Лабораторна робота №2

з дисципліни

Теорія формальних мов та компіляція

на тему

«Розробка синтаксичного аналізатора»

Варіант №17

|  |  |
| --- | --- |
| Виконав студент  групи КВ-64М  Подольський С. В.  залікова книжка № КВ6415 | Перевірив:  Марченко О. І.  \_\_\_\_\_\_\_\_\_ |

1. **Постановка задачі**

Розробити програму синтаксичного аналізатора (СА) для підмножини мови програмування SIGNAL.

Програма має забезпечувати наступне:

* читання рядку лексем та таблиць, згенерованих лексичним аналізатором, який було розроблено в лабораторній роботі №1;
* синтаксичний аналіз (розбір) програми, поданої рядком лексем (алгоритм розбору вибирається згідно з варіантом);
* побудову дерева розбору;
* формування таблиць ідентифікаторів та різних констант з повною інформацією, необхідною для генерування коду;
* формування лістингу вхідної програми з повідомленнями про лексичні та синтаксичні помилки.

**Входом** СА має бути наступне:

* закодований рядок лексем;
* таблиці ідентифікаторів, числових, символьних та рядкових констант (якщо це передбачено граматикою варіанту), згенеровані лексичним аналізатором;
* вхідна програма на підмножині мови програмування SIGNAL згідно з варіантом (необхідна для формування лістнигу програми).

**Виходом** СА має бути наступне:

* дерево розбору вхідної програми;
* таблиці ідентифікаторів та різних констант з повною інформацією, необхідною для генерування коду;
* лістинг вхідної програми з повідомленнями про лексичні та синтаксичні помилки.

1. **Індивідуальне завдання згідно із варіантом**

Висхідний розбір за алгоритмом граматик простого передування

**Задана граматика підмножини мови SIGNAL:**

<signal-program> → <program>

<program> → PROGRAM <procedure-identifier> ; <block>.

<block> → BEGIN <statements-list> END

<statements-list> → <statement> <statements-list> | <empty>

<statement> → LOOP <statements-list> ENDLOOP ; | FOR <variable-identifier> := <loop-declaration> ENDFOR ;

<loop-declaration> → <expression> TO <expression> DO <statements-list>

<expression> → <summand> <summands-list> | - <summand> <summands-list>

<summands-list> → <add-instruction> <summand> <summands-list> | <empty>

<add-instruction> → + | -

<summand> → <variable-identifier> | <unsigned-integer>

<variable-identifier> → <identifier>

<procedure-identifier> → <identifier>

<identifier> → <letter><string>

<string> → <letter><string> | <digit><string> | <empty>

<unsigned-integer> → <digit><digits-string>

<digits-string> → <digit><digits-string> | <empty>

<digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<letter> → A | B | C | D | ... | Z

**Перетворена граматика:**

Граматика перетворена з метою задовольняти всі обмеження, що накладаються на граматику простого передування за визначенням граматики простого передування, а саме  
G = (*N*, Σ, *P*, *S*) is a simple precedence grammar if all the production rules in *P* comply with the following constraints:

* There are no [erasing rules](http://en.wikipedia.org/wiki/Erasing_rule) (ε-productions)
* There are no [useless rules](http://en.wikipedia.org/wiki/Useless_Rules) (unreachable symbols or unproductive rules)
* For each pair of symbols *X*, *Y* (*X*, *Y* \in (*N* ∪ Σ)) there is only one [Wirth-Weber precedence relation](http://en.wikipedia.org/wiki/Wirth-Weber_precedence_relationship)
* G is [uniquely inversible](http://en.wikipedia.org/wiki/Uniquely_Inversible_Grammar)

<signal-program> → <program>

<program> → PROGRAM <identifier> ; <block> .

* 1. <block> → BEGIN <statements-list> END
  2. <block> → BEGIN END

<statements-list> → <statements>

<statements> → <statement>

<statements> → <statement> <statements>

<statement> → LOOP <statements-list> ENDLOOP ;

<statement> → LOOP ENDLOOP ;

<statement> → FOR <identifier> := <loop-declaration> ENDFOR ;

<loop-declaration> → <expression> TO <expression> DO <statements-list>

<loop-declaration> → <expression> TO <expression> DO

<expression> → <summands-list>

<expression> → - <summands-list>

<summands-list> → <summand>

<summands-list> → <summand> - <summands-list>

<summands-list> → <summand> + <summands-list>

<summand> → <identifier>

<summand> → <unsigned-integer>

<unsigned-integer> → N1 | N2 | … | NK

<identifier> → T1 | T2 | … | TN

Де N1, N2, …, NK – константи користувача, згорнуті лексичним аналізатором;

T1, T2, …, TN – ідентифікатори користувача, згорнуті лексичним аналізатором.

