SPRAWOZDANIE		Data wykonania: 15/12/2018
Tytuł zadania:	**Wykonał:**	**Sprawdził:**
Gramatyka Bezkontekstowa	*Krzysztof Maciejewski* *299262*	*dr inż. Konrad Markowski*

Spis treści

1. Cel projektu

Celem ćwiczenia było napisanie programu wypisującego wyrazy z języka opisanego daną gramatyką bezkontekstową. Każdy student dostał indywidualne zadanie z inną gramatyką bezkontekstową. Program miał wyświetlać konkretną liczbę wyrazów, podaną przy wywołaniu, zaś wyrazy miały być uporządkowane w postaci kanonicznej.

Zadanie 23

Napisać program, który wypisze na ekranie n łańcuchów z języka opisanego za pomocą gramatyki bezkontekstowej:

S→aB|bA A→a|As|bAA B→b|bS|aBB

Program powinien wyświetlać opis gramatyki bezkontekstowej, a wypisywane łańcuchy powinny być uporządkowane w postaci kanonicznej.

2. Teoria

Gramatyka bezkontekstowa jest gramatyką formalną, w której wszystkie reguły wyprowadzania wyrażeń są postaci:

$$A \to T,$$

gdzie:

A to dowolny symbol nieterminalny, jego znaczenie nie zależy od kontekstu, w jakim występuje,

T to dowolny ciąg symboli terminalnych i nieterminalnych.

Gramatykę bezkontekstową możemy zapisać jako **uporządkowaną czwórkę** (T, N, P, S):

T jest skończonym zbiorem symboli terminalnych,

N jest skończonym zbiorem symboli nieterminalnych,

P jest skończonym zbiorem reguł,

$S \in N$ jest symbolem startowym.

Symbol nieterminalny to symbol, który można definiować. Symbole te umożliwiają tworzenie ciągów zawierających kombinacje symboli terminalnych i nieterminalnych.

Symbol terminalny to symbol elementarny tworzący wyrazy języka formalnego – jest równoważny symbolowi alfabetu języka. Symbole te pozostają w wyprowadzonym słowie – w przeciwieństwie do symboli nieterminalnych używanych tylko podczas wyprowadzania słowa.

Symbol startowy to wyróżniony symbol nieterminalny, od którego zaczyna się wyprowadzanie wszystkich wyrazów języka.

3. Szczegóły implementacyjne

Program przechowuje wyraz jako listę kolejnych symboli.

```
typedef struct lista {//nasz wyraz zapiszemy jako listę kolejnych znaków
        char litera;
        struct lista *next;
} symbol;
```

Przy pomocy funkcji **search_nt** program szuka pierwszego elementu listy będącego symbolem nieterminalnym i zwraca wskaźnik na niego.

```
symbol *search_nt(symbol *lista){
        /*szuka na liście pierwszego elementu z symbolem nieterminalnym (A/B/S)
         *zwraca ten element lub ostatni, jeżeli nie znajdzie wcześniej takiego symbolu*/
        symbol *wsk = lista;
        while (wsk->litera!='A'&&wsk->litera!='B'&&wsk->litera!='S'&&wsk->next!=NULL)
                wsk=wsk->next;
        return wsk;
}
```

Następnie, dzięki funkcjom **add**, możemy zamienić ten element (symbol nieterminalny) na inne elementy (symbole terminalne i nieterminalne, zgodnie z założeniami naszej gramatyki bezkontekstowej).

```
void add2(symbol *wsk, char znak1, char znak2){
        symbol *nowy2=malloc(sizeof(symbol));
        nowy2->next=wsk->next;
        nowy2->litera=znak2;
        wsk->next=nowy2;
        wsk->litera=znak1;
}
```

Najważniejszą funkcją w naszym programie jest funkcja **generuj**, która tworzy i wypisuje nam gotowy wyraz, wraz ze ścieżką jego uzyskania. Funkcja pobiera *limit* jako argument, co pozwala manipulować długością wyrazu.

W funkcji, jako pierwszy element listy definiujemy nasz symbol startowy S.

```
void generuj(int limit){//generator wyrazu, argument określa nam limit

        symbol *first=malloc(sizeof(symbol));
        first->litera='S';
        first->next=NULL;
        printf("%c", first->litera);
```

Tworzymy pętle, która wykonuje się dopóki nasza funkcja **search_nt** znajduje symbol nieterminalny.

```
symbol *wsk;
while ((wsk=search_nt(first))!=NULL){//dopóki znajdujemy znak A,B lub S, wykonuj pętle
```

Warto wspomnieć również o zmiennej *cnt*. Służy ona do liczenia z ilu symboli będzie składał się wyraz, jeżeli postanowimy zakończyć go w najkrótszy możliwy sposób (zgodnie z założeniami gramatyki - każde A na a, każde B na b). Jeżeli *cnt* jest mniejsze od określonego jako argument funkcji limitu, łańcuch rośnie. Jeżeli natomiast *cnt* zrówna się lub przekroczy limit, to kończymy wyraz w najkrótszy możliwy sposób. Dzięki całej tej procedurze jesteśmy w stanie manipulować długością generowanych wyrazów i wypisywać je coraz dłuższe. Tym samym porządkujemy je kanonicznie, zgodnie z założeniami zadania.

```
if(cnt>=limit){//kończenie łańcucha w najkrótszy możliwy sposób

        if(wsk->litera=='S'){
                int a=rand()%2;
                switch (a) {
                        case 0: add2(wsk, 'a', 'B'); break;
                        case 1: add2(wsk, 'b', 'A'); break;}
                wyswietl(first);}

        else if(wsk->litera=='A'){i
                add1(wsk, 'a');
                wyswietl(first);}

        else if(wsk->litera=='B'){
                add1(wsk, 'b');
                wyswietl(first);}}



else if(wsk->litera=='A'){
        int a=rand()%2;
        switch (a) {
                case 0: add2(wsk, 'A', 's'); cnt+=1; break;
                case 1: add3(wsk, 'b', 'A', 'A'); cnt+=2; break;}
        wyswietl(first);}
```

