УДК 681.3

РОЗРОБКА СИНТАКСИЧНОГО АНАЛІЗАТОРА МОВИ ПРОГРАМУВАННЯ PL/I ДЛЯ РЕІНЖЕНІРИНГУ БЛОК-СХЕМ АЛГОРИТМІВ

М.Т. Фісун, О.В. Гнездьонова

Миколаївський державний гуманітарний університет ім. Петра Могили
54003, Миколаїв, вул. 68 Десантників, 10
тел.: 8 (0512) 24 30 18; E-mail: ntfis@kma.mk.ua, ognezdyonova@bonustec.com

В статье рассмотрены вопросы построения лексического и синтаксического анализаторов для языка программирования PL/1 с целью получения на выходе блок-схемы алгоримов програм на этом языке. Проанализированы особенности грамматики PL/1 для получения информации о структуре программы. Описан подход к реализации названного транслятора.

The questions of the lexical and syntacsic analyzers construction for programming language PL/1 are considered in the article with the purpose of getting flowchart from source code on an output. The PL/1 grammar features for getting information about program structure are analyzed. The approach for compiler realization is described.

Розглядаються питання побудови лексичного та синтаксичного аналізаторів для мови програмування PL/1 з ціллю одержання на виході блок-схем алгоритмів програм на цій мові. Проаналізовані особливості граматики PL/1 для одержання інформації про структуру програми. Описано підхід до реалізації названого транслятора.

Загальна постановка проблеми та її зв'язок з науково-практичними задачами. Однією із галузей інформаційних технологій, що займають певну частку ринку, є реінжиніринг комп'ютерних систем, у тому числі і їх програмного забезпечення. Це пов'язано з тим, що на підприємствах, організаціях, установах функціонують комп'ютерні системи різного призначення, і з часом виникає необхідність переробки системи в цілому або її складових частин. Останнім часом спостерігається тенденція до збільшення тривалості життєвого циклу вдалих програмних проектів. У наслідок цього збільшується обсяг успадкованого коду, який підтримується організацією – розробником [1]. Саме це пояснює виняткову важливість задач, пов'язаних з полегшенням супроводу та розвитку програмного коду. Водночас, цим задачам приділяється недостатньо уваги зі сторони розробників інструментальних засобів. Особливо це стосується комп'ютерних систем, розроблених на платформі ЕОМ класу Mainframe (IBM 370/390 та інші моделі). Не дивлячись на те, що нині на порядок денний ставиться більш складна задача трансформації старого коду на рівні абстракції, архітектури, програмного забезпечення [2], – задача трансформації первинного коду „старої системи" в код сучасної мови програмування залишається *актуальною*, тому що вона не зводиться для простої перекомпіляції коду і підходи до її розв'язання залежать від конкретної ситуації.

Огляд публікацій та аналіз невирішених проблем. Більша частина сучасних мов програмування придатні для побудови синтаксичних аналізаторів з використанням стандартних LALR–граматик [3]. Однак написання синтаксичного аналізатора для „старої" мови програмування найчастіше ставить перед програмістом більш складні завдання [3]. Основні труднощі пов'язані з тим, що дуже часто програмне забезпечення погано документоване, неодноразово змінювалося, дописувалося різними командами програмістів, що створює значні труднощі при його супроводі.

Існує багато програм за допомогою яких можна представити програмний код у вигляді блок-схеми алгоритмів**.** На наш погляд, найбільш вдалою є програма "Code Visual to Flowchart". Вона аналізує програмний код, що написаний такими мовами як VC, C, C++, Java, JavaScript, VB, BASIC, ASP, vbscript, Delphi, Pascal, PHP, C#, VB.NET, Perl, але – мова PL/1 у цьому переліку відсутня. Водночас як раз ця мова програмування широко використовувалася для створення прикладних програм на Mainframe. На рис. 1 показано перетворення коду, що написаний мовою C++, за допомогою Code Visual to Flowchart:

Наведена реалізація має два суттєвих недоліки:

1. Блок-схема не відповідає загальним правилам побудови [4, 5].
2. За такою блок-схемою стає неможливим відтворення програмного коду.

Як відомо, блок – схема алгоритму описує процес або послідовність дій. Вона повинна мати початок та кінець. Початок може бути лише один, а виходів – декілька [6].