1. **Лістинг програми синтаксичного аналізатора (основна частина)**

using System.Collections.Generic;

using System.Linq;

namespace SyntaxAnalyzer

{

/// <summary>

/// Parse Tree

/// </summary>

public class Tree : List<Tree>

{

public string Token { get; set; }

public Tree() : base() { }

public Tree(IEnumerable<Tree> collection) : base(collection) { }

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using RelationsComputing;

namespace SyntaxAnalyzer

{

/// <summary>

/// Simple precedence parser

/// </summary>

/// <see href="http://en.wikipedia.org/wiki/Simple\_precedence\_parser"/>

public class Parser

{

/// <summary>

/// Wirth-Weber precedence relationship table

/// </summary>

RelationTable relationTable;

/// <summary>

/// Create syntax analyser instance

/// </summary>

/// <param name="relationTable">Wirth-Weber precedence relationship table</param>

public Parser(RelationTable relationTable)

{

this.relationTable = relationTable;

}

/// <summary>

/// Parse input

/// </summary>

/// <param name="input">A list of tuples.

/// <para>Each tuple is defined by two items.</para>

/// <para>Item1: symbol code assigned by scanner;</para>

/// <para>Item2: symbol;</para></param>

public Tree Parse(List<Tuple<int, string>> input)

{

/\* Start with a stack with only the starting marker \*/

var stack = new List<Tree> { new Tree() { Token = RelationTable.limitMarker } };

/\* Start with the string being parsed (input) ended with an ending marker \*/

input.Add(Tuple.Create(-1, RelationTable.limitMarker));

/\* Stack of pivots offsets indices

\* Each element is an index of such token B that satisfies a condition "A < B",

\* where A is a previous token before B

\*/

var pivotsOffsets = new Stack<int>();

/\* While not (Stack equals to $S and Input equals to $) (S = Initial symbol of the grammar) \*/

while (stack.Last().Token != "<signal-program>" || input.First().Item2 != RelationTable.limitMarker)

{

/\* Search in the table the relationship between Top(stack) and NextToken(Input) \*/

switch (relationTable[stack.Last().Token, input.First().Item2])

{

case Relation.Lower:

/\* Push relationship to pivots offsets stack \*/

pivotsOffsets.Push(stack.Count);

goto case Relation.Equal;

case Relation.Equal:

/\* Push next token to stack \*/

stack.Add(new Tree { Token = input.First().Item2 });

/\* Remove remove next token from input \*/

input.RemoveAt(0);

break;

case Relation.Greater:

/\* Search production to reduce \*/

var reduced = Reduce(stack.Skip(pivotsOffsets.Peek()));

/\* Remove pivot from stack \*/

stack = stack.Take(pivotsOffsets.Peek()).ToList();

/\* Pop index from pivots offsets stack if the relationship between the Non-terminal

\* from the production and first symbol in the stack (Starting from top) is not equal to "<"

\*/

if (relationTable[stack.Last().Token, reduced.Token] == Relation.Equal)

pivotsOffsets.Pop();

/\* Push reduced non-terminal to stack instead of pivot \*/

stack.Add(reduced);

break;

default:

throw new SyntaxException("unexpected symbol '" + input.First().Item2 + "'");

}

}

return stack.Last();

}

/// <summary>

/// Serch production to reduce in all productions

/// </summary>

/// <param name="pivot">List of tokens to reduce</param>

/// <returns>Reduced non-terminal</returns>

Tree Reduce(IEnumerable<Tree> pivot)

{

foreach (var nonTerminal in RelationTable.NonTerminals)

foreach (var production in relationTable.Productions(nonTerminal))

if (Enumerable.SequenceEqual(from tree in pivot select tree.Token, production))

return new Tree(pivot) { Token = nonTerminal };

throw new SyntaxException("can not reduce the following code\n" + pivot.Aggregate("", (code, tree) => code + tree.Token + "\n"));

}

}

}

1. **Контрольні приклади**

PROGRAM sample;

BEGIN

LOOP

ENDLOOP;

