Перетворення виразів ЕТ з інфіксної форми в постфіксну

Перетворення виразів – це є реалізація програми, яка інфіксне зображення однієї формули електронної таблиці перетворює на постфіксне. Отже, та сама формула набуває постфіксного вигляду, що дозволяє ефективно реалізувати обчислення формул. Наприклад, якщо маємо формулу (з попереднього розділу):

```
= B5+B6-SUM(C4:C10)/2
```

то поділ на лексеми і інфіксна форма виглядали би так:

=	B5	+	B6	–	SUM	(	C4	:	C10	)	/	2

Після перетворення в постфіксну форму запис буде таким:

B5	B6	+	C4	C10	:	SUM	2	/	–

В постфіксній формі операція має бути записана за операндами, до яких вона стосується. Знак "=", який є на початку формули, можна викреслити з постфіксної форми, бо він не є операцією, а лише ознакою формули в комірці таблиці. Дужки в постфіксній формі відсутні, бо всі операції виконують зліва направо в порядку записування.

Постфіксну форму виразу можна отримати за схемою, наведеною раніше – за допомогою процедур генерування постфіксної форми на основі граматики виразів. Реалізацію граматики виразів і розпізнавання виразу реалізуємо методом нисхідного граматичного розбору за алгоритмом рекурсивного спуску. З теорії формальних мов і граматик відомо, що для реалізації методу рекурсивного спуску потрібно переписати граматичні правила у вигляді, де немає ліворекурсивних правил. При цьому використовують символи дужок "{", "}", "(", ")", "[", "]" як метасимволи, тобто для зображення самої формули. Якщо ж будь-яка дужка є елементом самої формули, то дужку записують в апострофах, наприклад '('.

Значення дужок як метасимволів є таким. Фігурні дужки {*} означають, що заключена в них конструкція може повторюватись нуль або більше разів. Квадратні дужки [*] означають, що заключена в них конструкція може бути або відсутня, або присутня лише один раз, тобто альтернатива присутності. Круглі дужки (*) вживаються тоді, коли потрібно зробити вибір однієї з альтернатив, записаних в круглих дужках.

Отже, визначені раніше граматичні правила для формул перепишемо в такому вигляді:

Формули:

```
<формула> ::=   = <вираз>
<вираз> ::=  <доданок>  { ( + | - ) <доданок> }
<доданок> ::=  <множник>  {  ( * | / )  <множник>  }
<множник> ::=  <число>  |  <комірка>  |  <функція>  |  '(' <вираз> ')'
<комірка>  ::=  <букви> <цифри>
<букви> ::= буква | буква буква        } не використовуємо, бо вже розпізнано
<цифри> ::= цифра | цифра цифра
<функція> ::=  ( SUM | MAX | MIN ) '(' <ряд> ')'
             | ( ABS | INT ) '(' < вираз > ')'
<ряд> ::=  <комірка> : <комірка>
```

За методом рекурсивного спуску до кожного нетермінального символа будують окрему функцію, яка повинна розпізнати праву частину граматичного правила. При цьому функція може викликати в потрібні моменти інші функції для розпізнавання частини конструкції. Наприклад, функція для <вираз> буде викликати функцію для <доданок>, функція для <доданок> буде викликати функцію для <множник>, і т.д. Врешті за такою схемою можливі непрямі рекурсії, наприклад, <вираз> → <доданок> → <множник> → <вираз> → … .

Вхідними даними для програми перетворення з інфіксної форми в постфіксну є масив лексем, побудований сканером. Отже, всі лексеми вже мають однаковий числовий формат, що значно спрощує програмування такого перетворення. Відповідно, вихідними даними цієї програми є перебудований масив лексем в тому самому числовому форматі.

Головна ідея генерування постфіксної форми полягає в правильному визначенні послідовності записування елементів виразу в постфіксну форму. Постфіксна форма – це є

спільний масив (потік) для всіх функцій розпізнавання нетермінальних символів. Функції почергово записують в такий спільний масив елементи виразу, які вони розпізнали. Якщо маємо, наприклад, два доданки виразу за формулою

```
<вираз> ::= <доданок> + <доданок>
```