Кожен блок має своє функціональне навантаження, відповідно до якого вони повинні заповнюватися текстом. Тобто, якщо ми використовуємо блок умовного переходу, то текст цього блоку повинен містити логічну умову, а не ключове слово «IF».

Мета досліджень. Створити інструментальні програмні засоби для реінжинірингу структури даних, що представлені мовою PL/1, та автоматизувати побудову блок-схем алгоритмів, що реалізовані цим програмним кодом.

Код функції мовою C++	Візуальне представлення

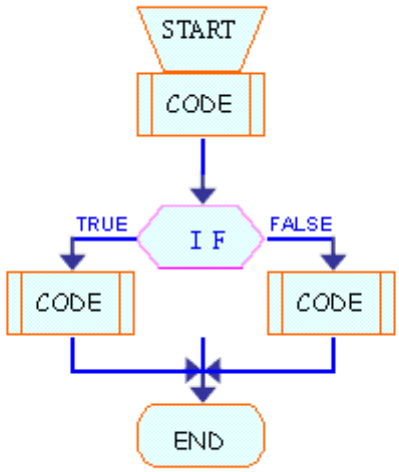
void CDecal::Activate()
{
BaseClass::Activate();
 	
if (!GetEntityName())
{
StaticDecal();
}
else
{
// if there IS a targetname, the decal sprays itself
SetThink (&CDecal::SUB_DoNothing);
SetUse(&CDecal::TriggerDecal);
}
}
	

Рис.1. Приклад створення блок-схеми алгоритму за допомогою програми Code Visual to Flowchart

Сформулюємо основні вимоги до програми реінжинірингу:

- дружній і інтуїтивно зрозумілий інтерфейс;
- можливість виділення ділянки коду для складання блок-схеми;
- розбір й групування вихідного коду на процедури, змінні, функції й т.д.;
- можливість вибору рівня деталізації блок-схеми;
- редагування блок-схеми;
- друк як вихідного тексту, так і блок-схеми;
- експорт блок-схеми в MS Word та MS Paint;.
- можливість згортання й розгортання логічно завершених ділянок блок-схеми;
- вибір елементів для відображення;
- програма повинна бути легко розширюваною й зручною для супроводу.

Результати досліджень. Далі викладається підхід до розв'язання сформульованої задачі шляхом розробки відповідного транслятора як одного із етапів у рамках проекту з автоматизації реінжинірингу блок-схем алгоритмів із програмного коду.

Найбільш складним етапом розробки й реалізації програми реінжинірингу є написання синтаксичного аналізатора мови програмування PL/1 [3]. Для цього розглянемо *особливості цієї мови*.

Найочевиднішої, але водночас не самої складної для реалізації особливістю мови PL/1 є незвичайне, за сучасними мірками, форматування вихідного тексту, коли значення символу залежить від його положення в рядку. Таким чином, перед синтаксичним аналізом вхідної програми необхідно позбутися від ігнорованого тексту, що може стояти з першої по сьому позицію та після сімдесят другої позиції у рядку. Далі необхідно перетворити коментарі в зручний для аналізу формат і т.д.

У граматиці мови PL/1 є й інші особливості, які значно ускладнюють написання синтаксичного й лексичного аналізаторів.

- Для написання програм можна користуватися одним із двох алфавітів: з 60 або з 48 символів.
- Підтримуються вкладені процедури й функції.
- Опис змінних можливо після їх використання. Або можливо взагалі не описувати змінні, тоді їм будуть привласнені типи за замовчуванням залежно від букви, на яку починається ім'я змінної.
- Можливість написання складних умовних операторів:

IF THEN THEN = ELSE; ELSE ELSE = THEN і т.д.

- Ключові слова можуть використатися як звичайні ідентифікатори.

Приклад: DECLARE (ARG1, ARG2, ..., ARGn)....

У даному прикладі залежно від того, що стоїть після символу „)", можна визначити, чи є DECLARE ім'ям підпрограми, чи оголошенням.

Довжина такого рядка може бути досить великою і вже неможливо відокремити фазу лексичного аналізу від синтаксичного, тому що для вищезазначеного прикладу лексичний аналізатор повинен використовуватись як підпрограма, яку викликає синтаксичний аналізатор для того, щоб отримати нову лексему. Процедура отримання лексем представлена на рис. 2.

