Zadání: Referát č. 9 (Dynamické programování)

Navrhněte a detailně popište polynomiální algoritmus, který řeší následující problém:
VSTUP: Bezkontextová gramatika G v Chomského normální formě a slovo w.
VÝSTUP: Celkový počet různých derivačních stromů, které odpovídají derivacím slova w v gramatice G.
Nápověda: Vyjděte z algoritmu Cocke-Younger-Kasami, který vhodným způsobem upravte.

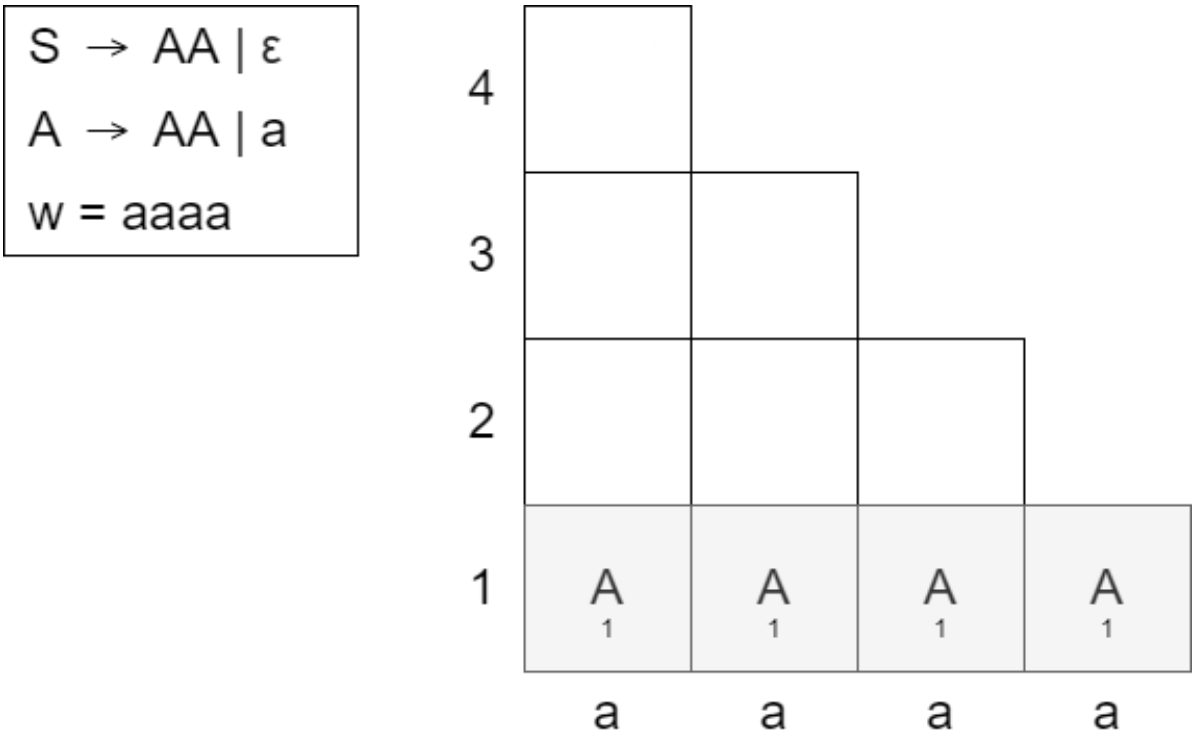
Vypracování:

$n = |w|, r = |G|$

Algoritmus Cocke-Younger-Kasami (CYK) standartně na vstupu přijímá bezkontextovou gramatiku v Chomského normální formě (CNF) a slovo a je schopný určit, zda je dané slovo možné pomocí této gramatiky vygenerovat. Jelikož původní algoritmus vrací pouze hodnoty true/false, tak je pro zjištění celkového počtu různých derivačních stromů nutné jej upravit.

