SPRAWOZDANIE		Data wykonania: 14/12/2018
Tytuł zadania:	**Wykonał:**	**Sprawdził:**
Gramatyka bezkontekstowa	*Kamil Sztandur 307354*	*dr inż. Konrad Markowski*

Spis treści

Cel projektu

Celem tego projektu było zaprogramowanie programu symulującego działanie maszyny generującej wybraną przez użytkownika ilość łańcuchów z wybranego języka gramatyki bezkontekstowej w postaci kanonicznej.

Zadanie 13

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

S→aSa|ε

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

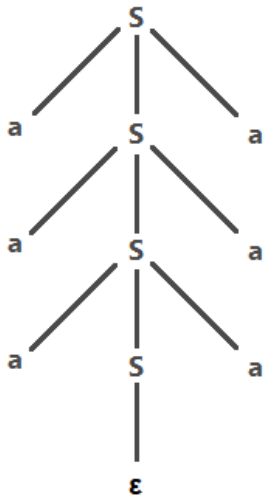
Teoria

Języki bezkontekstowa w przeciwieństwie do języków regularnych generują poprawne napisy poprzez sekwencję przepisywań, która ma strukturę drzewa. Ze względu na tę właściwość świetnie nadają się do opisu syntaktyki języków programowania.

Każda gramatyka bezkontekstowa ma postać G = (V, T, P, S), gdzie V to skończony zbiór zmiennych, T skończony zbiór symboli końcowych, P to skończony zbiór produkcji , a S to symbol początkowy. Język generowany przez ten program można zapisać w uproszczonej postaci S → aSa|ε, a jej język $\{ a^n * a^n \mid n \geq 0 \}$. Z deklaracji tej gramatyki można wywnioskować, że jej łańcuchy będą palindromami. Oznacza to, że odczytane od prawej do lewej będą takie same jak odczytane od lewej do prawej. Przykładami polskich wyrazów będącymi palindromami są np. anna czy kajak. Wyprowadzenie przykładowego łańcucha z naszej gramatyki wygląda następująco:

S → aSa → aaSaa → aaaSaaa →aaaaaa

Gdzie S, to symbol pusty zadeklarowany wraz z deklaracją gramatyki. Jak wspomniałem łańcuchy gramatyki bezkontekstowej generuje się poprzez sekwencję przepisywań, która ma strukturę drzewa. Drzewo powyższego łańcucha wygląda następująco:



Szczegóły implementacyjne

W czasie pracy nad programem zadbałem, aby był on przyjazny każdemu użytkownikowi. Reaguje on inaczej zależnie od treści podanych przez użytkownika. W przypadku czystego wywołania programów bez podawania żadnych parametrów program „przedstawia się” i zachęca użytkownika do dodania parametru -h, -help lub -?, aby wyświetlić instrukcję korzystania z programu. Analogiczna reakcja następuje w przypadku podania nieprawidłowej liczby łańcuchów np. wpisania litery lub znaku zamiast liczby. Program informuje wówczas użytkownika, że podana przez niego wartość nie jest liczbą i również zachęca do wyświetlenia instrukcji help.

```
if( argc < 2 ) {  /* Jezeli uzytkownik nie wpisze zadnego argumentu */
        printf(".-------------------------(GBK GENERATOR)-------------------------.\n");
        printf("\n");
        printf("              Zaimplementowana gramatyka:  S->aSa|Îµ\n");
        printf("             Obslugiwany jezyk: { a^n a^n | n >= 0 }.\n");
        printf("\n");
        printf("        Dodaj parametr -h / -help / -? aby wyswietlic Pomoc.\n");
        printf("\n");
        printf("*------------------------------------------------------------------*\n");
        return 0;
}

printf(".-------------------------(GBK GENERATOR)-------------------------.\n");
printf("\n");
printf("               Zaimplementowana gramatyka:  S->aSa|Îµ\n");
printf("              Obslugiwany jezyk: { a^n a^n | n >= 0 }.\n");
printf("\n");
printf("*------------------------------------------------------------------*\n\n");
if( strcmp(argv[1], "-help") == 0 || strcmp(argv[1], "-h") == 0 || strcmp(argv[1], "-?") == 0 ) help();

if( atoi( argv[1] ) == 0 ) { /* Jezeli uzytkownik poda jako pierwszy parametr co innego, niz liczba */
        printf("Nieprawidlowe uzycie programu (%s to nie liczba calkowita).\n", argv[1] );
        printf("Dodaj parametr -h / -help / -? aby wyswietlic Pomoc.\n\n");
        return 1;
}
```

