**Київський національний університет імені Тараса Шевченка**

**Факультет комп’ютерних наук та кібернетики**

**Звіт**

з дисципліни «Системне програмування»

по четвертій лабораторній роботі

на тему: «**LL(k)-Parser (k=1)**»

студента 3 курсу, групи МІ-31

спеціальності 122 «Комп’ютерні науки»

Рубан Д.С.

Київ –2023

Постановка задачі:

Розробити LL(1)-синтаксичний аналізатор для заданої граматики, який будує AST або визначає та локалізує синтаксичну помилку:

1) запрограмувати всі необхідні функції: Firsk(k), Follow(k), побудова таблиці управління, власне аналізатор по таблиці

2) запрограмувати допоміжні функції: пошук епсилон-нетерміналів, читання і розбір введеної граматики, тощо.

Загальний підхід до вирішення задачі:

Створити клас граматики, що матиме такі методи(мінімум):

* First()
* Follow()
* FindEpsilonNonTerminals()
* AnalyseString()
* ConstructControlTable()

Та зберігатиме термінали, нетермінали, контрольну таблицю та епсилон.

Реалізація:

* **First()**

public HashSet<string> First(string symbol, int k=1)

{

if (k < 0) throw new ArgumentException("k cannot be negative.");

else if (!ContainsSymbol(symbol))

throw new ArgumentException($"symbol '{symbol}' is not known");

var result = new HashSet<string>();

if (k == 0)

return result;

else if (IsTerminal(symbol))

{

result.Add(symbol);

return result;

}

else if (!RulesDictionary.ContainsKey(symbol))

throw new ArgumentException($"{symbol} is a not productive");

foreach (var production in RulesDictionary[symbol])

{

var Y = ParseProduction(production);

var FirstY0 = First(Y[0]);

result.UnionWith(FirstY0);

if (Epsilon == null) continue;

if (EpsilonNonTerminals.Contains(symbol))

{

result.Add(Epsilon);

}

if (!EpsilonNonTerminals.Contains(Y[0]) || IsTerminal(Y[0]))

{

continue;

}

// while non-terminal is epsilon one, add also next non-terminal

for (int i = 0; i < Y.Count; i++)

{

result.UnionWith(First(Y[i]).Where(symbol => symbol != Epsilon));

var FirstYi = First(Y[i]);

if (!EpsilonNonTerminals.Contains(Y[i]))

{

break;

}

}

}

return result;

}

Ключові моменти:

1. Перевірка валідності параметрів:
   * if (k < 0) throw new ArgumentException("k cannot be negative.");: Генерує виняток, якщо k вказано від'ємним.
   * else if (!ContainsSymbol(symbol)) throw new ArgumentException($"symbol '{symbol}' is not known in this grammar");: Генерує виняток, якщо символ не є частиною граматики.
   * Ініціалізація змінної result для зберігання результату.
2. Обробка випадків k == 0 та терміналів: Якщо k дорівнює 0, повертається пустий результат. Якщо символ є терміналом, додає його до результату і повертає його.
3. Перевірка, чи наданий нетермінал є продуктивним: Якщо символ не має правил продукції, генерується виняток.
4. Обробка правил продукції для заданого нетерміналу:
   * *foreach (var production in RulesDictionary[symbol])*: Для кожного правила продукції для даного нетерміналу.
   * *var Y = ParseProduction(production);*: Розбір правила продукції на компоненти.
   * *var FirstY0 = First(Y[0]);*: Обчислення множини FIRST для першого символу правила продукції.
5. Обробка символу "ε" (порожнього рядка):
   * if (Epsilon == null) continue;: Якщо порожній рядок не визначений, продовжити до наступного правила.
   * if (EpsilonNonTerminals.Contains(symbol)) result.Add(Epsilon);: Якщо даний нетермінал може породжувати порожній рядок, додати "ε" до результату.
6. Додавання наступних символів до результату:
   * Якщо перший символ правила продукції не є нетерміналом або не може породжувати порожній рядок, продовжити з іншими правилами.
   * Якщо перший символ є нетерміналом та може породжувати порожній рядок, додати до результату всі можливі символи FIRST для цього нетерміналу, за виключенням "ε". Повторювати цей процес для інших символів у правилі, доки зустрічається нетермінал, який не породжує "ε".