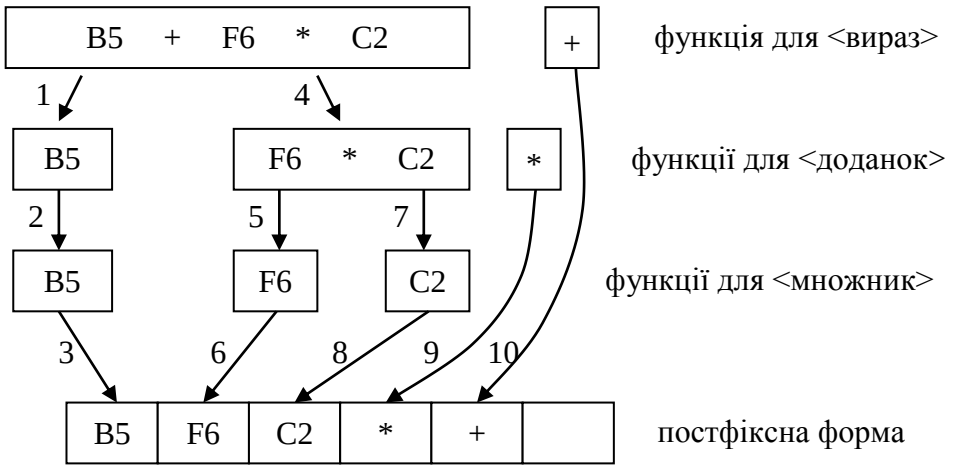
то функція для <вираз> повинна спочатку викликати функцію <доданок> для обчислення першого доданка, потім ще раз цю саму функцію для обчислення другого доданка, після чого записати до постфіксної форми знак операції '+'. Але функції для <доданок> першими запишуть до постфіксної форми свої доданки. Отже, якщо мали, наприклад, вираз B5+F6, то в результаті вийде B5,F6,+. Якщо ж вираз буде мати вигляд

B5 + F6 * C2

то будуть використані формули у вигляді

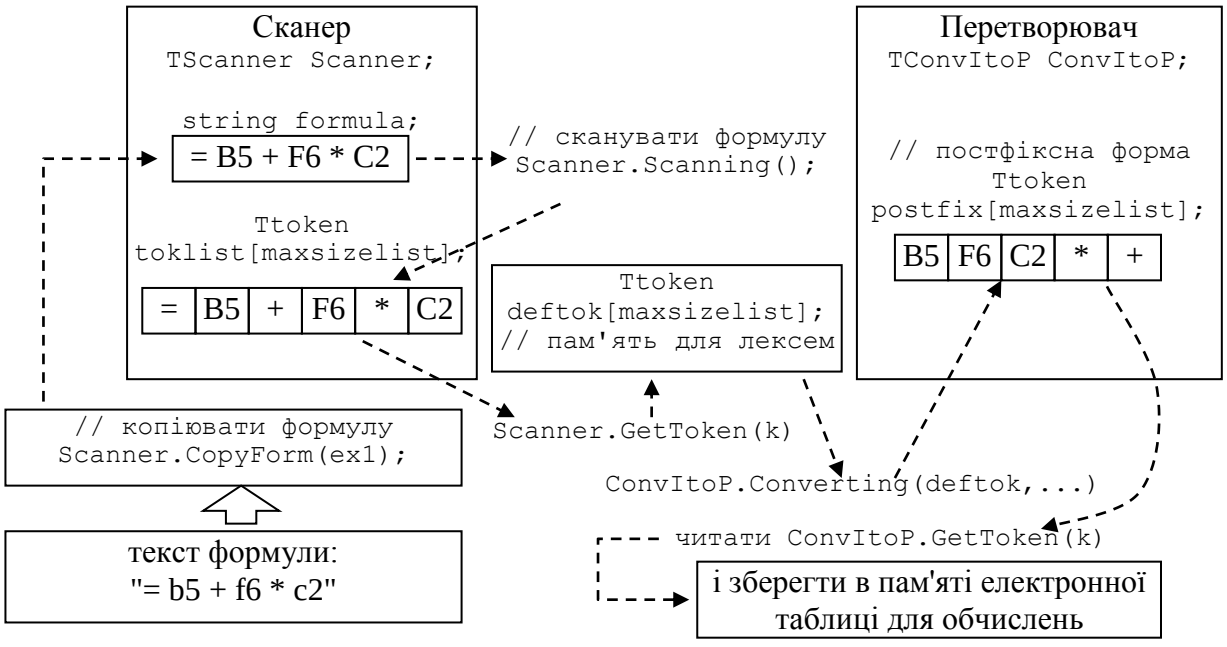
```
<вираз>   ::=   <доданок>   +  <доданок>
<доданок>  ::=   <множник>   |   <множник> *  <множник>
<множник>  ::=    <комірка>
```

В результаті побудова постфіксної форми буде виконана за схемою, як на рисунку.



В процесі перетворення в постфіксну форму можуть бути виявлені різні помилки, пов'язані з неправильним записом самого виразу, наприклад: між двома іменами комірок немає знака операції; відсутня закриваюча кругла дужка у виразі; у тексті формули знак операції записаний двічі підряд '++'; і інші. У випадку знаходження помилки процес перетворення припиняється і повідомляється зміст помилки та її імовірна позиція.

