | ‘**b**‘ | ‘**c**‘ | ‘**d**‘ | … | ‘**z**‘
<цифра> = ‘**1**‘ | ‘**2**‘ | ‘**3**‘ | ‘**4**‘ | ‘**5**‘ | ‘**6**‘ | ‘**7**‘| ‘**8**‘ | ‘**9**‘ | ‘**0**‘
<тіло програми> = <оператор> ‘**;**‘ { <оператор> ‘**;**‘ }
<оператор> = <присвоєння> | <ввід> | <вивід> | <умовний оператор> | <складений оператор>
<присвоєння> = <ідентифікатор> ‘**:=**‘ <арифметичний вираз>
<арифметичний вираз> = <доданок> { ‘**+**‘ <доданок> |‘**-**‘ <доданок> }
<доданок> = <множник> { ‘*****‘ <множник> | ‘**/**‘ <множник> }
<множник> = <ідентифікатор> | <число> | ‘**(**‘ <арифметичний вираз> ‘**)**‘
<число> = <цифра> { <цифра> }
<ввід> = ‘**input**‘ <ідентифікатор>
<вивід> = ‘**output**‘ <ідентифікатор>
<умовний оператор> = ‘**if**‘ <логічний вираз> ‘**then**‘ <оператор> [‘**else**‘ <оператор>]
<логічний вираз> = <вираз І> { ‘**|**‘ <вираз І> }
<вираз І> = <порівняння> { ‘**&**‘ <порівняння> }
<порівняння> = <операція порівняння> | ‘**!**‘ ‘**(**‘ <логічний вираз> ‘**)**‘ | ‘**(**‘ <логічний вираз> ‘**)**‘
<операція порівняння> = <арифметичний вираз> <менше-більше> <арифметичний вираз>
<менше-більше> = ‘**>**‘ | ‘**<**‘ | ‘**=**‘ | ‘**<>**‘
<складений оператор> = ‘**start**‘ <тіло програми> ‘**stop**‘

Схема дерева розбору виглядає наступним чином:



Далі **визначимо назви процедур**, що відповідають нетерміналам граматики таким чином:

<програма> = ‘**start**‘ ‘**var**‘ <оголошення змінних> ‘**;**’ <тіло програми> ‘**stop**‘
```
void program();
```

<оголошення змінних> = [<тип даних> <список змінних>]
```
void variable_declaration();
```

<тип даних> = ‘**integer**‘
<список змінних> = <ідентифікатор> { ‘**,**‘ <ідентифікатор> }
```
void variable_list();
```

<тіло програми> = <оператор> ‘**;**‘ { <оператор> ‘**;**‘ }
```
void program_body();
```

<оператор> = <присвоєння> | <ввід> | <вивід> | <умовний оператор> | <складений оператор>
```
void statement();
```

<присвоєння> = <ідентифікатор> ‘**:=**‘ <арифметичний вираз>
```
void assignment();
```

<арифметичний вираз> = <доданок> { ‘**+**‘ <доданок> |‘**-**‘ <доданок> }
```
void arithmetic_expression();
```

<доданок> = <множник> { ‘*****‘ <множник> | ‘**/**‘ <множник> }
```
void term();
```

<множник> = <ідентифікатор> | <число> | ‘**(**‘ <арифметичний вираз> ‘**)**‘
```
void factor();
```

<ввід> = ‘**input**‘ <ідентифікатор>
```
void input();
```

<вивід> = ‘**output**‘ <ідентифікатор>
```
void output();
```

<умовний оператор> = ‘**if**‘ <логічний вираз> ‘**then**‘ <оператор> [‘**else**‘ <оператор>]
```
void conditional();
```

<логічний вираз> = <вираз I> { ‘**|**‘ <вираз I> }
```
void logical_expression();
```

<вираз I> = <порівняння> { ‘**&**‘ <порівняння> }
```
void and_expression();
```

<порівняння> = <операція порівняння> | ‘**!**‘ ‘**(**‘ <логічний вираз> ‘**)**‘ | ‘**(**‘ <логічний вираз> ‘**)**‘

<операція порівняння> = <арифметичний вираз> <менше-більше> <арифметичний вираз>

<менше-більше> = ‘**>**‘ | ‘**<**‘ | ‘**=**‘ | ‘**<>**‘
```
void comparison();
```

<складений оператор> = ‘**start**‘ <тіло програми> ‘**stop**‘
```
void compound_statement();
```

Структура синтаксичного аналізатора буде такою:

```
// Вхідна таблиця лексем
extern Token* TokenTable;
int pos = 0;

void parser()
{
    program();
    printf("\nThe program is syntax correct.\n");
}
```