Tento upravený algoritmus nejprve vytvoří tabulku o n řádcích a n sloupcích. V této tabulce poté prochází jednotlivé řádky zespoda nahoru a vždy nejprve projde celý řádek zleva doprava, než se přesune na řádek nad ním.

V nejspodnějším řádku tabulky tento algoritmus funguje trochu jinak, než ve zbytku tabulky a to tak, že jednotlivé buňky naplní neterminály, které dle gramatiky generují přímo dané terminály daného slova.



Obr. 1 - Vyplnění prvního řádku tabulky

Následně poté na zbylých řádcích pro vybranou buňku vybírá vždy dvojice buněk a provádí mezi nimi kartézský součin jejich neterminálů. Pokud je dvojice neterminálů vycházející z tohoto součinu součástí některého z pravidel gramatiky, tak jsou jejich počty duplikátů vzájemně vynásobeny a přičteny k počtu duplikátů neterminálu ve vybrané buňce.

S → AA | ε
A → AA | a
w = aaaa

	a	a	a	a
4	S,A 2 2			
3	S,A 2 2	S,A 2 2		
2	S,A 1 1	S,A 1 1	S,A 1 1	
1	A 1	A 1	A 1	A 1

Obr. 2 - První krok při vyplňování posledního řádku tabulky

S → AA | ε
A → AA | a
w = aaaa

	a	a	a	a
4	S,A 3 3			
3	S,A 2 2	S,A 2 2		
2	S,A 1 1	S,A 1 1	S,A 1 1	
1	A 1	A 1	A 1	A 1

Obr. 3 - Druhý krok při vyplňování posledního řádku tabulky

S → AA | ε
A → AA | a
w = aaaa

	a	a	a	a
4	S,A 5 5			
3	S,A 2 2	S,A 2 2		
2	S,A 1 1	S,A 1 1	S,A 1 1	
1	A 1	A 1	A 1	A 1

Obr. 4 - Třetí krok při vyplňování posledního řádku tabulky

Po vyplnění tabulky získáme celkový počet různých derivačních stromů z nejvyššího prvku tabulky kde, pokud najdeme počáteční neterminál, tak jeho počet duplikátů je onen počet derivačních stromů.

Maximální počet různých derivačních stromů vychází z Catalanových čísel, což jsou taková přirozená čísla C_n, která jsou určena následujícím předpisem:

$$C_n = \frac{1}{n+1}\binom{2n}{n}, \forall n \geq 0$$

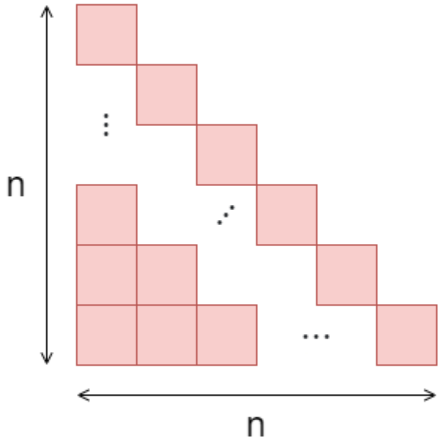
$$C_n = \frac{(2n)!}{(n+1)!\,n!}$$

Pro $n = 0, 1, 2, 3, 4, 5, 6, 7, \ldots$ jsou první hodnoty $C_n = 1, 1, 2, 5, 14, 42, 132, 429$ [1]

Paměťová složitost upraveného algoritmu CYK je:

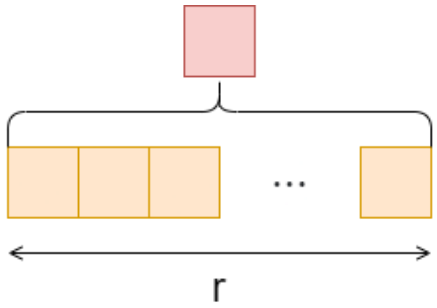
$$O(n^2 r\lceil \log_2(C_n)\rceil) \Rightarrow O(n^2\lceil \log_2(C_n)\rceil) \Rightarrow O\left(n^2\left\lceil \log_2\left(\frac{(2n)!}{(n+1)!\,n!}\right)\right\rceil\right)$$

Paměťová složitost je v rámci PSPACE a odvíjí se od velikosti tabulky použité při výpočtu CYK algoritmu, která obsahuje maximálně n řádků a n sloupců.