Jeżeli mamy więcej niż jedną opcję na przekształcenie symbolu nieterminalnego, wybieramy ją losowo.

Po każdym przekształceniu symbolu nieterminalnego wypisujemy aktualny stan łańcucha na *stdout*. Dzięki temu uzyskujemy ścieżkę przekształceń i jesteśmy w stanie zobaczyć w jaki sposób powstał końcowy wyraz.

Na koniec pętli wracamy do poszukiwania symbolu terminalnego od początku listy.

```
wsk=search_nt(first);//rozpoczynamy poszukiwanie symbolu nieterm. od początku
```

W funkcji **main** pobieramy argument wywołania jako ilość wyrazów które chcemy wypisać. Przeprowadzamy kontrolę błędów – jeżeli podamy w argumencie symbole inne niż cyfry, zwróci błąd.

```
int main(int n, char **argv){

        if (n<2){
                fprintf(stderr,"Nie podano argumentu\n");
                return 1;}
        if (n>2){
                fprintf(stderr,"Podaj jeden argument!\n");
                return 2;}
        for(int i=0;i<(strlen(argv[1]));i++){
                int z=argv[1][i];
                if (z<48||z>57){
                        fprintf(stderr,"Zły argument\n");
                        return 3;}}
```

Następnie wyświetlamy opis naszej gramatyki i uruchamiamy pętlę generującą kolejne wyrazy.

```
int ile = atoi(argv[1]);
printf("Opis gramatyki bezkontekstowej: S->aB|bA  A->a|As|bAA  B->b|bS|aBB\n\n");
srand(time(NULL));
for (int i=1; i<=ile; i++)
        generuj(i*2);
```

4. Sposób wywołania programu

Przy wywoływaniu programu należy podać jeden argument – ilość wyrazów, które chcemy wygenerować. Jeżeli podamy więcej argumentów lub nie podamy go w ogóle, program zwróci błąd. Jeżeli w argumencie podamy coś innego niż cyfry, program również zwróci błąd.

Przykłady wywołania programu:

4.1. Brak argumentu wywołania

```
root@kristoph:~# ./a.out
Nie podano argumentu
root@kristoph:~#
```

4.2. Więcej niż jeden argument wywołania

```
root@kristoph:~# ./a.out 2 3
Podaj jeden argument!
root@kristoph:~# ./a.out a 1
Podaj jeden argument!
root@kristoph:~#
```

4.3. Nieprawidłowy argument wywołania

```
root@kristoph:~# ./a.out 2,5
Zły argument
root@kristoph:~# ./a.out a
Zły argument
root@kristoph:~# ./a.out -5
Zły argument
root@kristoph:~# ./a.out 3.0
Zły argument
root@kristoph:~#
```

4.4. Prawidłowy argument wywołania

```
root@kristoph:~# ./a.out 3
Opis gramatyki bezkontekstowej: S->aB|bA  A->a|As|bAA  B->b|bS|aBB

S -> aB -> ab

S -> aB -> aaBB -> aabB -> aabb

S -> aB -> aaBB -> aaaBBB -> aaabBB -> aaabbB -> aaabbb

root@kristoph:~# ./a.out 0
Opis gramatyki bezkontekstowej: S->aB|bA  A->a|As|bAA  B->b|bS|aBB

root@kristoph:~#
```

5. Wnioski i spostrzeżenia

Napisanie tego programu było sporym wyzwaniem. Było to zadanie niewątpliwie trudniejsze od poprzedniego, związanego z automatem skończonym. Zmusiło mnie do głębszego zapoznania się z tematem gramatyk bezkontekstowych, ale też pomogło lepiej ten temat zrozumieć.

Najtrudniejszą częścią zadania było dla mnie uporządkowanie kanoniczne wyrazów. Jako, że zdecydowałem się wyświetlać cała ścieżkę tworzenia wyrazu, przechowywanie oraz sortowanie wyrazów byłoby dość kłopotliwe. W tym wypadku postanowiłem skorzystać z metody generowania coraz dłuższych wyrazów. Mimo, że rozwiązanie to wydaje się być dosyć banalne, stanowi najbardziej praktyczne rozwiązanie tego problemu.

Ogółem, jestem bardzo zadowolony z napisanego programu. Spełnia on wszystkie założenia zadania, jest zabezpieczony przed błędnymi argumentami wywołania, korzysta z dynamicznej alokacji pamięci. Wyświetlanie ścieżki tworzenia wyrazu stanowi praktyczny dodatek, który pozwala zobaczyć w jaki sposób powstał wyraz oraz dowodzi, że program działa prawidłowo.