Загальну схему взаємодії і використання сканера і перетворювача формул до постфіксної форми можна подати такою діаграмою:



Схему програмування перетворювача можна подати такими фрагментами програми:

```
class TConvItoP  // клас перетворювача
{
  protected:
     Ttoken * toklist; // вказівник на масив лексем в інфіксній формі
     int size;  // кількість лексем в інфіксному масиві
     int ff;       // номер поточної лексеми в інфіксному масиві
     Ttoken * token; // вказівник на поточну лексему в інфіксному масиві
     Ttoken postfix[maxsizelist];  // масив лексем в постфіксній формі
     int N;       // номер останньої лексеми в постфіксному масиві (0<=N<100)
     Ttoken * errcontext; // лексема, імовірно до якої стосується помилка
     bool error; // ознака наявності помилки при перетворенні
     int codeerror;  // номер помилки за класифікацією перетворювача

     // допоміжні функції для побудови постфіксної форми
     void GetNextTok() // перейти до наступної лексеми інфіксної форми
     {
       if(ff < size-1) { ff++; token= &toklist[ff]; }
       else { token= &NulTok; } // список лексем вичерпаний
     } // void GetNextTok()

     void SaveTok(Ttoken * t) // записати лексему t в постфіксний масив
     {
       N++; postfix[N]= *t; // копія лексеми t
     } // void SaveTok(Ttoken * t)
. . . . . .

     void Formula(); // функція для <формула>
     void Vyraz();   // функція для <вираз>
     void Dodanok(); // функція для <доданок>
     void Mnoznyk(); // функція для <множник>
     void ETFunction(); // функція для <функція>
     void ETRjad();  // функція для <ряд>

  public:
. . . . . .

// визначення методів класу: стартова функція для перетворення
int TConvItoP::Converting(Ttoken * t, int c, Ttoken * badtoken, Ttoken *
current)
{ // перетворити в постфіксну форму
  // t - масив лексем в інфіксній формі; c - кількість лексем
  // badtoken - може означати лексему, до якої стосується помилка
  // current - лексема на момент виявлення помилки
  toklist=t;  size=c; // запам'ятати параметри
  ff = -1; // найперша лексема в інфіксному масиві
  N= -1;  // порожній масив лексем постфіксної форми
  error=false; // ознака наявності помилки
  errcontext = &NulTok; // імовірна позиція помилки

  // побудова дерева синтаксичного розбору з реагуванням на помилки
  try
  {
    // всі функції викликаються при першій лексемі, яка до них належить
    GetNextTok(); // вибрати найпершу лексему
    Formula();  // запуск дерева розбору
  }
  catch (int ne) // номер (код) виявленої помилки
  {
    error=true; codeerror=ne;
  }

  if(!error) return 0; // помилок не було
  else
```

```
   { // у випадку помилки передається лексема з помилкою
     //  і повертається номер помилки
      *badtoken = *errcontext; // лексема, до якої може стосуватися помилка
      *current = *token; // лексема на момент виявлення помилки
     return codeerror; // номер знайденої помилки
   }
} // int TConvItoP::Converting(Ttoken * t, int c, Ttoken & badtoken, Ttoken *
current)
. . . . . .

void TConvItoP::Formula() // функція для <формула>
{
   if(*token != s_equal)
   { errcontext = token;
     throw 2001; // немає знака = на початку формули
   }
   GetNextTok(); // знак = не зберігаємо
   Vyraz(); // перехід до аналізу виразу
   // чи всі лексеми були проаналізовані?
   if(*token != end_list)
   { errcontext=token; // позиція помилки
     throw 2002; // відсутній або неправильний знак операції
   }
} // void TConvItoP::Formula()

void TConvItoP::Vyraz() // функція для <вираз>
{
   Ttoken * opr; // вказівник на лексему зі знаком
   Dodanok();  // найперший доданок виразу
   while(*token == s_plus || *token == s_minus)
   {
     opr = token; // запам'ятати лексему зі знаком
     GetNextTok(); // вибрати наступну лексему після знака
     Dodanok();  // наступний доданок виразу
     SaveTok(opr); // записати в постфіксну форму знак
   }
} // void TConvItoP::Vyraz()
. . . . . .
void TConvItoP::Mnoznyk() // функція для <множник>
{
   if(*token == number_et || *token == cell_et) // це число або комірка
   { SaveTok(token); // записати в постфіксну форму число або комірку
     GetNextTok(); // вибрати наступну лексему після числа або комірки
   }
   else if
   (*token >= fsum && *token <= fint) // це функція
   { ETFunction(); // перейти до аналізу функції
   }
   else if
   (*token == openbracket) // це вираз в дужках
   {
     GetNextTok(); // наступна лексема після відкриваючої дужки
     Vyraz(); // аналіз виразу в дужках
     if(*token!=closebracket)
     { errcontext = token;
       throw 2003; // немає закриваючої дужки для виразу
     }
     GetNextTok(); // наступна лексема після закриваючої дужки
   }
   else
   { errcontext = token;
     throw 2004; // неправильний запис елемента виразу
   }
} // void TConvItoP::Mnoznyk()
```