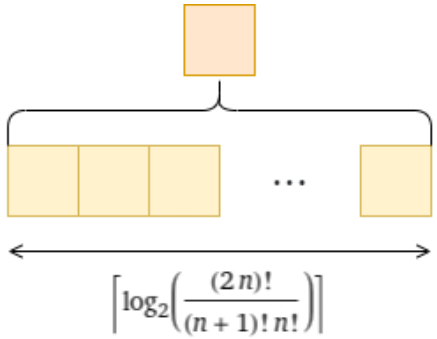
Obr. 5 - Tabulka použitá při výpočtu CYK algoritmu

Jedna buňka tabulky poté může obsahovat až r neterminálů. Každý neterminál obsahuje počet duplikátů, které v paměti zaznamenává pomocí binárního čísla. Zaznamenávání počtu duplikátů umožňuje eliminovat redundantní kroky při kombinaci stejných neterminálů tím, že je místo každého takového kroku přičten vzájemný součin jejich počtu duplikátů.



Obr. 6 - Buňka tabulky použité při výpočtu CYK algoritmu

Jelikož je maximální možný počet duplikátů daného neterminálu dán Catalanovým číslem, tak se od něj odvíjí i maximální velikost paměti pro zapsání jeho binárního čísla. Např. čísla 4-7 je možné zapsat pomocí binárního čísla o velikosti 3, jelikož jsou větší než $2^2 - 1$ a menší nebo rovno $2^3 - 1$.



Obr. 7 – Počet duplikátů neterminálu v buňce tabulky použité při výpočtu CYK algoritmu

Zdrojové kódy v C#:

```
public static List<List<List<NonterminalWithCount>>> CYK_ParseTableWithCounts(CFG grammar, string word)
{
    List<List<List<NonterminalWithCount>>> table = new List<List<List<NonterminalWithCount>>>(word.Length);

    if (word.Length > 0)
    {
        table.Add(new List<List<NonterminalWithCount>>(word.Length));

        for (int i = 0; i < word.Length; i++)
        {
            table[0].Add(new List<NonterminalWithCount>(grammar.Rules.Length));

            for (int j = 0; j < grammar.Rules.Length; j++)
            {
                if (grammar.Rules[j].Right.Contains(char.ToString(word[i])))
                {
                    table[0][i].Add(new NonterminalWithCount(grammar.Rules[j].Left, 1));
                }
            }
        }

        for (int i = 1; i < word.Length; i++)
        {
            table.Add(new List<List<NonterminalWithCount>>(word.Length - i));

            for (int j = 0; j < word.Length - i; j++)
            {
                table[i].Add(new List<NonterminalWithCount>(grammar.Rules.Length));

                for (int k = 0; k < grammar.Rules.Length; k++)
                {
                    for (int l = 0; l < i; l++)
                    {
                        for (int m = 0; m < table[l][j].Count; m++)
                        {
                            for (int o = 0; o < table[i - 1 - l][j + 1 + l].Count; o++)
                            {
                                string concatenatedNonterminal = table[l][j][m].Nonterminal + table[i - 1 - l][j + 1 + l][o].Nonterminal;

                                if (grammar.Rules[k].Right.Contains(concatenatedNonterminal))
                                {
                                    uint concatenatedNonterminalCount = table[l][j][m].NonterminalCount * table[i - 1 - l][j + 1 + l][o].NonterminalCount;
                                    int tableCellNonterminalIndex = table[i][j].FindIndex(nonterminal => nonterminal.Nonterminal == grammar.Rules[k].Left);

                                    if (tableCellNonterminalIndex > -1)
                                    {
                                        table[i][j][tableCellNonterminalIndex].NonterminalCount += concatenatedNonterminalCount;
                                    }
                                    else
                                    {
                                        table[i][j].Add(new NonterminalWithCount(grammar.Rules[k].Left, concatenatedNonterminalCount));
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    else
    {
        table.Add(new List<List<NonterminalWithCount>>(1));
        table[0].Add(new List<NonterminalWithCount>(grammar.Rules.Length));

        for (int i = 0; i < grammar.Rules.Length; i++)
        {
            if (grammar.Rules[i].Right.Contains("ε"))
            {
                table[0][0].Add(new NonterminalWithCount(grammar.Rules[i].Left, 1));
            }
        }
    }

    return table;
}
```