Jeżeli na tym etapie wszystko przebiegło pomyślnie, to program przechodzi do generowania żądanej ilości łańcuchów oraz wypisania drzewa najdłuższego z nich. Zacznijmy od sposobu generowania łańcucha. Nasza gramatyka bezkontekstowa ma postać palindromu „S → aSa|ε” zatem musimy znaleźć pewien powtarzający się schemat, który ułatwi naszemu symulatorowi wypisywanie kolejnych łańcuchów w postaci kanonicznej. Najłatwiej jest dojść do tego poprzez ręczne wypisywanie kolejnych wyrazów długopisem na kartce do skutku aż spostrzeżemy pewien schemat wypisywania kolejnych wyrazów. Chciałem, aby mój program wypisywał te łańcuchy w postaci kanonicznej wraz z „historią” wyprowadzenia każdego z nich (jak w przykładzie w zakładce Teoria).

Zauważyłem, że w każdej linijce znajduje się n wyrazów, gdzie n to numer aktualnej linijki, a ilość symboli po każdej stronie symbolu nieoznaczonego S jest równa n-1 w tym wyrazie. Pozostaje kwestia wypisywania historii wyprowadzenia n-tego wyrazu w n-tej linijce. Można to zrobić za pomocą pętli for, która wypisze kolejne wyrazy z j-tą ilością symboli po każdej stronie S dopóki j nie będzie równe numerowi aktualnej linijki. Następnie z ostatecznego łańcucha usuwa się symbol pusty S lub zastępuje się go epsilonem, jeżeli jego długość ma być 1.

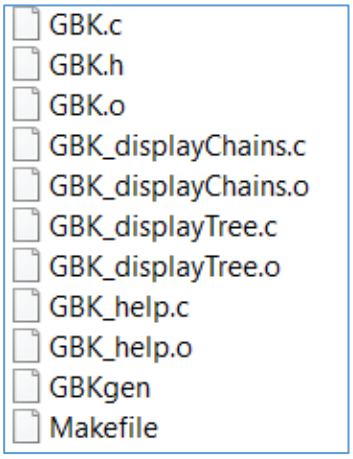
Zilustrowanie tego schematu w postaci kodu wygląda następująco:

```
for( int i = 0; i < n+1 ; i++ ) {
        /* Tutaj obslugujemy kolejne linijki */

        for( int j = 0; j < i; j++) {
                /* Tutaj obslugujemy pojedynczy wyraz */

                for( int a = 0; a < nr_wyrazu; a++)
                        printf("a");
                printf("S");
                for(int a = 0; a < nr_wyrazu; a++)
                        printf("a");
                printf(" -> ");
                if( i == 1) printf(" ε ");

                nr_wyrazu++;
        }

        for( int k = 0; k < 2*(i-1); k++ )
                printf("a");

        if( i != n ) printf("\n");
        nr_wyrazu = 0;
}
```

Następnym zadaniem symulatora jest wypisanie drzewa wyprowadzenia najdłuższego łańcucha. Tutaj również ważne jest zauważenie schematu wypisywania. Po przyjrzeniu się wypisanym drzewom kilku łańcuchów z tego język można zauważyć, że będzie miało ono n+1 kondygnacji z czego n kondygnacji powstanie dzięki instrukcji wypisywania, a ostatnia, n+1, będzie zawierać sam symbol pusty ε. Oczywiście, „n” to liczba żądanych łańcuchów wprowadzona przez użytkownika. Wymyślona przeze mnie instrukcja mogłaby działać wadliwie dla n=1, więc musiałem uwzględnić ten wyjątek w kodzie.

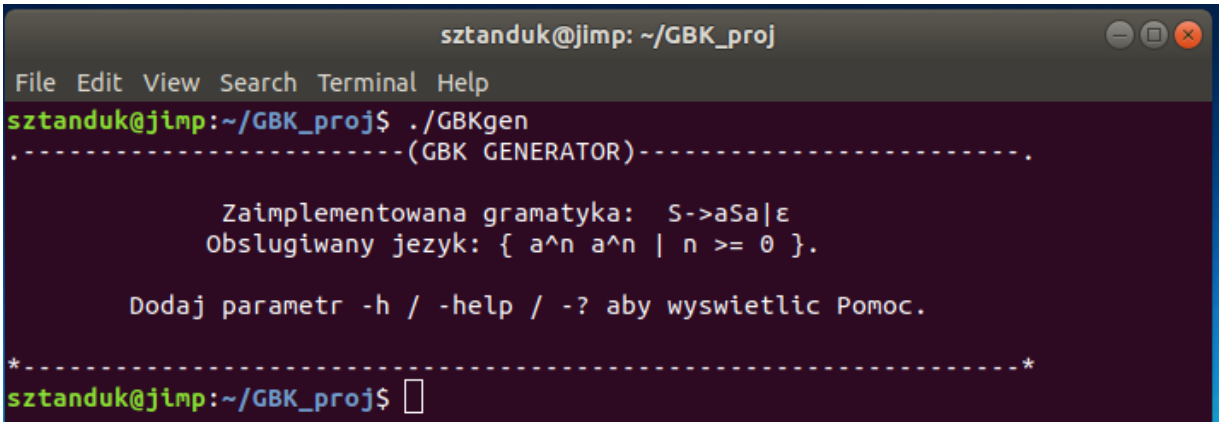
```
printf("       S\n");
if( n == 1 ) printf("       |\n");
else {
        for( int i = 0; i < n; i++) {
                if( i != 0 ) printf("       S\n");
                if( i != n-1 ) {
                        printf("      /|\\ \n");
                        printf("     / | \\ \n");
                        printf("    a  |  a\n");
                } else {
                        printf("       |\n");
                        printf("       |\n");
                }
        }
}

printf("       ε\n\n");
```