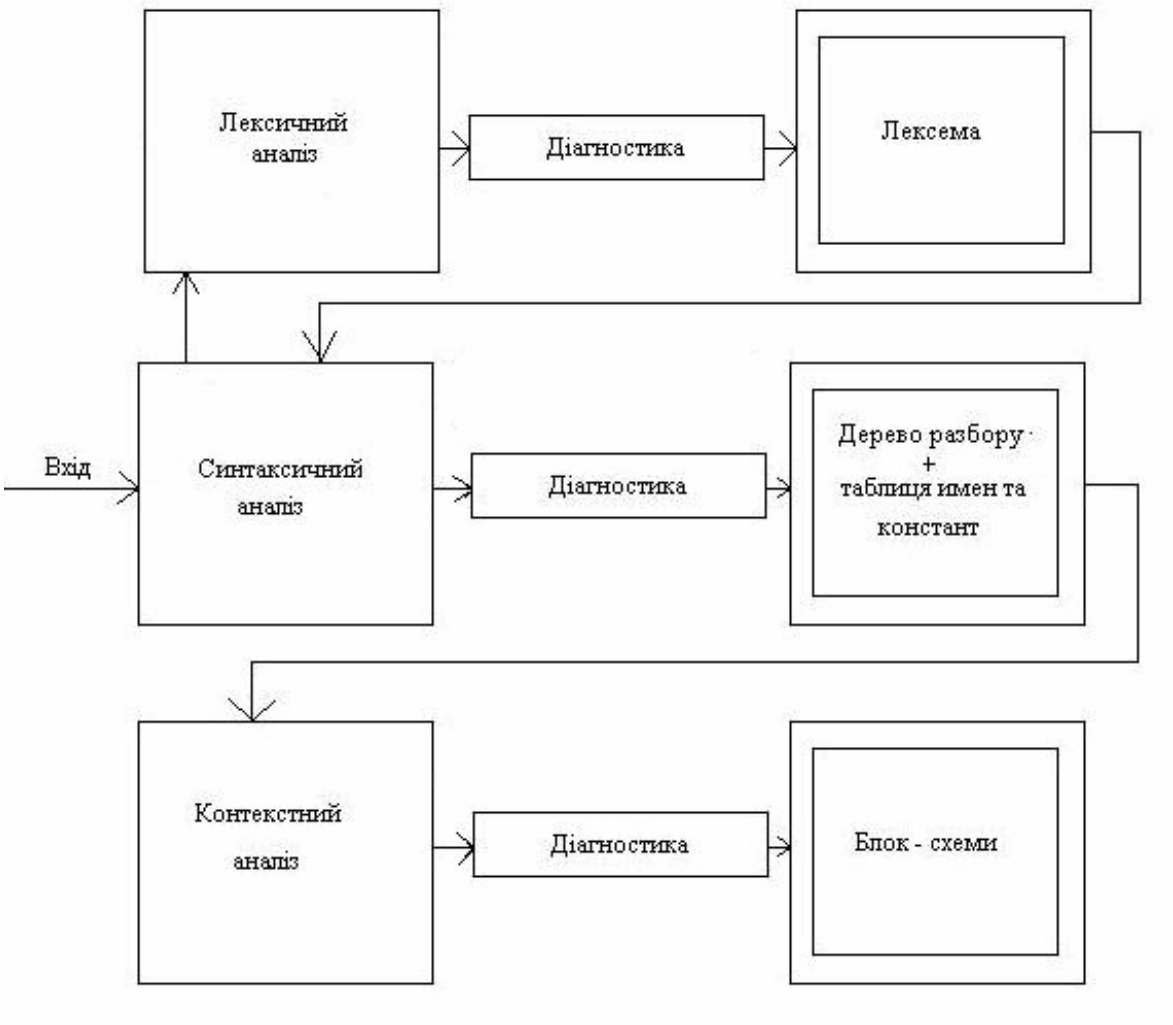


Рис. 2. Схема процедури отримання лексем

Лексичний аналіз – перша фаза процесу трансляції, призначена для групування символів вхідного ланцюжка в більші конструкції, які називаються лексемами. З кожною лексемою пов'язано два поняття:

- ***клас лексеми***, який визначає загальну назву для категорії елементів, які мають загальні властивості (наприклад, ідентифікатор, ціле число, рядок символів і т.д.);
- ***значення лексеми***, що визначає підрядок символів вхідного ланцюжка, що відповідає розпізнаному класу лексеми.

У принципі, завдання, які розв'язує лексичний аналізатор, можна покласти на синтаксичний аналізатор. Але такий підхід незручний [7].