Obr. 8 - Funkce implementující upravený algoritmus CYK, která vrací vyplněnou tabulku

```
public static uint CYK_ParseTreeCount(CFG grammar, string word)
{
    List<List<List<NonterminalWithCount>>> table = CYK_ParseTableWithCounts(grammar, word);
    uint treeCount = 0;

    for (int i = 0; i < table[table.Count - 1][0].Count; i++)
    {
        if (table[table.Count - 1][0][i].Nonterminal == grammar.Start)
        {
            treeCount = table[table.Count - 1][0][i].NonterminalCount;
            break;
        }
    }

    return treeCount;
}
```

Obr. 9 - Funkce implementující upravený algoritmus CYK, která vrací celkový počet různých derivačních stromů

```
public readonly struct Rule
{
    public readonly string   Left;
    public readonly string[] Right;

    public Rule(string left, string[] right)
    {
        Left  = left;
        Right = right;
    }

    public override string ToString()
    {
        return Left + " → " + string.Join(" | ", Right);
    }
}

public readonly struct CFG
{
    public readonly string[] Nonterminals;
    public readonly string[] Terminals;
    public readonly Rule[]   Rules;
    public readonly string   Start;

    public CFG(string[] nonterminals, string[] terminals, Rule[] rules, string start)
    {
        Nonterminals = nonterminals;
        Terminals    = terminals;
        Rules        = rules;
        Start        = start;
    }

    public override string ToString()
    {
        return
            "G = (N, T, R, S)\n" +
            "N = [" + string.Join(", ", Nonterminals) + "]\n" +
            "T = [" + string.Join(", ", Terminals) + "]\n" +
            "R : " + string.Join("\n    ", Rules) + "\n" +
            "S = \"" + Start + "\"";
    }
}

public class NonterminalWithCount
{
    public readonly string Nonterminal;
    public          uint   NonterminalCount;

    public NonterminalWithCount(string nonterminal, uint nonterminalCount)
    {
        Nonterminal = nonterminal;
        NonterminalCount = nonterminalCount;
    }

    public override string ToString()
    {
        return "(" + Nonterminal + "," + NonterminalCount + ")";
    }
}
```