W czasie pracy na projektem starałem się zadbać o czytelność i przystępność programu w przyszłych modyfikacjach. Każda funkcja znajduje się w oddzielnym pliku, dzięki czemu nad programem może pracować równocześnie wielu programistów. Deklarację każdej z funkcji zawarłem w bibliotece GBK.h, co pilnuje kompatybilności każdej funkcji z programem na wypadek nieprawidłowej jej modyfikacji przez programistę. Całość kompilujemy za pomocą Makefile, co jest bardzo wygodne i oszczędne w zasobach. Zadbałem również o czystość i przejrzystość samego kodu, starannie dobierając nazwy funkcji i zmiennych oraz umieszczając komentarze w bardziej skomplikowanych fragmentach kodu, co usprawni przyszłą pracę nad nim.

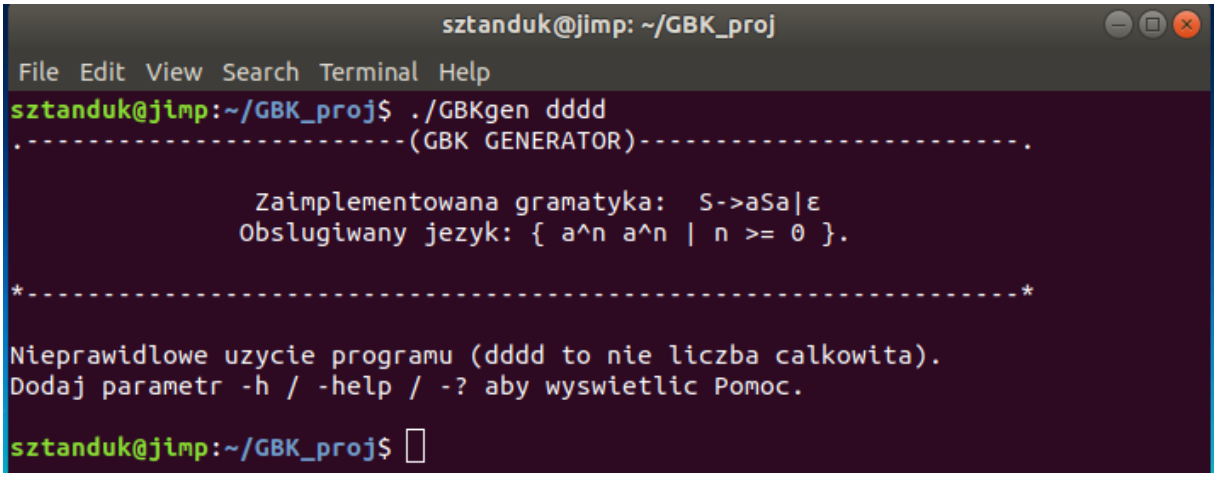


Sposób wywołania programu

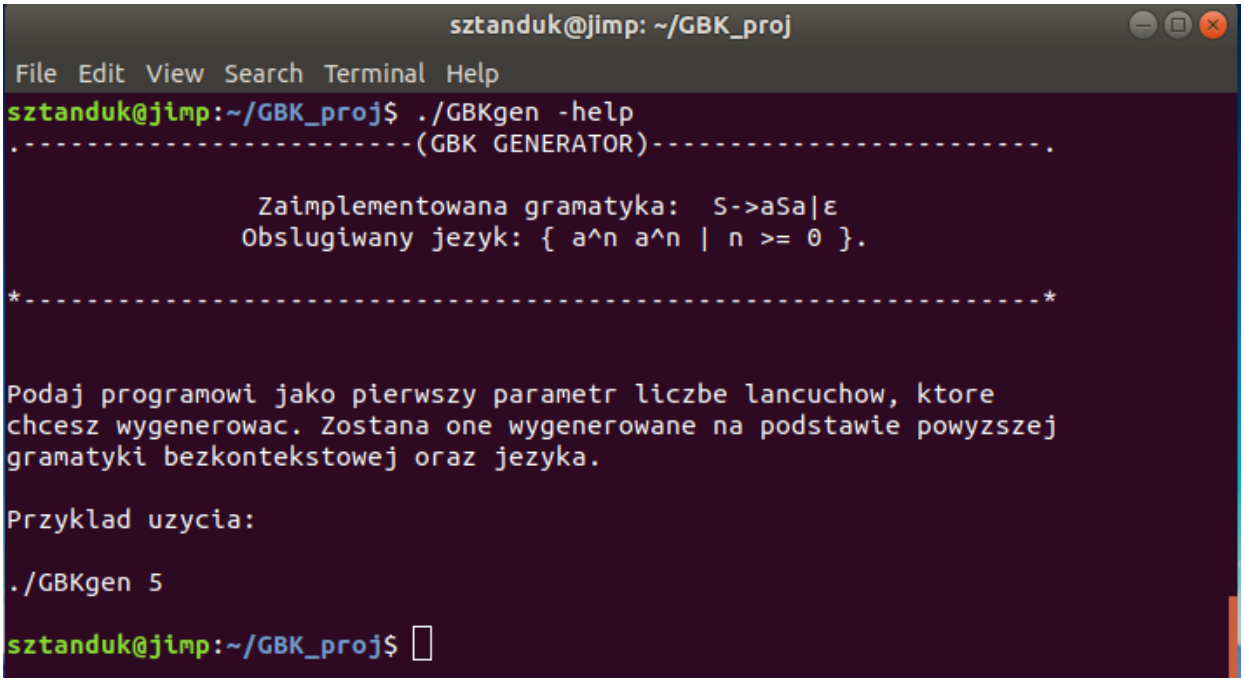
Po skompilowaniu programu instrukcją Makefile uruchamiamy go komendą ./GBKgen. Wówczas program przedstawi się i zachęci użytkownika do wyświetlenia instrukcji help, dodając parametr -h, -help lub -?. **Program obsługuje tylko podawanie argumentów poprzez stdin.**