LOOP

FOR i := 1 TO -4+20+i DO

FOR j := 2 TO 42 DO

LOOP

ENDLOOP;

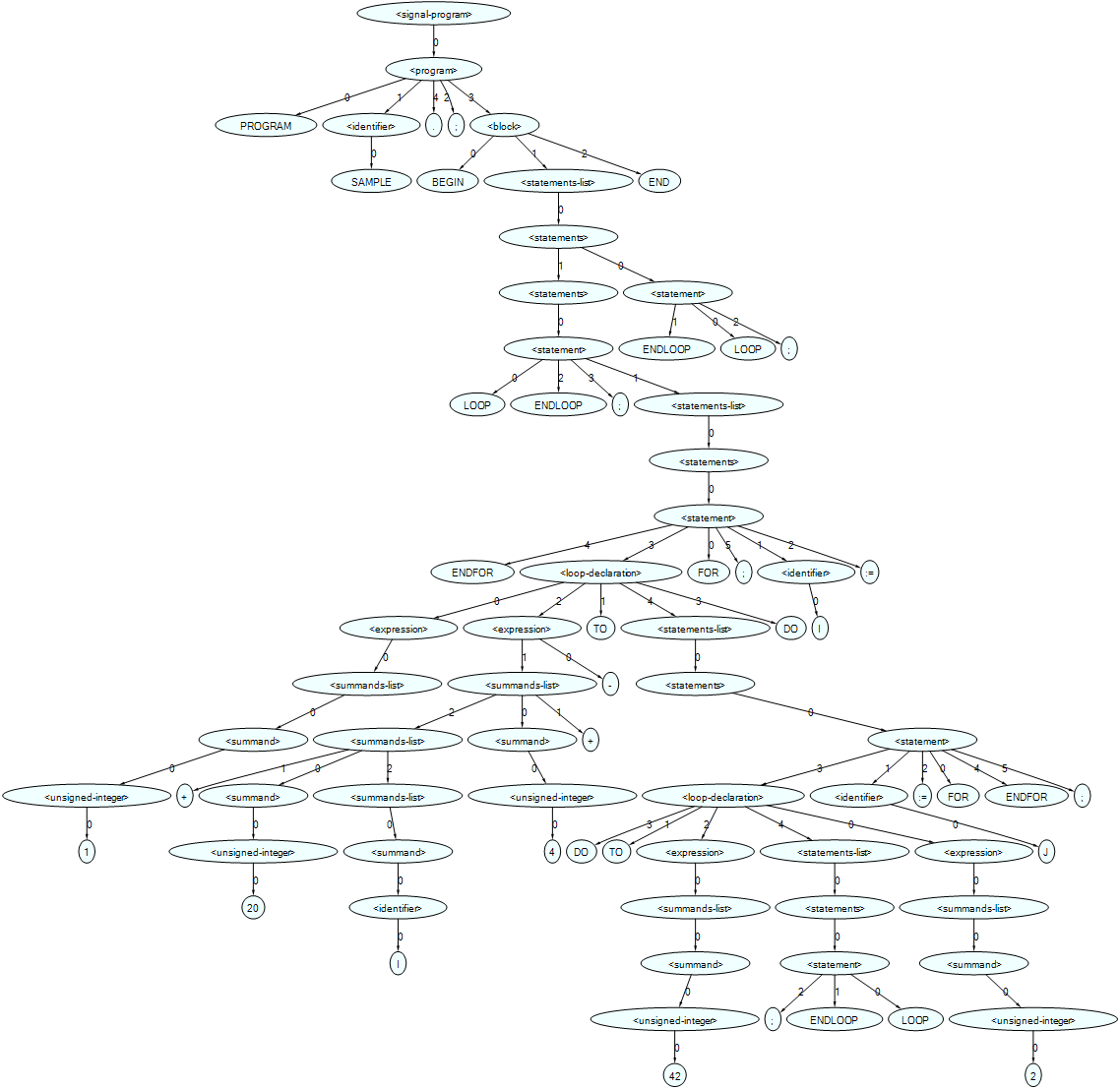
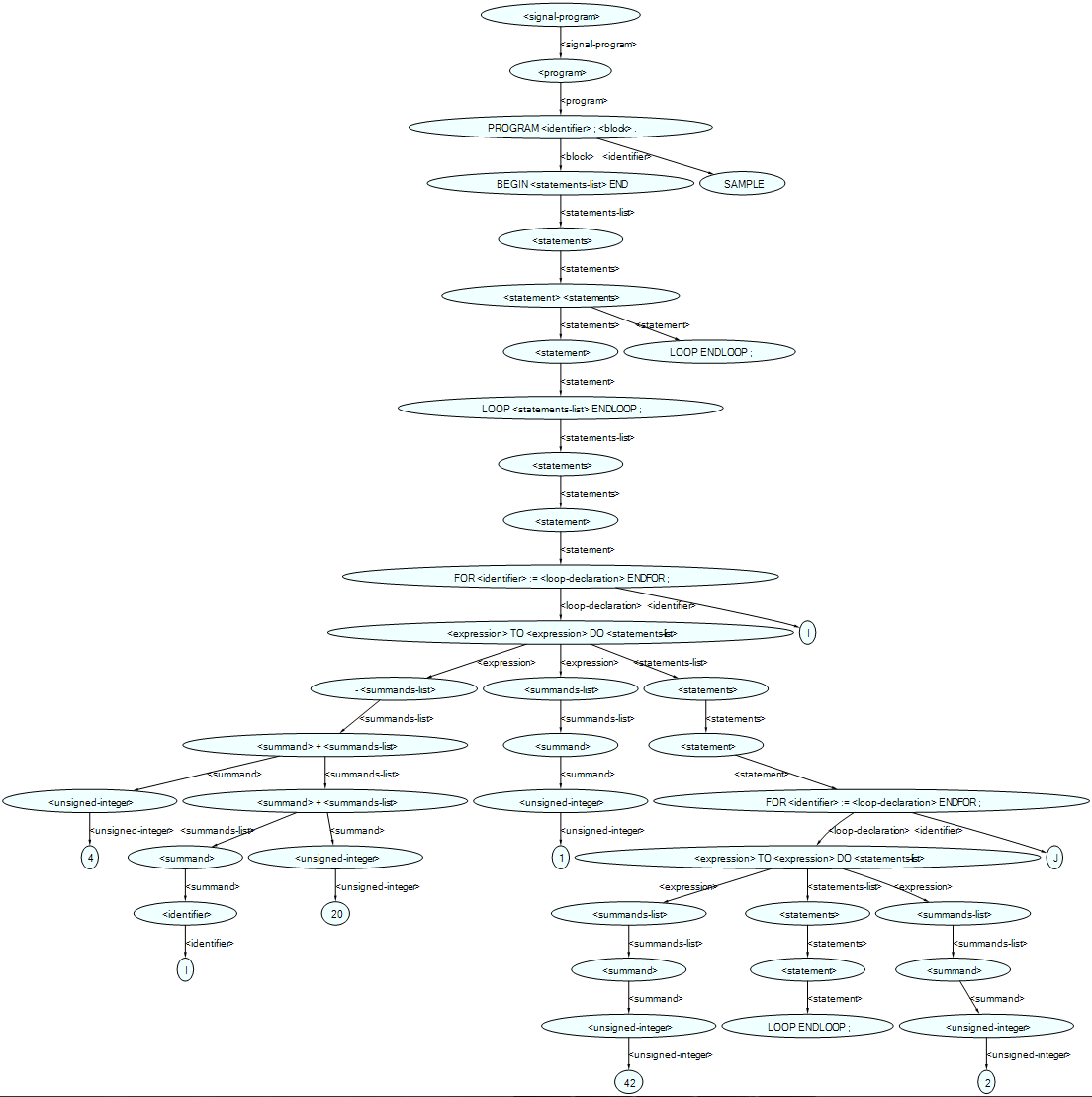
ENDFOR;

ENDFOR;

ENDLOOP;

(\*comment\*)

END.

|  |  |
| --- | --- |
| PROGRAM sample;  BEGIN  LOOP  ENDLOOP(\*;\*)  LOOP  FOR i := 1 TO -4+20+i DO  FOR j := 2 TO 42 DO  LOOP  ENDLOOP;  ENDFOR;  ENDFOR;  ENDLOOP;  (\*comment\*)  END. |  |
| PROGRAM sample;  BEGIN  LOOP  (\*ENDLOOP;\*)  LOOP  FOR i := 1 TO -4+20+i DO  FOR j := 2 TO 42 DO  LOOP  ENDLOOP;  ENDFOR;  ENDFOR;  ENDLOOP;  (\*comment\*)  END. |  |