Obr. 10 - Struktury a třída, které jsou součástí funkcí implementujících upravený algoritmus CYK

```
/* INPUT(G, w) : context-free grammar in Chomsky normal form and word
 * G (context-free grammar), N (nonterminals), T (terminals), R (rules), S (start)
 * w (word) */
G = (N, T, R, S)
N = [A, B, C, AB, BA, BC, CC, S]
T = [a, b]
R : S → AB | BC
    A → BA | a
    B → CC | b
    C → AB | a
S = "S"
w = "baaba"

/* OUTPUT(bool) : At least 1 derivation tree? (original CYK)
 * CYK (Cocke-Younger-Kasami algorithm) */
CYK_ParseTable(G, w) : S,A,C
                       -              S,A,C
                       -              B                    B
                       S,A            B                    S,C                  S,A
                       B              A,C                  A,C                  B                    A,C
CYK(G, w) : True

/* OUTPUT(uint) : Count of different derivation trees. (updated CYK) */
CYK_ParseTableWithCounts(G, w) : (S,2),(A,2),(C,1)
                                 -                  (S,2),(A,2),(C,1)
                                 -                  (B,1)                (B,1)
                                 (S,1),(A,1)        (B,1)                (S,1),(C,1)          (S,1),(A,1)
                                 (B,1)              (A,1),(C,1)          (A,1),(C,1)          (B,1)                (A,1),(C,1)
CYK_ParseTreeCount(G, w) : 2

------------------------------------------------------------------------------------------------------------------------

/* INPUT(G, w) : context-free grammar in Chomsky normal form and word
 * G (context-free grammar), N (nonterminals), T (terminals), R (rules), S (start)
 * w (word) */
G = (N, T, R, S)
N = [S, AA, A]
T = [ε, a]
R : S → AA | ε
    A → AA | a
S = "S"
w = "aaaa"

/* OUTPUT(bool) : At least 1 derivation tree? (original CYK)
 * CYK (Cocke-Younger-Kasami algorithm) */
CYK_ParseTable(G, w) : S,A
                       S,A            S,A
                       S,A            S,A                  S,A
                       A              A                    A                    A
CYK(G, w) : True

/* OUTPUT(uint) : Count of different derivation trees. (updated CYK) */
CYK_ParseTableWithCounts(G, w) : (S,5),(A,5)
                                 (S,2),(A,2)        (S,2),(A,2)
                                 (S,1),(A,1)        (S,1),(A,1)          (S,1),(A,1)
                                 (A,1)              (A,1)                (A,1)                (A,1)
CYK_ParseTreeCount(G, w) : 5

------------------------------------------------------------------------------------------------------------------------

/* INPUT(G, w) : context-free grammar in Chomsky normal form and word
 * G (context-free grammar), N (nonterminals), T (terminals), R (rules), S (start)
 * w (word) */
G = (N, T, R, S)
N = [S, AA, A]
T = [ε, a]
R : S → AA | ε
    A → AA | a
S = "S"
w = ""

/* OUTPUT(bool) : At least 1 derivation tree? (original CYK)
 * CYK (Cocke-Younger-Kasami algorithm) */
CYK_ParseTable(G, w) : S
CYK(G, w) : True

/* OUTPUT(uint) : Count of different derivation trees. (updated CYK) */
CYK_ParseTableWithCounts(G, w) : (S,1)
CYK_ParseTreeCount(G, w) : 1

------------------------------------------------------------------------------------------------------------------------

/* INPUT(G, w) : context-free grammar in Chomsky normal form and word
 * G (context-free grammar), N (nonterminals), T (terminals), R (rules), S (start)
 * w (word) */
G = (N, T, R, S)
N = [S, AA, A]
T = [ε, a]
R : S → AA | ε
    A → AA | a
S = "S"
w = "b"

/* OUTPUT(bool) : At least 1 derivation tree? (original CYK)
 * CYK (Cocke-Younger-Kasami algorithm) */
CYK_ParseTable(G, w) : -
CYK(G, w) : False

/* OUTPUT(uint) : Count of different derivation trees. (updated CYK) */
CYK_ParseTableWithCounts(G, w) : -
CYK_ParseTreeCount(G, w) : 0
```

Obr. 11 - Průběh programu po zadání několika různých příkladů

Zdroje:

[1] Tutorial #15: Parsing I: context-free grammars and the CYK algorithm [online]. [cit. 2021-12-09]. Dostupné z: https://www.borealisai.com/en/blog/tutorial-15-parsing-i-context-free-grammars-and-cyk-algorithm/
[2] CYK Parsing Algorithm [online]. [cit. 2021-12-09]. Dostupné z: https://www.cs.bgu.ac.il/~michaluz/seminar/CKY1.pdf