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

Розбір зазначений для доказу того, що аналізована вхідна послідовність, записана у вхідному рядку, належить, або не належить множині послідовностей, які породжуються граматикою даної мови.

Щоб одержати відповідь "так", щодо всієї послідовності, треба її одержати для кожного правила, що забезпечує розбір окремого підрядка. Далі аналізуються оброблені вузли, і вже в них отримані відповіді складаються в загальну відповідь нового вузла. І так далі до самої вершини.

Результатом синтаксичного аналізу є синтаксичне дерево із посиланнями на таблицю імен. У процесі синтаксичного аналізу також знаходяться помилки, пов’язані із структурою програми.

На етапі контекстного аналізу виявляються залежності між частинами програми, які не можуть бути описані контекстно-незалежним синтаксисом. Це, в основному, зв’язки «опис-використання», аналіз типів об’єктів, аналіз області видимості, відповідність параметрів та ін. У процесі контекстного аналізу будується таблиця символів, яку можна розглядати як таблицю імен, поповнену інформацією про опис об’єктів.

Основним формалізом, який використовується при контекстному аналізі, є атрибутні граматики [7]. Результатом роботи фази контекстного аналізу є атрибутивне дерево програми. Інформація про об’єкти може

бути як розосереджена в самому дереві, так і зосереджена в окремих таблицях символів. У процесі контекстного аналізу також можуть бути знайдені помилки, пов'язані з невірним використанням об'єктів. Потім на базі отриманих дерев будується блок-схема.

Тепер перейдемо до опису синтаксичного аналізатора, функцією якого є одержання дерева розбору, з яким працюють всі наступні перегляди. При цьому він повинен задовольняти наступним вимогам:

- видавати коректне дерево синтаксичного розбору у випадку синтаксично коректного вхідного коду;
- видавати повідомлення про помилки, якщо це необхідно;
- кожен вузол дерева повинен мати прив'язку до вихідного тексту;
- відновлення після помилок, що забезпечує продовження синтаксичного аналізу вхідної програми.

При розробці аналізатора було прийняте рішення не використовувати існуючу систему **YACC** [3], а написати власний синтаксичний аналізатор. Таке рішення обумовлене тим, що при обробці досить складної граматики PL/I використання вбудованих механізмів системи **YACC** не дає бажаних результатів.

Тіло синтаксичного аналізатора – діаграма переходів відповідного детермінованого скінченого автомата (ДСА). ДСА – це п'ятірка M = (Q,T,D,q0,F), де

1) Q – скінчена множина станів;
2) T – скінчена множина допустимих вхідних символів;
3) D – функція переходів, яка відображає множини QxT в множину Q і визначає поведінки керуючого пристрою;
4) q0 < Q – початковий стан керуючого пристрою початкове;
5) F <= Q – множина кінцевих станів.

Робота скінченого автомата – деяка послідовність кроків або тактів. Такт визначається поточним станом керуючого пристрою та вхідним символом, який фіксується вхідною голівкою. Сам крок складається із

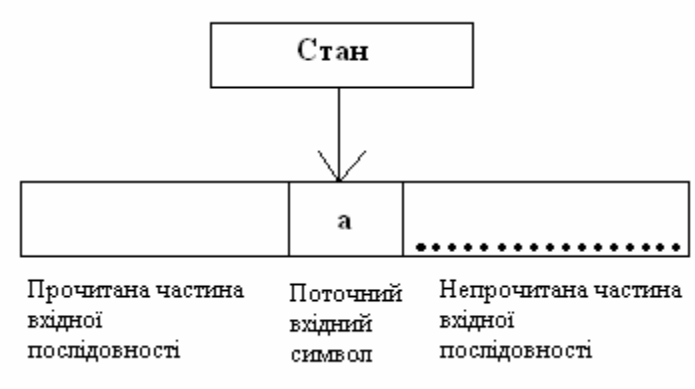


Рис. 3. Процес зсуву вхідної голівки

зміни стану та зсуву вхідної головки на одну комірку направо (рис. 3).

Лістинг з прикладом опису станів скінченого автомату з поясненнями показано на рис. 4.

Не дивлячись на те, що лексичний аналізатор обробляє вхідну послідовність, краще на його вхід подавати не окремі символи, а символи, які згруповані за категоріями. Тому перед лексичним аналізатором

```
TPLMSState = (    msMainProc, – початок програми
                  msProcessBody, – тіло програми
                  msComment ); – коментар

TStartPointState = (_msLineNumber, – начался номер строки
                  _msProgrammName, – початок ім'я програми
                  _msProgrammProc, – початок ключового слова PROCEDURE
                  _msProcParameters, – початок параметрів
                  _msProcOptions, – початок розділа опцій
                  _msOptionsParameters, – початок опису параметрів
                  _msEndOperator); – знайдено оператор кінця рядка

THandlerResult = ( hzNextChar, – читати наступний символ
                  hzRestartChar, – прочитати ще раз попередній символ
                  hzError, – знайдена помилка
                  hzComplete )
```

Рис.4. Лістинг станів для розпізнавання рядка початку програми

виконується додаткова обробка, яка співставляє з кожним символом його клас. Це дозволяє сканеру маніпулювати єдиним поняттям для цілої групи символів, іноді для достатньо великої. Пристрій, який впроваджує співставлення класу з кожним окремим символом, називається транс літератором [8]. Найбільш типовими класами символів є:

- ***літера*** – клас, з яким зіставляється множина літер і необов'язково лише з одного алфавіту;
- ***цифра*** – множина символів, які належать цифрам, частіше за все від 0 до 9;
- ***розділювач*** – пробіл, перехід на інший рядок, повернення каретки;
- ***ігноруємий*** – може зустрічатися у вхідному потоці, але ігнорується і тому видаляється з нього (наприклад, невидимий код звукового сигналу або інші аналогічні коди);
- ***недозволений*** – символи, які не належать алфавіту мови, але зустрічаються у вхідній послідовності;
- ***інші*** – символи, які не ввійшли до жодної з вищеперерахованих категорій. Символи, які не належать жодній з визначених категорій.

Лістинг функції перевірки приналежності вхідного символу до класу цифр показана на рис.5. На вхід функції подається символ з вхідної послідовності і перевіряється, чи входить він в зазначений діапазон від 0 до 9. Якщо входить, то функція повертає підтвердження того, що символ належить до класу цифр.

```
function TPLParser.isNumber ( C : Char ) : Boolean;
begin
   Result := False;
   if ( C in ['0'..'9'] ) then Result:=True;
end;
```

Рис. 5. Лістинг функції перевірки приналежності вхідного символу до класу цифр

На рис. 6 показано лістинг функції підтвердження: чи дійсно отримана частина вхідного рядка (Token) є числом.

```
function TPLParser.isNumber ( Token : String ) : Boolean;
var
  Counter : Integer;
begin
   Result := True;
   If ( Token = '' ) then Result := False;
   for Counter:=1 to Length(Token)-1 do
      begin
         if not isNumber(Token[Counter]) then
           begin
              Result := False;
              Exit;
           end;
      end;
end;
```

Рис. 6. Лістинг функції перевірки на число

Лістинг функції перевірки приналежності вхідного символу до класу літер показана на рис. 7. На вхід функції подається символ з вхідної послідовності і перевіряється чи, входить він в один з діапазонів 'A'..'Z','_','$','@','#','0'..'9'. Якщо входить, то функція повертає підтвердження того, що символ належить до класу літералів.

```
function TPLParser.isLiteral( C : Char ) : Boolean;
begin
   Result := False;
   if C in ['A'..'Z','_','$','@','#','0'..'9']  then Result := True;
end;
```

Рис. 7. Лістинг функції перевірки приналежності вхідного символу до класу літер

На рис. 8 показано лістинг функції підтвердження: чи дійсно отримана частина вхідного рядка (Token) є послідовністю літер.

```
function TPLParser.isLiteral( Token : String ) : Boolean;
var
  Counter : Integer;
begin
    Result := True;
    If ( Token = '' ) then Result := False;
    for Counter:=1 to Length(Token)-1 do
       if not isLiteral( Token[Counter] ) then
         begin
            Result := False;
            Exit;
         end;
end;
```

Рис. 8. Лістинг функції підтвердження послідовності літер

Лістинг функції обробки коментарів показно на рис. 9.

```
function TPLParser.CommentHandler( C : Char ) : THandlerResult;
begin
    if ( C <> SPACE ) then
      begin
         // Знайдено закінчення коментаря
         if ( EndsWith( FToken + C, COMMENT_END ) ) then
           begin
              FToken := FToken + C;
              FFireMessage('Найден комментарий (' + FToken + ')');
              // !!!!! Зберігаємо коментар
              FToken := '';
              // Восстанавливаем старое значение
              FToken := FSavedToken;
              FMSState := FMSNextState;
           end
         else
           begin
              // Ні, ще не закінчення рядка
              FToken := FToken + C; //
           end;
      end
    else
      begin
         // Пробіл – частина коментаря
         FToken := FToken + C;
      end;
end;
```