* **Follow()**

public HashSet<string> Follow(string symbol, int k=1)

{

var result = new HashSet<string>();

if (symbol == StartNonTerminal && Epsilon != null)

{

result.Add(Epsilon);

}

foreach(var SymbolProductions in RulesDictionary)

{

foreach(var production in SymbolProductions.Value)

{

result.UnionWith(FollowFromProduction(symbol, SymbolProductions.Key, production));

}

}

return result;

}

private HashSet<string> FollowFromProduction(string symbol,string productionFrom,string productionTo)

{

var result = new HashSet<string>();

var producedSymbols = ParseProduction(productionTo);

if (!producedSymbols.Contains(symbol))

{

return new HashSet<string>();

}

for (int i = 0; i < producedSymbols.Count; i++)

{

if (producedSymbols[i] == symbol)

{

if (i == producedSymbols.Count - 1 && productionFrom != symbol)

{

result.UnionWith(Follow(productionFrom));

}

else

{

var nextSymbolIndex = i + 1;

if (nextSymbolIndex >= producedSymbols.Count) break;

result.UnionWith(First(producedSymbols[nextSymbolIndex]).Where(symbol => symbol != Epsilon));

while (EpsilonNonTerminals.Contains(producedSymbols[nextSymbolIndex]))

{

nextSymbolIndex++;

if (nextSymbolIndex >= producedSymbols.Count)

{

result.UnionWith(Follow(productionFrom));

return result;

};

result.UnionWith(First(producedSymbols[nextSymbolIndex]).Where(symbol => symbol != Epsilon));

}

}

}

}

return result;

}

Ключові моменти:

1. *public HashSet<string> Follow(string symbol, int k=1*): Метод, який обчислює множину FOLLOW для заданого символу граматики.

* Створення пустої множини result.
* Додавання "ε" до множини, якщо оброблюваний символ є початковим нетерміналом та "ε" визначено в граматиці.
* Ітерація через всі правила продукції в граматиці та виклик *FollowFromProduction* для обробки кожного правила.
* Повернення об'єднаної множини результатів.

1. *private HashSet<string> FollowFromProduction(string symbol, string productionFrom, string productionTo):* Метод, який обчислює множину FOLLOW для даного символу з певного правила продукції.

* Створення пустої множини *result*.
* Розбір правила продукції на компоненти.
* Перевірка, чи містить це правило оброблюваний символ. Якщо ні, повертається порожня множина.
* Ітерація через всі символи правила.
* Перевірка, чи оброблюваний символ є на останньому місці та чи він не є самим собою. Якщо так, то додається множина FOLLOW для символа, з якого починалася продукція.
* Якщо оброблюваний символ не є останнім в правилі, обчислюється FIRST для наступного символу та додається до множини результатів. При цьому оброблюються всі можливі "ε"-переходи.
* Повертається множина результатів.
* **FindEpsilonNonTerminals()**

private void AddEpsilonNonTerminals(HashSet<string> EpsilonNonTerminals)

{

if (Epsilon == null) return;

if (EpsilonNonTerminals.Count == 0)

{

EpsilonNonTerminals.UnionWith(RulesDictionary

.Where(production => production.Value.Contains(Epsilon))

.Select(production => production.Key));

}

EpsilonNonTerminals.UnionWith(RulesDictionary

.Where(production =>

production.Value.Any(prodValue =>

{ var productionResult = ParseProduction(prodValue);

return productionResult.All(symbol => symbol == Epsilon || EpsilonNonTerminals.Contains(symbol)); })).Select(production => production.Key));

}

private HashSet<string> FindEpsilonNonTerminals()

{

var result = new HashSet<string>();

int countBefore;

do{

countBefore = result.Count;

AddEpsilonNonTerminals(result);

} while (result.Count - countBefore > 0);

return result;

}

Ключові моменти:

1. *private void AddEpsilonNonTerminals(HashSet<string> EpsilonNonTerminals):* Метод, який додає до заданої множини нетерміналів ті, які можуть породжувати порожній рядок ("ε").