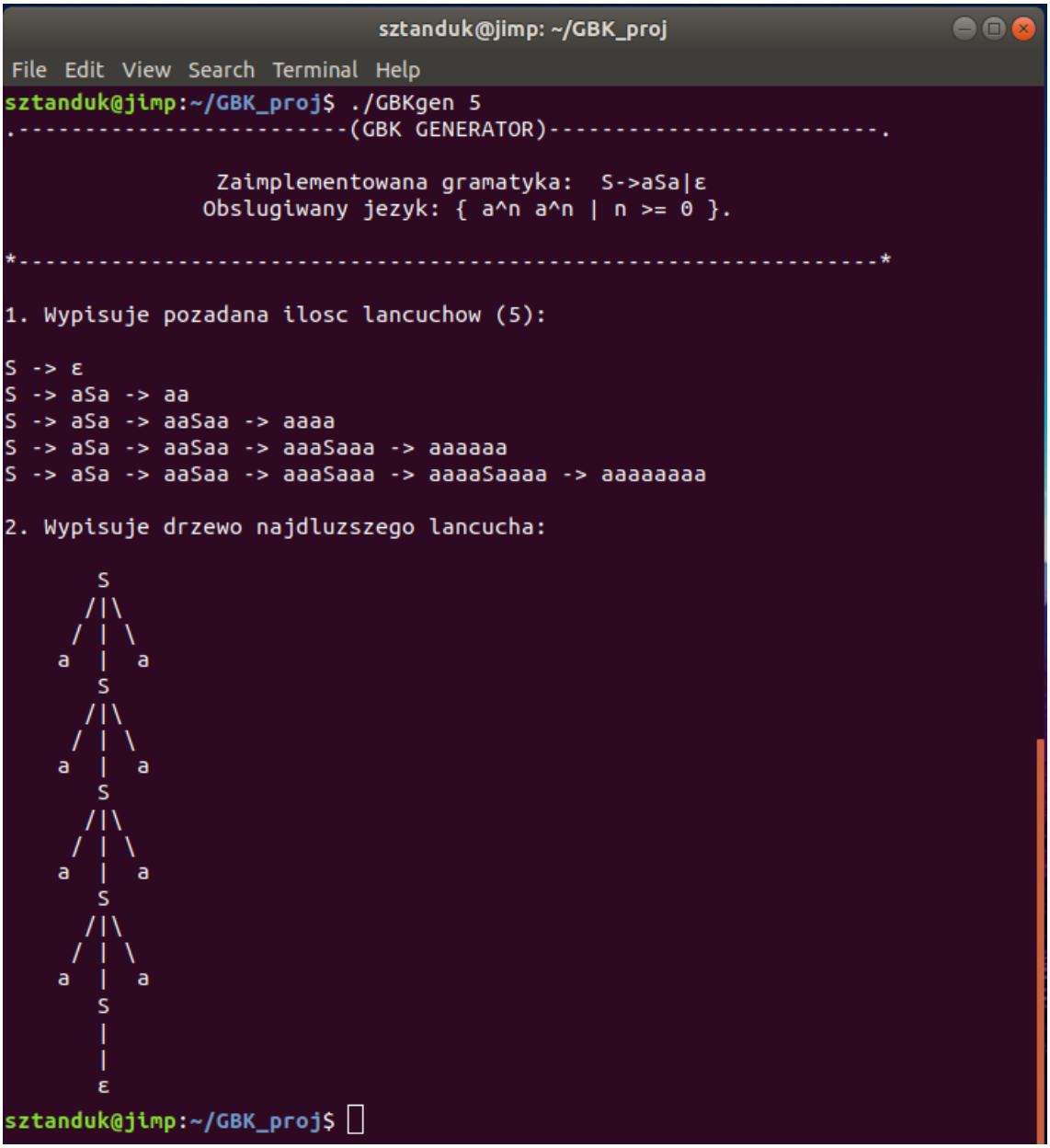
W ramach obsługi błędów podobna sytuacja ma miejsce, gdy użytkownik poda nieprawidłową liczbę łańcuchów. Wówczas program wypisuje błąd, wyświetla wpisaną przez użytkownika wartość i informuje go, że nie jest ona liczbą całkowitą, a następnie poleca mu wyświetlenie instrukcji help. Algorytm może działać nieprawidłowo, jeżeli użytkownik zażąda zbyt dużej liczby łańcuchów. Podana liczba n jest zapisywana w zmiennej typu int, więc nie obsługuje liczb większych niż jest w stanie pomieścić ta zmienna.



A tak prezentuje się instrukcja help:



W przypadku prawidłowego wywołania programu, czyli np. ‘’./GBKgen 5’’, wyświetli on swoją nazwę, obsługiwaną gramatykę, język i autora. Program wyświetli liczbę podaną przez użytkownika i wypisze pożądaną ilość łańcuchów w postaci kanonicznej wraz z historią wyprowadzenia każdego z nich. Następnie program wyświetli drzewo najdłuższego łańcucha. Po wykonanej instrukcji program zakończy działanie, nie zwracając żadnego błędu.



Wnioski i spostrzeżenia

Poprawne wykonanie tego projektu wymagało ode mnie nauczenia się gramatyk bezkontekstowych oraz zrozumienia sposobu ich wypisywania. Bardzo zależało mi na usprawnieniu programu tak, aby był on przyjazny zarówno dla wszystkich użytkowników jak i programistów, którzy w przyszłości mogą chcieć modyfikować czy rozszerzać mój kod. Jestem z tego najbardziej zadowolony.

Program jest bardzo przyjazny dla zielonych użytkowników dzięki czytelnemu interfejsowi tekstowemu, wbudowanej instrukcji help oraz sprawnego jej sugerowania i wyświetlania zależnie od złego sposobu wywoływania programu przez użytkownika. Dzięki temu szybko skoryguje on swoje błędy. Kod programu jest rozbity na moduły i podprogramy, nad których kompatybilnością czuwa stworzona przeze mnie biblioteka. Starałem się, aby każdy moduł i funkcja był maksymalnie niezależny od całej reszty programu oraz czytelny. Dzięki temu program jest przyjazny wielu równoczesnym modyfikacjom i na bieżąco sprawdza ich poprawność, co uprzyjemni pracę nad nim w przyszłości.

Wszystko to pomogło mi podnieść swoje umiejętności programisty oraz zwiększyć znajomość języka C. Cieszę się też ze znalezienia łatwego i przyjemnego do implementacji sposobu wyświetlania łańcuchów oraz ich drzew dzięki ręcznemu wypisywaniu ich na kartce aż do skutku. Choć zdaję sobie sprawę, że nie jest to zbyt skomplikowana gramatyka, to właśnie doświadczenia płynące z takich małych rzeczy później decydują, czy poradzimy sobie w tych większych.