Рис. 9. Лістинг функції обробки коментарів

Функція ***CommentHandler*** викликається в тому випадку, коли стан автомату дорівнює msComment, тобто на вхід аналізатора поступила послідовність «/*». Ця функція працює за алгоритмом, представленим блок – схемою на рис. 10.

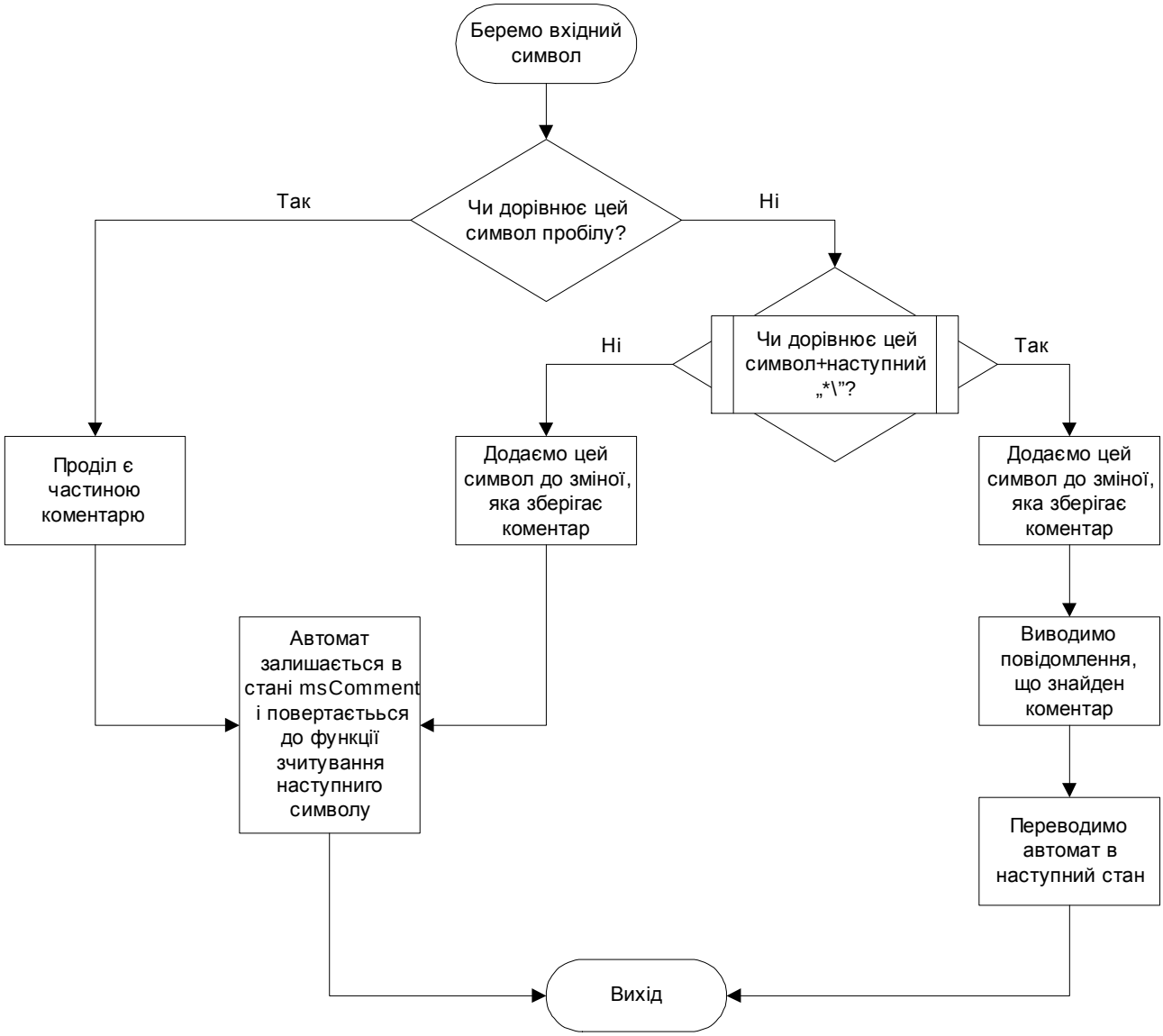


Рис.10. Блок – схема алгоритму функції CommentHandler

Продемонструємо роботу програми на прикладі. На рис.11 показано лістинг фрагмента програми мовою PL/1.

```
INDANI2: /* FOPMIPOBAHIE FAJLA DLA PE4ATI BEDOMOCTI INDANI */
  PROCEDURE (PARM) OPTIONS (MAIN);
/*-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i-i*/
/*  HA3HA4EHIE:                                                    */
/* HA OCHOBAHII 3 BXODH#X FAJLOB FOPMIPYETCA EDIH#J FAJL DLA PE4A-  */
 /* TI BEDOMOCTI INDANI PO PEPCOHIFICIPOBAHHOMY Y4ETY DLA PEH.FOHDA