* Перевірка, чи визначено "ε" в граматиці. Якщо ні, просто повертається.
* Якщо задана множина EpsilonNonTerminals порожня, ітерація через правила продукції та додавання нетерміналів, які можуть породжувати "ε".
* Подальша ітерація через правила продукції та додавання тих нетерміналів, для яких всі символи правила продукції або порожні, або належать до EpsilonNonTerminals.

1. *private HashSet<string> FindEpsilonNonTerminals():* Метод, який знаходить всі нетермінали, які можуть породжувати порожній рядок, та повертає їх у множині.

* Створення порожньої множини result.
* Виклик методу AddEpsilonNonTerminals в циклі, доки кількість елементів у множині result збільшується після виклику.
* Повертається результат, тобто множина нетерміналів, які можуть породжувати "ε".
* **ConstructControlTable()**

public void ConstructControlTable()

{

ControlTable = new ControlTable(Terminals);

for (int i = 0; i < RulesList.Count; i++)

{

var A = RulesList[i].SymbolFrom;

var w = ParseProduction(RulesList[i].Production)[0];

var finalSet = ConcatSets(First(w), Follow(A));

foreach (var symbol in finalSet)

{

ControlTable.AddCell(A, symbol, i);

}

}

}

Ключові моменти:

1. Створення нового екземпляру ControlTable з терміналами граматики:

*ControlTable = new ControlTable(Terminals);*

1. Ітерація через всі правила продукції граматики:

* for (int i = 0; i < RulesList.Count; i++): Цикл по всіх правилах продукції.

1. Отримання нетерміналу A та першого символу з продукції w:

* var A = RulesList[i].SymbolFrom;: Нетермінал, для якого буде заповнюватися рядок в таблиці управління.
* var w = ParseProduction(RulesList[i].Production)[0];: Перший символ правила продукції.

1. Обчислення множини finalSet, яка включає FIRST(w) та FOLLOW(A):

* *var finalSet = ConcatSets(First(w), Follow(A));*: Об'єднання множин FIRST(w) та FOLLOW(A).

1. Заповнення відповідних комірок таблиці управління:

* foreach (var symbol in finalSet): Ітерація через символи множини finalSet.
* ControlTable.AddCell(A, symbol, i);: Додавання індексу правила продукції у відповідну комірку таблиці для пари (A, symbol).
* **AnalyseString()**

public string AnalyseString(string StringOfLexemes)

{

List<int> ruleSequence = new List<int>();

Stack<Leaf> stack = new Stack<Leaf>();

var str = StringOfLexemes;

AstRoot = new Leaf(StartNonTerminal, this);

stack.Push(AstRoot);

string header = "| STACK | RULE INDEX | RULE | CURRENT LEXEME |";

string divider = new string('\_', header.Length);

Console.WriteLine(header);

Console.WriteLine(divider);

ParseString(str, stack, ruleSequence, false);

if (Epsilon == null && stack.Count > 0)

{

throw new Exception("No epsilon but stack is not empty after reading all string");

}

ParseString(str, stack, ruleSequence, true);

Console.WriteLine(divider);

if (stack.Count > 0)

{

throw new Exception("Stack is not empty after reading all string");

}

return string.Join(", ", ruleSequence);

}

private void ParseString(string str, Stack<Leaf> stack, List<int> ruleSequence, bool destroy)

{

int i = 0;

while (i < str.Length && stack.Count > 0)

{

var topLeaf = stack.Peek();

var topStackLexeme = stack.Peek().Symbol;

var currentLexeme = destroy ? Epsilon : str[i].ToString();

if (topStackLexeme == Epsilon)

{

stack.Pop();

continue;

}

if (topStackLexeme == currentLexeme)

{

stack.Pop();

i++;

continue;

}

if (IsTerminal(topStackLexeme))

{

throw new Exception($"Expected lexeme {topStackLexeme}");

}

int ruleIndex = ControlTable.GetRule(topStackLexeme, currentLexeme);

ruleSequence.Add(ruleIndex + 1);

var ruleProduct = ParseProduction(RulesList[ruleIndex].Production);