Розглянемо рекурсивні процедури, що реалізують синтаксичний аналізатор:

```
void match(TypeOfTokens expectedType)
{
    if (TokenTable[pos].type == expectedType)
        pos++;
    else
    {
        printf("\nSyntax error in line %d : another type of lexeme was
expected.\n", TokenTable[pos].line);
        exit(1);
    }
}

// <програма> ::= 'start' 'var' <оголошення змінних> ';' <тіло програми> 'stop'
void program()
{
    match(StartProgram);
    match(Variable);
    variable_declaration();
    match(Semicolon);
    program_body();
    match(EndProgram);
}

// <оголошення змінних> ::= [<тип даних> <список змінних>]
void variable_declaration()
{
    if (TokenTable[pos].type == Type)
    {
        pos++;
        variable_list();
    }
}

// <список змінних> ::= <ідентифікатор> { ',' <ідентифікатор> }
void variable_list()
{
    match(Identifier);
    while (TokenTable[pos].type == Comma)
    {
        pos++;
        match(Identifier);
    }
}

// <тіло програми> ::= <оператор> ';' { <оператор> ';' }
void program_body()
{
    do
    {
        statement();
        match(Semicolon);
    }
    while (TokenTable[pos].type != EndProgram);
}
```

```
// <оператор> ::= <присвоєння> | <ввід> | <вивід> | <умовний оператор> |
<складений оператор>
void statement()
{
    switch (TokenTable[pos].type)
    {
    case Input: input(); break;
    case Output: output(); break;
    case If: conditional(); break;
    case StartProgram: compound_statement(); break;
    default: assignment();
    }
}

// <присвоєння> ::= <ідентифікатор> ':=' <арифметичний вираз>
void assignment()
{
    match(Identifier);
    match(Assign);
    arithmetic_expression();
}

// <арифметичний вираз> ::= <доданок> { ('+' | '-') <доданок> }
void arithmetic_expression()
{
    term();
    while (TokenTable[pos].type == Add || TokenTable[pos].type == Sub)
    {
        pos++;
        term();
    }
}

// <доданок> ::= <множник> { ('*' | '/') <множник> }
void term()
{
    factor();
    while (TokenTable[pos].type == Mul || TokenTable[pos].type == Div)
    {
        pos++;
        factor();
    }
}

// <множник> ::= <ідентифікатор> | <число> | '(' <арифметичний вираз> ')'
void factor()
{
    if (TokenTable[pos].type == Identifier)
    {
        match(Identifier);
    }
    else
        if (TokenTable[pos].type == Number)
        {
            match(Number);
        }
        else
            if (TokenTable[pos].type == LBraket)
            {
            match(LBraket);
            arithmetic_expression();
            match(RBraket);
            }
            else
            {
                printf("\nSyntax error in line %d : A multiplier was
expected.\n", TokenTable[pos].line);
```

```
                exit(1);
            }
}

// <ввід> ::= 'input' <ідентифікатор>
void input()
{
    match(Input);
    match(Identifier);
}

// <вивід> ::= 'output' <ідентифікатор>
void output()
{
    match(Output);
    match(Identifier);
}

// <умовний оператор> ::= 'if' <логічний вираз> 'then' <оператор> [ 'else'
<оператор> ]
void conditional()
{
    match(If);
    logical_expression();
    match(Then);
    statement();
    if (TokenTable[pos].type == Else)
    {
        pos++;
        statement();
    }
}

// <логічний вираз> ::= <вираз I> { '|' <вираз I> }
void logical_expression()
{
    and_expression();
    while (TokenTable[pos].type == Or)
    {
        pos++;
        and_expression();
    }
}

// <вираз I> ::= <порівняння> { '&' <порівняння> }
void and_expression()
{
    comparison();
    while (TokenTable[pos].type == And)
    {
        pos++;
        comparison();
    }
}

// <порівняння> = <операція порівняння> | ‘!‘ ‘(‘ <логічний вираз> ‘)‘ | ‘(‘
<логічний вираз> ‘)‘
// <операція порівняння> = <арифметичний вираз> <менше-більше> <арифметичний
вираз>
// <менше-більше> = ‘>‘ | ‘<‘ | ‘=‘ | ‘<>‘
void comparison()
{
    if (TokenTable[pos].type == Not)
    {
        // Варіант: ! (<логічний вираз>)
        pos++;
        match(LBraket);
        logical_expression();
```

```
        match(RBraket);
    }
    else
        if (TokenTable[pos].type == LBraket)
        {
            // Варіант: ( <логічний вираз> )
            pos++;
            logical_expression();
            match(RBraket);
        }
        else
            {
            // Варіант: <арифметичний вираз> <менше-більше> <арифметичний вираз>
            arithmetic_expression();
            if (TokenTable[pos].type == Greate || TokenTable[pos].type == Less
||
                TokenTable[pos].type == Equality || TokenTable[pos].type ==
NotEquality)
            {
                pos++;
                arithmetic_expression();
            }
            else
                {
                    printf("\nSyntax error: A comparison operation is
expected.\n");
                    exit(1);
                }
            }
}

// <складений оператор> ::= 'start' <тіло програми> 'stop'
void compound_statement()
{
    match(StartProgram);
    program_body();
    match(EndProgram);
}
```