*/
 /* B DBYX BAPIAHTAX: "BE3" I "C" DAHH#MI "OPGAHI3ACIA, CEX, TABEL$- */
 /* H#J HOMEP", DLIHOJ COOTBETCTBEHHO 946 I 957 BAJTOB.            */
```

Рис. 11. Лістинг фрагменту програми на мові PL/1

Після обробки модуля, на рис. 12, програма знайшла такі лексичні та синтаксичні конструкції: назву програми; ключові слова (PROCEDURE, OPTIONS); коментарі; араметри головної процедури; параметри опцій; розділювач операцій.

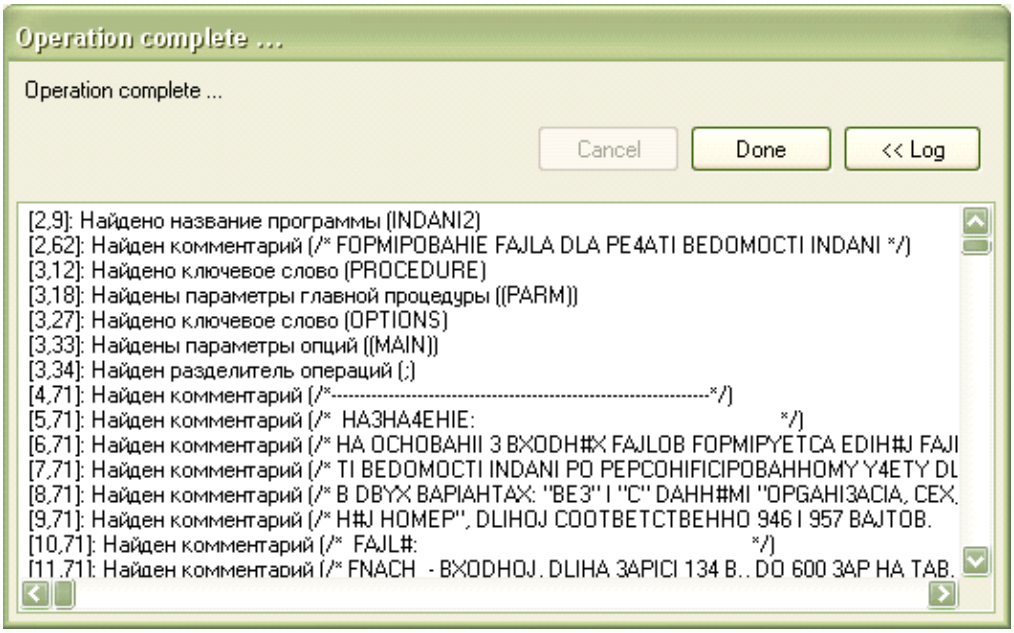


Рис. 12. Результат роботи аналізатора

Перевіримо програму на розпізнавання помилок. На рис.13 показано фрагмент коду програми мовою PL/1, написаний з помилкою:

Як ми можемо побачити на рис. 14, програма знайшла помилку і видала повідомлення.

```
INDANI2: /* FOPMIPOBAHIE FAJLA DLA PE4ATI BEDOMOCTI INDANI */
  PROC(PARM;) OPTIONS(MAIN);
```