List<Leaf> newLeaves = new List<Leaf>();

foreach (var temp in ruleProduct)

{

var newLeaf = new Leaf(temp, this);

newLeaves.Add(newLeaf);

topLeaf.AddChild(newLeaf);

}

newLeaves.Reverse();

stack.Pop();

ruleProduct.Reverse();

foreach (var temp in newLeaves)

{

stack.Push(temp);

}

Console.WriteLine($"|{string.Join(" ", stack.Select(s => s.Symbol)),28} | {ruleIndex + 1,12} | {RulesList[ruleIndex],30} | {currentLexeme,17} |");

}

}

Ключові моменти (метод, як і кілька попередніх, розбито на 2):

*Метод AnalyseString:*

1. Ініціалізація змінних:

* List<int> ruleSequence = new List<int>();: Список для зберігання індексів правил, які використовуються під час аналізу.
* Stack<Leaf> stack = new Stack<Leaf>();: Стек для виконання аналізу стекового автомата.
* var str = StringOfLexemes;: Введена строка для аналізу.
  + AstRoot = new Leaf(StartNonTerminal, this);: Створення кореневого листа AST, який представляє початковий нетермінал граматики.

1. stack.Push(AstRoot);: Додавання кореневого листа на стек.
2. Виведення заголовка таблиці для відстеження аналізу.
3. Виклик методу ParseString для аналізу строки з увімкненою та вимкненою функцією destroy.
4. Перевірка стеку на пустоту після завершення аналізу та виведення результатів аналізу.
5. Повернення рядка, який містить індекси використаних правил під час аналізу.

*Метод ParseString:*

1. *private void ParseString(string str, Stack<Leaf> stack, List<int> ruleSequence, bool destroy):* Метод, який виконує аналіз строки та маніпулює стеком та AST.
2. int i = 0;: Індекс для проходження по введеній строці.
3. Основний цикл аналізу:

* Перевірка, чи введена строка ще не закінчилася та чи стек не порожній.
* Отримання верхнього листа стеку та поточного лексему зі строки або "ε".
* Обробка різних сценаріїв:
* Якщо верхній лист стеку є "ε", видаляє його зі стеку.
* Якщо верхній лист стеку співпадає з поточною лексемою, видаляє його зі стеку та збільшує індекс у введеній строці.
* Якщо верхній лист стеку є терміналом, генерує виняток, якщо не співпадає з поточною лексемою.
* В іншому випадку викликається ControlTable.GetRule для отримання індексу правила з таблиці управління.
* Додає індекс правила до списку ruleSequence.
* Розбирає правило продукції та створює листи AST для кожного символу продукції.
* Додає нові листи до стеку у зворотньому порядку.
* Виведення інформації про стан стеку та правило продукції.
* Перевірка наявності "ε" в граматиці:
* Якщо "ε" не визначено, але стек не порожній, генерує виняток.
* Додатковий виклик методу ParseString зі зміненою конфігурацією destroy.

Клас граматики:

Граматика зберігається так:

public partial class LL1Grammar

{

public LL1Grammar(string startNonTerminal)

{

RulesDictionary = new Dictionary<string, List<string>>();

Terminals = new List<string>();

NonTerminals = new List<string>();

StartNonTerminal = startNonTerminal;

NonTerminals.Add(StartNonTerminal);

EpsilonNonTerminals = new HashSet<string>();

RulesList = new List<Rule>();

}

public Dictionary<string, List<string>> RulesDictionary { get; }

public List<Rule> RulesList { get; }

public List<string> Terminals { get; }

public List<string> NonTerminals { get; }

public string? Epsilon { get; set; }

public string StartNonTerminal { get; }

public HashSet<string> EpsilonNonTerminals { get; private set; }

}

Тестування:

**Граматика:**

S -> ACB | Cbb | Ba

A -> da | BC

B -> g | Є

C -> h | Є

**First / Follow:**

Код

try

{

foreach (var nonTerminal in grammar.NonTerminals)

{

Console.WriteLine($"FIRST for {nonTerminal}: {"{"} {string.Join(", ", grammar.First(nonTerminal))} {"}"}");