Рис.13. Програма мовою PL/1, написана з помилкою

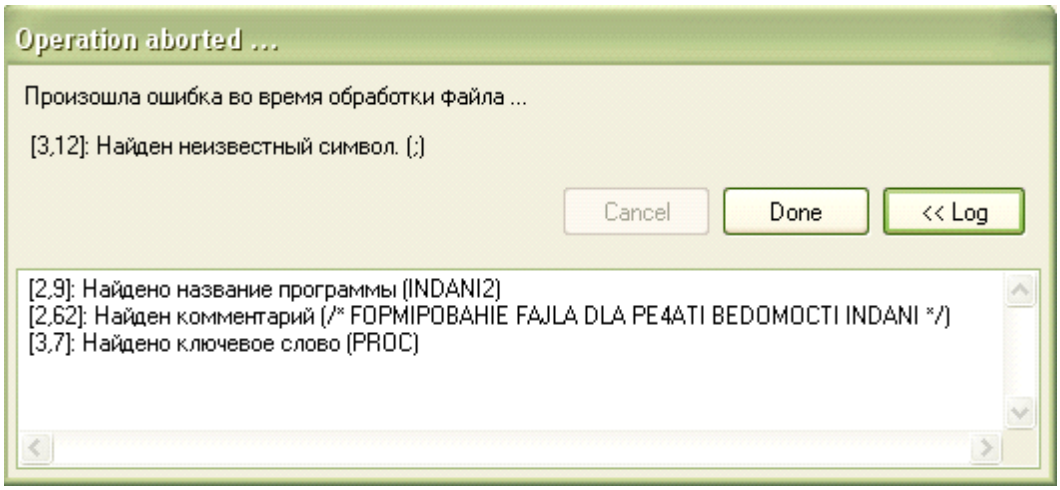


Рис. 13. Приклад роботи аналізатора при наявності помилки

Програмово синтаксичний аналізатор реалізовано в середовищі Delphi. Для прискорення обробки вхідного тексту процес аналізу організований у паралельних потоках.

У результаті синтаксичного аналізу ми отримаємо дерево [9], розбору на базі якого буде побудована блок-схема алгоритму.

Висновки. Графічне представлення програми більш зрозуміло відображає її структуру. Використання блок-схем дозволить швидше і якісно розроблювати та підлагоджувати програми, а також полегшить їх супровід. Внаслідок всього вищесказаного, представляє зацікавленість система реінжинірингу, в рамках якої буде представлена можливість визначення алгоритму в графічному вигляді, подібно блок-схемі, але з елементами потокового програмування та використанням у повному обсязі графічних та інтерактивних можливостей комп'ютера. Це не лише полегшить сприйняття вже написаного коду, але й суттєво прискорить процес написання програм. У подальшому планується розширити, розробити програму реінжинірингу структур даних із програмного коду PL/1, після цього приступити до створення засобів рефакторингу програмного коду [10].

1. *Ксензов М.В.* Рефакторинг архитектуры программного обеспечения. – М.: ИСП РАН. – Препринт 4, 2004. – 12 с.
2. *Ксензов М.В.* Рефакторинг архитектуры программного обеспечения: выделение слоев. – М.: ИСП РАН. – Труды ИСП РАН, 2004.
3. *Богданов В.Л., Гордєєв В.С.* Практический опыт написания синтаксического анализатора языка программирования Кобол - http://se.math.spbu.ru/reeng.html.
4. *ГОСТ* 19.003-80 ЕСПД (Единая Система Программной Документации). Схемы алгоритмов и программ. Обозначения условные и графические.
5. *ГОСТ* 19.002-80 ЕСПД (Единая Система Программной Документации). Схемы алгоритмов и программ. Правила выполнения.
6. *Демин А.Ю., Гусев А.В.* Визуальное программирование программ на основе блок-схем. // Материалы научно-практической конференции “Новые информационные технологии в университетском образовании” . – Новосибирск: Издательство ИДМИ, 1999.– 227с.
7. *Легалов А.И.* Основы разработки трансляторов. – http://www.softcraft.ru/translat/lect/content.shtml.
8. *Ахо А.В., Сети Р., Ульман Дж.* Компиляторы: принципы, технологии и инструменты. – Киев: Издательство «Вильямс», 2003. – 768 с.
9. *Серебряков В.А.* Лекции по конструированию компиляторов. Москва: Издательство МГУ, 1993 – 175с.
10. *Мартин Ф.* Рефакторинг: Улучшение существующего кода. – Москва: Издательство «Символ Плюс», 2002 – 390 с.