}

Console.WriteLine();

foreach (var nonTerminal in grammar.NonTerminals)

{

Console.WriteLine($"FOLLOW for {nonTerminal}: {"{"} {string.Join(", ", grammar.Follow(nonTerminal))} {"}"}");

}

Console.WriteLine();

Console.WriteLine($"Epsilon symbols are: {"{"} {string.Join(", ", grammar.EpsilonNonTerminals)} {"}"}");

Console.WriteLine();

}

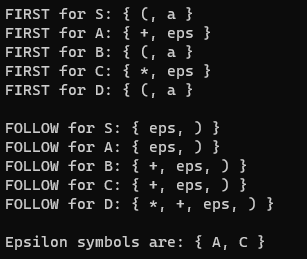
catch (Exception e)

{

Console.WriteLine(e.Message);

}

Результат



**Контрольна таблиця:**

Код:

// for rule A -> w

Console.WriteLine("RULE | RULE NUMBER | FIRST(w) + FOLLOW(A)");

Console.WriteLine("----------------------------------------------------");

for (int i = 0; i < grammar.RulesList.Count; i++)

{

var A = grammar.RulesList[i].SymbolFrom;

var w = grammar.ParseProduction(grammar.RulesList[i].Production)[0];

ruleProduction[i] = grammar.ConcatSets(grammar.First(w), grammar.Follow(A));

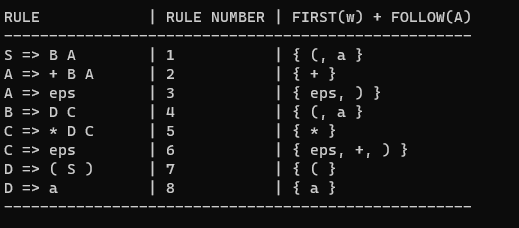
Console.WriteLine($"{grammar.RulesList[i],-15} | {i+1,-11} | {"{"} {string.Join(", ", ruleProduction[i])} {"}"}");

}

Console.WriteLine("----------------------------------------------------");

Console.WriteLine();

Результат:



**Аналіз рядка:**

Код:

try

{

string w1 = "(a+a)\*a";

Console.WriteLine($"Result of parsing {w1}:\n");

Console.WriteLine("\n" + grammar.AnalyseString(w1));

}

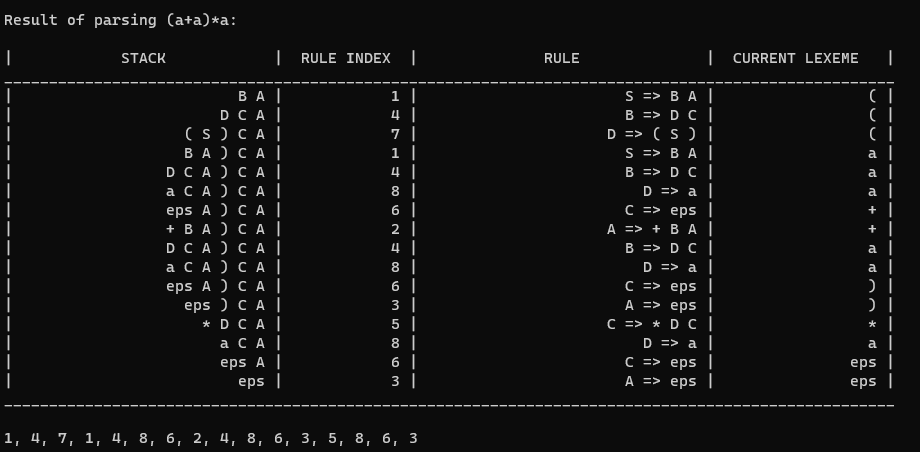
catch(Exception e)

{

Console.WriteLine(e.Message);

}

Результат:



Інший рядок:

try

{

string w1 = "(a+a)\*(a";

Console.WriteLine($"Result of parsing {w1}:\n");

Console.WriteLine("\n" + grammar.AnalyseString(w1));

}

catch(Exception e)

{

Console.WriteLine(e.Message);

}

Результат:

